Document Number: MCUXSDKAPIRM
Rev 2.14.0
Jul 2023

MCUXpresso SDK API Reference Manual

NXP Semiconductors

Contents

Section No. Title Page No.

Chapter 1 Introduction

The MCUXpresso Software Development Kit (MCUXpresso SDK) is a collection of software enablement for NXP Microcontrollers that includes peripheral drivers, multicore support and integrated RTOS support for FreeRTOS^{TM} . In addition to the base enablement, the MCUXpresso SDK is augmented with demo applications, driver example projects, and API documentation to help users quickly leverage the support provided by MCUXpresso SDK. The MCUXpresso SDK Web Builder is available to provide access to all MCUXpresso SDK packages. See the *MCUXpresso Software Development Kit (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.

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

MCUXpresso SDK Folder Structure

Chapter 2
Trademarks

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

How to Reach Us:

Home Page: nxp.com

Web Support: nxp.com/support

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

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

© 2021 NXP B.V.

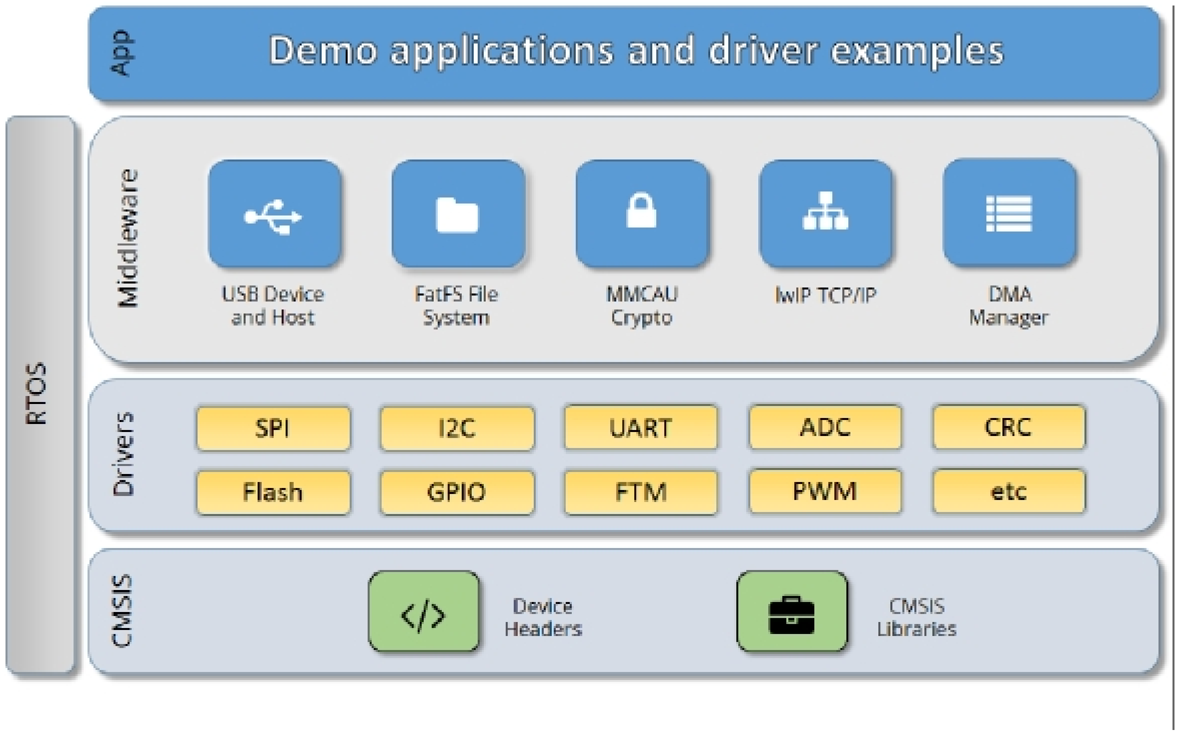
Chapter 3
Architectural Overview

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

Overview

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

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



MCUXpresso SDK Block Diagram

MCU header files

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

CMSIS Support

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

MCUXpresso SDK Peripheral Drivers

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

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

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

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

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

Interrupt handling for transactional APIs

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

```
        PUBWEAK SPI0_IRQHandler
        PUBWEAK SPI0_DriverIRQHandler
SPI0_IRQHandler
```

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

The first level of the weak implementation are the functions defined in the vector table. In the devices/<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
Clock Driver

4.1 Overview

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

The clock driver supports:

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

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

4.2 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

4.2.1 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.

4.2.2 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.

4.2.3 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.

4.2.4 SYSCON flash wait states

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

4.3 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

Macros

- #define CLOCK_FRO_SETTING_API_ROM_ADDRESS (0x0F001CD3U)
  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 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 MRT_CLOCKS
  Clock ip name array for MRT.
- #define WKT_CLOCKS
  Clock ip name array for WKT.
- #define PLU_CLOCKS
  Clock ip name array for PLU.
- #define CLK_GATE_DEFINE(reg, bit) ((((reg)&0xFFU) << 8U) | ((bit)&0xFFU))
  Internal used Clock definition only.

Enumerations

- enum clock_ip_name_t {
  kCLOCK_IpInvalid = 0U,
  kCLOCK_Sys = CLK_GATE_DEFINE(SYS_AHB_CLK_CTRL0, 0U),
  kCLOCK_Rom = CLK_GATE_DEFINE(SYS_AHB_CLK_CTRL0, 1U),
  kCLOCK_Ram0 = CLK_GATE_DEFINE(SYS_AHB_CLK_CTRL0, 2U),
  kCLOCK_Flash = CLK_GATE_DEFINE(SYS_AHB_CLK_CTRL0, 4U),
  kCLOCK_I2c0 = CLK_GATE_DEFINE(SYS_AHB_CLK_CTRL0, 5U),
  kCLOCK_Gpio0 = CLK_GATE_DEFINE(SYS_AHB_CLK_CTRL0, 6U),
  kCLOCK_Swm = CLK_GATE_DEFINE(SYS_AHB_CLK_CTRL0, 7U),
  kCLOCK_Wkt = CLK_GATE_DEFINE(SYS_AHB_CLK_CTRL0, 9U),
  kCLOCK_Mrt = CLK_GATE_DEFINE(SYS_AHB_CLK_CTRL0, 10U),
  kCLOCK_Spi0 = CLK_GATE_DEFINE(SYS_AHB_CLK_CTRL0, 11U),
  kCLOCK_Crc = CLK_GATE_DEFINE(SYS_AHB_CLK_CTRL0, 13U),
  kCLOCK_Uart0 = CLK_GATE_DEFINE(SYS_AHB_CLK_CTRL0, 14U),
  kCLOCK_Uart1 = CLK_GATE_DEFINE(SYS_AHB_CLK_CTRL0, 15U),
  kCLOCK_Wwdt = CLK_GATE_DEFINE(SYS_AHB_CLK_CTRL0, 17U),
  kCLOCK_Iocon = CLK_GATE_DEFINE(SYS_AHB_CLK_CTRL0, 18U),
  kCLOCK_Acmp = CLK_GATE_DEFINE(SYS_AHB_CLK_CTRL0, 19U),
  kCLOCK_I2c1 = CLK_GATE_DEFINE(SYS_AHB_CLK_CTRL0, 21U),
  kCLOCK_Adc = CLK_GATE_DEFINE(SYS_AHB_CLK_CTRL0, 24U),
  kCLOCK_Ctimer0 = CLK_GATE_DEFINE(SYS_AHB_CLK_CTRL0, 25U),
  kCLOCK_Dac = CLK_GATE_DEFINE(SYS_AHB_CLK_CTRL0, 27U),
  kCLOCK_GpioInt = CLK_GATE_DEFINE(SYS_AHB_CLK_CTRL0, 28U),
  kCLOCK_Capt = CLK_GATE_DEFINE(SYS_AHB_CLK_CTRL1, 0U),
  kCLOCK_PLU = CLK_GATE_DEFINE(SYS_AHB_CLK_CTRL1, 5U) }

  Clock gate name used for CLOCK_EnableClock/CLOCK_DisableClock.
- enum clock_name_t {
  kCLOCK_CoreSysClk,
  kCLOCK_MainClk,
  kCLOCK_Fro,
  kCLOCK_FroDiv,
  kCLOCK_ExtClk,
  kCLOCK_LPOsc,
  kCLOCK_Frg0 }

  Clock name used to get clock frequency.
- enum clock_select_t {

kCAPT_Clk_From_Fro = CLK_MUX_DEFINE(CAPTCLKSEL, 0U),
kCAPT_Clk_From_MainClk = CLK_MUX_DEFINE(CAPTCLKSEL, 1U),
kCAPT_Clk_From_Fro_Div = CLK_MUX_DEFINE(CAPTCLKSEL, 3U),
kCAPT_Clk_From_LPOsc = CLK_MUX_DEFINE(CAPTCLKSEL, 4U),
kADC_Clk_From_Fro = CLK_MUX_DEFINE(ADCCLKSEL, 0U),
kADC_Clk_From_Extclk = CLK_MUX_DEFINE(ADCCLKSEL, 1U),
kUART0_Clk_From_Fro = CLK_MUX_DEFINE(UART0CLKSEL, 0U),
kUART0_Clk_From_MainClk = CLK_MUX_DEFINE(UART0CLKSEL, 1U),
kUART0_Clk_From_Frg0Clk = CLK_MUX_DEFINE(UART0CLKSEL, 2U),
kUART0_Clk_From_Fro_Div = CLK_MUX_DEFINE(UART0CLKSEL, 4U),
kUART1_Clk_From_Fro = CLK_MUX_DEFINE(UART1CLKSEL, 0U),
kUART1_Clk_From_MainClk = CLK_MUX_DEFINE(UART1CLKSEL, 1U),
kUART1_Clk_From_Frg0Clk = CLK_MUX_DEFINE(UART1CLKSEL, 2U),
kUART1_Clk_From_Fro_Div = CLK_MUX_DEFINE(UART1CLKSEL, 4U),
kI2C0_Clk_From_Fro = CLK_MUX_DEFINE(I2C0CLKSEL, 0U),
kI2C0_Clk_From_MainClk = CLK_MUX_DEFINE(I2C0CLKSEL, 1U),
kI2C0_Clk_From_Frg0Clk = CLK_MUX_DEFINE(I2C0CLKSEL, 2U),
kI2C0_Clk_From_Fro_Div = CLK_MUX_DEFINE(I2C0CLKSEL, 4U),
kI2C1_Clk_From_Fro = CLK_MUX_DEFINE(I2C1CLKSEL, 0U),
kI2C1_Clk_From_MainClk = CLK_MUX_DEFINE(I2C1CLKSEL, 1U),
kI2C1_Clk_From_Frg0Clk = CLK_MUX_DEFINE(I2C1CLKSEL, 2U),
kI2C1_Clk_From_Fro_Div = CLK_MUX_DEFINE(I2C1CLKSEL, 4U),
kSPI0_Clk_From_Fro = CLK_MUX_DEFINE(SPI0CLKSEL, 0U),
kSPI0_Clk_From_MainClk = CLK_MUX_DEFINE(SPI0CLKSEL, 1U),
kSPI0_Clk_From_Frg0Clk = CLK_MUX_DEFINE(SPI0CLKSEL, 2U),
kSPI0_Clk_From_Fro_Div = CLK_MUX_DEFINE(SPI0CLKSEL, 4U),
kFRG0_Clk_From_Fro = CLK_MUX_DEFINE(FRG[0].FRGCLKSEL, 0U),
kFRG0_Clk_From_MainClk = CLK_MUX_DEFINE(FRG[0].FRGCLKSEL, 1U),
kCLKOUT_From_Fro = CLK_MUX_DEFINE(CLKOUTSEL, 0U),
kCLKOUT_From_MainClk = CLK_MUX_DEFINE(CLKOUTSEL, 1U),
kCLKOUT_From_ExtClk = CLK_MUX_DEFINE(CLKOUTSEL, 3U),
kCLKOUT_From_Lposc = CLK_MUX_DEFINE(CLKOUTSEL, 4U) }

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 {
kCLOCK_DivAhbClk = CLK_DIV_DEFINE(SYSAHBCLKDIV),
kCLOCK_DivAdcClk = CLK_DIV_DEFINE(ADCCLKDIV),
kCLOCK_DivClkOut = CLK_DIV_DEFINE(CLKOUTDIV) }

  Clock divider.
- enum clock_fro_osc_freq_t {
kCLOCK_FroOscOut18M = 18000U,
kCLOCK_FroOscOut24M = 24000U,
kCLOCK_FroOscOut30M = 30000U }

  fro output frequency source 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_MainClkSrcLPOsc = CLK_MAIN_CLK_MUX_DEFINE(2U, 0U),
kCLOCK_MainClkSrcFroDiv = CLK_MAIN_CLK_MUX_DEFINE(3U, 0U) }
PLL clock definition.

Variables

- volatile uint32_t g_LP_Osc_Freq
  lower power oscilltor clock frequency.
- volatile uint32_t g_Ext_Clk_Freq
  external clock frequency.
- volatile uint32_t g_Fro_Osc_Freq
  external clock frequency.

Driver version

- #define FSL_CLOCK_DRIVER_VERSION (MAKE_VERSION(2, 3, 3))
  CLOCK driver version 2.3.3.

Clock gate, mux, and divider.

PLL configuration structure

- 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.
- static void **CLOCK_SetFRGClkMul** (uint32_t ∗base, uint32_t mul)

Get frequency

Set FRO clock source

Parameters

src,please	reference _clock_fro_src definition.

- uint32_t CLOCK_GetFRG0ClkFreq (void)
  Return Frequency of FRG0 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_GetUart0ClkFreq (void)
  Get UART0 frequency.
- uint32_t CLOCK_GetUart1ClkFreq (void)
  Get UART1 frequency.
- uint32_t CLOCK_GetFreq (clock_name_t clockName)
  Return Frequency of selected clock.
- static uint32_t CLOCK_GetLPOscFreq (void)
  Get watch dog OSC frequency.
- static uint32_t CLOCK_GetExtClkFreq (void)
  Get external clock frequency.

Fractional clock operations

System PLL initialize.

Parameters

config	System PLL configurations.

- bool CLOCK_SetFRG0ClkFreq (uint32_t freq)
  Set FRG0 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.
- static void CLOCK_DeinitLpOsc (void)
  Deinit watch dog OSC.
- void CLOCK_SetFroOscFreq (clock_fro_osc_freq_t freq)
  Set FRO oscillator output frequency.

4.4 Macro Definition Documentation

4.4.1 #define FSL_CLOCK_DRIVER_VERSION (MAKE_VERSION(2, 3, 3))

4.4.2 #define CLOCK_FRO_SETTING_API_ROM_ADDRESS (0x0F001CD3U)

4.4.3 #define ADC_CLOCKS

Value:

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

4.4.4 #define ACMP_CLOCKS

Value:

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

4.4.5 #define DAC_CLOCKS

Value:

```
{               \
        kCLOCK_Dac, \
    }
```

4.4.6 #define SWM_CLOCKS

Value:

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

4.4.7 #define ROM_CLOCKS

Value:

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

4.4.8 #define SRAM_CLOCKS

Value:

```
{                \
        kCLOCK_Ram0, \
    }
```

4.4.9 #define IOCON_CLOCKS

Value:

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

4.4.10 #define GPIO_CLOCKS

Value:

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

4.4.11 #define GPIO_INT_CLOCKS

Value:

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

4.4.12 #define CRC_CLOCKS

Value:

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

4.4.13 #define WWDT_CLOCKS

Value:

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

4.4.14 #define SCT_CLOCKS

Value:

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

4.4.15 #define I2C_CLOCKS

Value:

```
{                              \
        kCLOCK_I2c0, kCLOCK_I2c1, \
    }
```

4.4.16 #define USART_CLOCKS

Value:

```
{                                \
        kCLOCK_Uart0, kCLOCK_Uart1, \
    }
```

4.4.17 #define SPI_CLOCKS

Value:

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

4.4.18 #define CAPT_CLOCKS

Value:

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

4.4.19 #define CTIMER_CLOCKS

Value:

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

4.4.20 #define MRT_CLOCKS

Value:

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

4.4.21 #define WKT_CLOCKS

Value:

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

4.4.22 #define PLU_CLOCKS

Value:

```
{                \
        kCLOCK_PLU, \
    }
```

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

4.5 Enumeration Type Documentation

4.5.1 enum clock_ip_name_t

Enumerator

kCLOCK_IpInvalid	Invalid Ip Name.

kCLOCK_Sys Clock gate name: Sys.
kCLOCK_Rom Clock gate name: Rom.
kCLOCK_Ram0 Clock gate name: Ram0.
kCLOCK_Flash Clock gate name: Flash.
kCLOCK_I2c0 Clock gate name: I2c0.
kCLOCK_Gpio0 Clock gate name: Gpio0.
kCLOCK_Swm Clock gate name: Swm.
kCLOCK_Wkt Clock gate name: Wkt.
kCLOCK_Mrt Clock gate name: Mrt.
kCLOCK_Spi0 Clock gate name: Spi0.
kCLOCK_Crc Clock gate name: Crc.
kCLOCK_Uart0 Clock gate name: Uart0.
kCLOCK_Uart1 Clock gate name: Uart1.
kCLOCK_Wwdt Clock gate name: Wwdt.
kCLOCK_Iocon Clock gate name: Iocon.
kCLOCK_Acmp Clock gate name: Acmp.
kCLOCK_I2c1 Clock gate name: I2c1.
kCLOCK_Adc Clock gate name: Adc.
kCLOCK_Ctimer0 Clock gate name: Ctimer0.
kCLOCK_Dac Clock gate name: Dac.
kCLOCK_GpioInt Clock gate name: GpioInt.
kCLOCK_Capt Clock gate name: Capt.
kCLOCK_PLU Clock gate name: PLU.

4.5.2 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_LPOsc Watchdog Oscillator.
kCLOCK_Frg0 fractional rate0

4.5.3 enum clock_select_t

Enumerator

kCAPT_Clk_From_Fro Mux CAPT_Clk from Fro.
kCAPT_Clk_From_MainClk Mux CAPT_Clk from MainClk.
kCAPT_Clk_From_Fro_Div Mux CAPT_Clk from Fro_Div.

kCAPT_Clk_From_LPOsc Mux CAPT_Clk from LPOsc.
kADC_Clk_From_Fro Mux ADC_Clk from Fro.
kADC_Clk_From_Extclk Mux ADC_Clk from Extclk.
kUART0_Clk_From_Fro Mux UART0_Clk from Fro.
kUART0_Clk_From_MainClk Mux UART0_Clk from MainClk.
kUART0_Clk_From_Frg0Clk Mux UART0_Clk from Frg0Clk.
kUART0_Clk_From_Fro_Div Mux UART0_Clk from Fro_Div.
kUART1_Clk_From_Fro Mux UART1_Clk from Fro.
kUART1_Clk_From_MainClk Mux UART1_Clk from MainClk.
kUART1_Clk_From_Frg0Clk Mux UART1_Clk from Frg0Clk.
kUART1_Clk_From_Fro_Div Mux UART1_Clk from Fro_Div.
kI2C0_Clk_From_Fro Mux I2C0_Clk from Fro.
kI2C0_Clk_From_MainClk Mux I2C0_Clk from MainClk.
kI2C0_Clk_From_Frg0Clk Mux I2C0_Clk from Frg0Clk.
kI2C0_Clk_From_Fro_Div Mux I2C0_Clk from Fro_Div.
kI2C1_Clk_From_Fro Mux I2C1_Clk from Fro.
kI2C1_Clk_From_MainClk Mux I2C1_Clk from MainClk.
kI2C1_Clk_From_Frg0Clk Mux I2C1_Clk from Frg0Clk.
kI2C1_Clk_From_Fro_Div Mux I2C1_Clk from Fro_Div.
kSPI0_Clk_From_Fro Mux SPI0_Clk from Fro.
kSPI0_Clk_From_MainClk Mux SPI0_Clk from MainClk.
kSPI0_Clk_From_Frg0Clk Mux SPI0_Clk from Frg0Clk.
kSPI0_Clk_From_Fro_Div Mux SPI0_Clk from Fro_Div.
kFRG0_Clk_From_Fro Mux FRG0_Clk from Fro.
kFRG0_Clk_From_MainClk Mux FRG0_Clk from MainClk.
kCLKOUT_From_Fro Mux CLKOUT from Fro.
kCLKOUT_From_MainClk Mux CLKOUT from MainClk.
kCLKOUT_From_ExtClk Mux CLKOUT from ExtClk.
kCLKOUT_From_Lposc Mux CLKOUT from Lposc.

4.5.4 enum clock_divider_t

Enumerator

kCLOCK_DivAhbClk Ahb Clock Divider.
kCLOCK_DivAdcClk Adc Clock Divider.
kCLOCK_DivClkOut Clk Out Divider.

4.5.5 enum clock_fro_osc_freq_t

fro oscillator output frequency value definition

Enumerator

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

4.5.6 enum clock_main_clk_src_t

< Main clock source definition

Enumerator

kCLOCK_MainClkSrcFro	main clock source from FRO
kCLOCK_MainClkSrcExtClk	main clock source from Ext clock
kCLOCK_MainClkSrcLPOsc	main clock source from lower power oscillator
kCLOCK_MainClkSrcFroDiv	main clock source from FRO Div

4.6 Function Documentation

4.6.1 void CLOCK_SetMainClkSrc (clock_main_clk_src_t *src*)

Parameters

src	Reference clock_main_clk_src_t to set the main clock source.

4.6.2 uint32_t CLOCK_GetFRG0ClkFreq (void)

Returns

Frequency of FRG0 Clock.

4.6.3 uint32_t CLOCK_GetMainClkFreq (void)

Returns

Frequency of Main Clock.

4.6.4 uint32_t CLOCK_GetFroFreq (void)

Returns

Frequency of FRO.

4.6.5 static uint32_t CLOCK_GetCoreSysClkFreq (void) `[inline]`, `[static]`

Returns

Frequency of core.

4.6.6 uint32_t CLOCK_GetClockOutClkFreq (void)

Returns

Frequency of ClockOut

4.6.7 uint32_t CLOCK_GetUart0ClkFreq (void)

Return values

UART0	frequency value.

4.6.8 uint32_t CLOCK_GetUart1ClkFreq (void)

Return values

UART1	frequency value.

4.6.9 uint32_t CLOCK_GetFreq (clock_name_t *clockName*)

Returns

Frequency of selected clock

4.6.10 static uint32_t CLOCK_GetLPOscFreq (void) `[inline]`, `[static]`

Return values

watch	dog OSC frequency value.

4.6.11 static uint32_t CLOCK_GetExtClkFreq (void) `[inline]`, `[static]`

Return values

external	clock frequency value.

4.6.12 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.

4.6.13 void CLOCK_InitExtClkin (uint32_t *clkInFreq*)

Parameters

clkInFreq	external clock in frequency.

4.6.14 void CLOCK_SetFroOscFreq (clock_fro_osc_freq_t *freq*)

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

Parameters

freq	Please refer to definition of clock_fro_osc_freq_t, frequency must be one of 18000, 24000 or 30000 KHz.

4.7 Variable Documentation

4.7.1 volatile uint32_t g_LP_Osc_Freq

This variable is used to store the lower power oscillator frequency which is set by CLOCK_InitLPOsc, and it is returned by CLOCK_GetLPOscFreq.

4.7.2 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.

4.7.3 volatile uint32_t g_Fro_Osc_Freq

This variable is used to store the FRO osc clock frequency.

Chapter 5
Power Driver

5.1 Overview

Power driver provides APIs to control peripherals power and control the system power mode.

Macros

- #define PMUC_PCON_RESERVED_MASK ((0xf << 4) | (0x6 << 8) | 0xfffff000u)
  PMU PCON reserved mask, used to clear reserved field which should not write 1.

Enumerations

- enum _power_wakeup
  Deep sleep and power down mode wake up configurations.
- enum _power_dpd_wakeup_pin
  Deep power down mode wake up pins.
- enum _power_deep_sleep_active
  Deep sleep/power down mode active part.
- enum power_gen_reg_t {
  kPmu_GenReg0 = 0U,
  kPmu_GenReg1 = 1U,
  kPmu_GenReg2 = 2U,
  kPmu_GenReg3 = 3U,
  kPmu_GenReg4 = 4U }
  pmu general purpose register index
- enum power_bod_reset_level_t { kBod_ResetLevel0 = 0U }
  BOD reset level, if VDD below reset level value, the reset will be asserted.
- enum power_bod_interrupt_level_t {
  kBod_InterruptLevelReserved = 0U,
  kBod_InterruptLevel1,
  kBod_InterruptLevel2,
  kBod_InterruptLevel3 }
  BOD interrupt level, if VDD below interrupt level value, the BOD interrupt will be asserted.

Driver version

- #define FSL_POWER_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))
  power driver version 2.1.0.

SYSCON Power Configuration

- static void POWER_EnablePD (pd_bit_t en)
  API to enable PDRUNCFG bit in the Syscon.

- static void POWER_DisablePD (pd_bit_t en)
  API to disable PDRUNCFG bit in the Syscon.

ARM core Power Configuration

- static void POWER_EnableDeepSleep (void)
  API to enable deep sleep bit in the ARM Core.
- static void POWER_DisableDeepSleep (void)
  API to disable deep sleep bit in the ARM Core.

PMU functionality

- void POWER_EnterSleep (void)
  API to enter sleep power mode.
- void POWER_EnterDeepSleep (uint32_t activePart)
  API to enter deep sleep power mode.
- void POWER_EnterPowerDown (uint32_t activePart)
  API to enter power down mode.
- void POWER_EnterDeepPowerDownMode (void)
  API to enter deep power down mode.
- static uint32_t POWER_GetSleepModeFlag (void)
  API to get sleep mode flag.
- static void POWER_ClrSleepModeFlag (void)
  API to clear sleep mode flag.
- static uint32_t POWER_GetDeepPowerDownModeFlag (void)
  API to get deep power down mode flag.
- static void POWER_ClrDeepPowerDownModeFlag (void)
  API to clear deep power down mode flag.
- static void POWER_ClrWakeupPinFlag (void)
  API to clear wake up pin status flag.
- static void POWER_EnableNonDpd (bool enable)
  API to enable non deep power down mode.
- static void POWER_EnableLPO (bool enable)
  API to enable LPO.
- static void POWER_WakeUpConfig (uint32_t mask, bool powerDown)
  API to config wakeup configurations for deep sleep mode and power down mode.
- static void POWER_DeepSleepConfig (uint32_t mask, bool powerDown)
  API to config active part for deep sleep mode and power down mode.

API to enable wake up pin for deep power down mode

Parameters

wakeup_pin	wake up pin for which to enable.reference _power_dpd_wakeup_pin.

Returns

none

- static void **POWER_DeepPowerDownWakeupSourceSelect** (uint32_t wakeup_pin)

- static void POWER_SetRetainData (power_gen_reg_t index, uint32_t data)
  API to retore data to general purpose register which can be retain during deep power down mode.
- static uint32_t POWER_GetRetainData (power_gen_reg_t index)
  API to get data from general purpose register which retain during deep power down mode.
- static void POWER_SetBodLevel (power_bod_reset_level_t resetLevel, power_bod_interrupt_-level_t interruptLevel, bool enable)
  Set Bod interrupt level and reset level.

5.2 Macro Definition Documentation

5.2.1 #define FSL_POWER_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

5.3 Enumeration Type Documentation

5.3.1 enum power_gen_reg_t

Enumerator

kPmu_GenReg0	general purpose register0
kPmu_GenReg1	general purpose register1
kPmu_GenReg2	general purpose register2
kPmu_GenReg3	general purpose register3
kPmu_GenReg4	general purpose reguster4

5.3.2 enum power_bod_reset_level_t

Enumerator

kBod_ResetLevel0	BOD Reset Level0: 1.51V.

5.3.3 enum power_bod_interrupt_level_t

Enumerator

kBod_InterruptLevelReserved	BOD interrupt level reserved.
kBod_InterruptLevel1	BOD interrupt level1: 2.24V.
kBod_InterruptLevel2	BOD interrupt level2: 2.52V.
kBod_InterruptLevel3	BOD interrupt level3: 2.81V.

5.4 Function Documentation

5.4.1 static void POWER_EnablePD (pd_bit_t *en*) `[inline]`, `[static]`

Note that enabling the bit powers down the peripheral

Parameters

en	peripheral for which to enable the PDRUNCFG bit

Returns

none

5.4.2 static void POWER_DisablePD (pd_bit_t *en*) `[inline]`, `[static]`

Note that disabling the bit powers up the peripheral

Parameters

en	peripheral for which to disable the PDRUNCFG bit

Returns

none

5.4.3 static void POWER_EnableDeepSleep (void) `[inline]`, `[static]`

Returns

none

5.4.4 static void POWER_DisableDeepSleep (void) `[inline]`, `[static]`

Returns

none

5.4.5 void POWER_EnterSleep (void)

Returns

none

5.4.6 void POWER_EnterDeepSleep (uint32_t *activePart*)

Parameters

activePart,:	should be a single or combine value of _power_deep_sleep_active .

Returns

none

5.4.7 void POWER_EnterPowerDown (uint32_t *activePart*)

Parameters

activePart,:	should be a single or combine value of _power_deep_sleep_active .

Returns

none

5.4.8 void POWER_EnterDeepPowerDownMode (void)

Returns

none

5.4.9 static uint32_t POWER_GetSleepModeFlag (void) `[inline]`, `[static]`

Returns

sleep mode flag: 0 is active mode, 1 is sleep mode entered.

5.4.10 static uint32_t POWER_GetDeepPowerDownModeFlag (void) `[inline]`, `[static]`

Returns

sleep mode flag: 0 not deep power down, 1 is deep power down mode entered.

5.4.11 static void POWER_EnableNonDpd (bool *enable*) `[inline]`, `[static]`

Parameters

enable,:	true is enable non deep power down, otherwise disable.

5.4.12 static void POWER_EnableLPO (bool *enable*) [inline], [static]

Parameters

enable,:	true to enable LPO, false to disable LPO.

5.4.13 static void POWER_WakeUpConfig (uint32_t *mask,* bool *powerDown*) [inline], [static]

Parameters

mask,:	wake up configurations for deep sleep mode and power down mode, reference _-power_wakeup.
powerDown,:	true is power down the mask part, false is powered part.

5.4.14 static void POWER_DeepSleepConfig (uint32_t *mask,* bool *powerDown*) [inline], [static]

Parameters

mask,:	active part configurations for deep sleep mode and power down mode, reference _-power_deep_sleep_active.
powerDown,:	true is power down the mask part, false is powered part.

5.4.15 static void POWER_SetRetainData (power_gen_reg_t *index,* uint32_t *data*) [inline], [static]

Parameters

index,:	general purpose data register index.
data,:	data to restore.

5.4.16 static uint32_t POWER_GetRetainData (power_gen_reg_t *index*) `[inline]`, `[static]`

Parameters

index,:	general purpose data register index.

Returns

data stored in the general purpose register.

5.4.17 static void POWER_SetBodLevel (power_bod_reset_level_t *resetLevel,* power_bod_interrupt_level_t *interruptLevel,* bool *enable*) `[inline]`, `[static]`

Parameters

resetLevel	BOD reset threshold level, please refer to power_bod_reset_level_t.
interruptLevel	BOD interrupt threshold level, please refer to power_bod_interrupt_level_t.
enable	Used to enable/disable the BOD interrupt and BOD reset.

Chapter 6
Reset Driver

6.1 Overview

Reset driver supports peripheral reset and system reset.

Macros

- #define FLASH_RSTS_N

Enumerations

- 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,
  kWKT_RST_N_SHIFT_RSTn = 0 | 9U,
  kMRT_RST_N_SHIFT_RSTn = 0 | 10U,
  kSPI0_RST_N_SHIFT_RSTn = 0 | 11U,
  kCRC_RST_SHIFT_RSTn = 0 | 13U,
  kUART0_RST_N_SHIFT_RSTn = 0 | 14U,
  kUART1_RST_N_SHIFT_RSTn = 0 | 15U,
  kIOCON_RST_N_SHIFT_RSTn = 0 | 18U,
  kACMP_RST_N_SHIFT_RSTn = 0 | 19U,
  kI2C1_RST_N_SHIFT_RSTn = 0 | 21U,
  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,
  kCAPT_RST_N_SHIFT_RSTn = 65536 | 0U,
  kFRG0_RST_N_SHIFT_RSTn = 65536 | 3U,
  kPLU_RST_N_SHIFT_RSTn = 65536 | 5U }
  Enumeration for peripheral reset control bits.

Functions

- void RESET_PeripheralReset (reset_ip_name_t peripheral)
  Reset peripheral module.

Driver version

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

6.2 Macro Definition Documentation

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

6.2.2 #define FLASH_RSTS_N

Value:

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

Array initializers with peripheral reset bits

6.3 Enumeration Type Documentation

6.3.1 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
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.
kCRC_RST_SHIFT_RSTn	CRC reset control
kUART0_RST_N_SHIFT_RSTn	UART0 reset control
kUART1_RST_N_SHIFT_RSTn	UART1 reset control
kIOCON_RST_N_SHIFT_RSTn	IOCON reset control
kACMP_RST_N_SHIFT_RSTn	Analog comparator reset control
kI2C1_RST_N_SHIFT_RSTn	I2C1 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
kCAPT_RST_N_SHIFT_RSTn	Capacitive Touch reset control
kFRG0_RST_N_SHIFT_RSTn	Fractional baud rate generator 0 reset control
kPLU_RST_N_SHIFT_RSTn	PLU reset control

6.4 Function Documentation

6.4.1 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 7
CAPT: Capacitive Touch

7.1 Overview

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

7.2 Typical use case

7.2.1 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.

Driver version

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

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.
- void CAPT_PollNow (CAPT_Type ∗base, uint16_t enableXpins)
  Start touch data polling using poll-now method.

7.3 Data Structure Documentation

7.3.1 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.

Field Documentation

(1) bool capt_config_t::enableWaitMode

Other-wise, measurements continue.

(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.

(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.

(4) uint8_t capt_config_t::timeOutCount

The time-out count value is calculated as 2^timeOutCount. The available range is 0-12.

(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.

(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 "enableXpins = kCAPT_X0Pin | kCAPT_X2Pin | kCAPT_X3Pin".

(7) capt_trigger_mode_t capt_config_t::triggerMode

(8) capt_inactive_xpins_mode_t capt_config_t::XpinsMode

(9) capt_measurement_delay_t capt_config_t::mDelay

(10) capt_reset_delay_t capt_config_t::rDelay

7.3.2 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.

Field Documentation

(1) bool capt_touch_data_t::yesTimeOut

(2) bool capt_touch_data_t::yesTouch

(3) uint8_t capt_touch_data_t::XpinsIndex

(4) uint8_t capt_touch_data_t::sequenceNumber

(5) uint16_t capt_touch_data_t::count

7.4 Macro Definition Documentation

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

7.5 Enumeration Type Documentation

7.5.1 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.

7.5.2 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.

7.5.3 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.

7.5.4 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.

7.5.5 enum capt_trigger_mode_t

Enumerator

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

7.5.6 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.

7.5.7 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.

7.5.8 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.

7.5.9 enum capt_polling_mode_t

Enumerator

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

7.5.10 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.

7.6 Function Documentation

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

Parameters

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

7.6.2 void CAPT_Deinit (CAPT_Type ∗ *base*)

Parameters

base	CAPT peripheral base address.

7.6.3 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

config	Pointer to the configuration structure.

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

Parameters

base	CAPT peripheral base address.
count	The count threshold.

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

Parameters

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

7.6.6 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".

7.6.7 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".

7.6.8 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".

7.6.9 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".

7.6.10 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.

7.6.11 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.

7.6.12 void CAPT_PollNow (CAPT_Type ∗ *base,* uint16_t *enableXpins*)

This function starts new data polling using polling-now method, CAPT stops when the polling is finished, application could check the status or monitor interrupt to know when the progress is finished.

Note that this is simultaneous poll of all X pins, all enabled X pins are activated concurrently, rather than walked one-at-a-time

Parameters

base	CAPT peripheral base address.
enableXpins	The X pins enabled in this polling.

Chapter 8
Common Driver

8.1 Overview

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

Macros

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

Typedefs

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

Enumerations

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

kStatusGroup_NETC = 166 }
Status group numbers.
- enum {
kStatus_Success = MAKE_STATUS(kStatusGroup_Generic, 0),
kStatus_Fail = MAKE_STATUS(kStatusGroup_Generic, 1),
kStatus_ReadOnly = MAKE_STATUS(kStatusGroup_Generic, 2),
kStatus_OutOfRange = MAKE_STATUS(kStatusGroup_Generic, 3),
kStatus_InvalidArgument = MAKE_STATUS(kStatusGroup_Generic, 4),
kStatus_Timeout = MAKE_STATUS(kStatusGroup_Generic, 5),
kStatus_NoTransferInProgress,
kStatus_Busy = MAKE_STATUS(kStatusGroup_Generic, 7),
kStatus_NoData }
Generic status return codes.

Functions

- void ∗ SDK_Malloc (size_t size, size_t alignbytes)
Allocate memory with given alignment and aligned size.
- void SDK_Free (void ∗ptr)
Free memory.
- void SDK_DelayAtLeastUs (uint32_t delayTime_us, uint32_t coreClock_Hz)
Delay at least for some time.
- static status_t EnableIRQ (IRQn_Type interrupt)
Enable specific interrupt.
- static status_t DisableIRQ (IRQn_Type interrupt)
Disable specific interrupt.
- static status_t EnableIRQWithPriority (IRQn_Type interrupt, uint8_t priNum)
Enable the IRQ, and also set the interrupt priority.
- static status_t IRQ_SetPriority (IRQn_Type interrupt, uint8_t priNum)
Set the IRQ priority.
- static status_t IRQ_ClearPendingIRQ (IRQn_Type interrupt)
Clear the pending IRQ flag.
- static uint32_t DisableGlobalIRQ (void)
Disable the global IRQ.
- static void EnableGlobalIRQ (uint32_t primask)
Enable the global IRQ.

Driver version

- #define FSL_COMMON_DRIVER_VERSION (MAKE_VERSION(2, 4, 0))
common driver version.

Min/max macros

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

UINT16_MAX/UINT32_MAX value

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

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

Suppress fallthrough warning macro

- #define **SUPPRESS_FALL_THROUGH_WARNING**()

Atomic modification

These macros are used for atomic access, such as read-modify-write to the peripheral registers.

- SDK_ATOMIC_LOCAL_ADD
- SDK_ATOMIC_LOCAL_SET
- SDK_ATOMIC_LOCAL_CLEAR
- SDK_ATOMIC_LOCAL_TOGGLE
- SDK_ATOMIC_LOCAL_CLEAR_AND_SET

Take SDK_ATOMIC_LOCAL_CLEAR_AND_SET as an example: the parameter `addr` means the address of the peripheral register or variable you want to modify atomically, the parameter `clearBits` is the bits to clear, the parameter `setBits` it the bits to set. For example, to set a 32-bit register bit1:bit0 to 0b10, use like this:

```
volatile uint32_t * reg = (volatile uint32_t *)REG_ADDR;

SDK_ATOMIC_LOCAL_CLEAR_AND_SET(reg, 0x03, 0x02);
```

In this example, the register bit1:bit0 are cleared and bit1 is set, as a result, register bit1:bit0 = 0b10.

Note

For the platforms don't support exclusive load and store, these macros disable the global interrupt to pretect the modification.
These macros only guarantee the local processor atomic operations. For the multi-processor devices, use hardware semaphore such as SEMA42 to guarantee exclusive access if necessary.

- #define **SDK_ATOMIC_LOCAL_ADD**(addr, val)
- #define **SDK_ATOMIC_LOCAL_SUB**(addr, val)
- #define **SDK_ATOMIC_LOCAL_SET**(addr, bits)
- #define **SDK_ATOMIC_LOCAL_CLEAR**(addr, bits)
- #define **SDK_ATOMIC_LOCAL_TOGGLE**(addr, bits)
- #define **SDK_ATOMIC_LOCAL_CLEAR_AND_SET**(addr, clearBits, setBits)

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_L1DCACHE_ALIGN(var) SDK_ALIGN(var, FSL_FEATURE_L1DCACHE_LINESIZE_BYTE)
  Macro to define a variable with L1 d-cache line size alignment.
- #define SDK_SIZEALIGN(var, alignbytes) ((unsigned int)((var) + ((alignbytes)-1U)) & (unsigned int)(∼(unsigned int)((alignbytes)-1U)))
  Macro to define a variable with L2 cache line size alignment.

8.2 Macro Definition Documentation

8.2.1 #define FSL_DRIVER_TRANSFER_DOUBLE_WEAK_IRQ 1

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

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

The driver version is a 32-bit number, for both 32-bit platforms(such as Cortex M) and 16-bit platforms(such as DSC).

```
| Unused    || Major Version || Minor Version ||  Bug Fix    |
31        25  24            17  16            9  8            0
```

8.2.4 #define FSL_COMMON_DRIVER_VERSION (MAKE_VERSION(2, 4, 0))

8.2.5 #define DEBUG_CONSOLE_DEVICE_TYPE_NONE 0U

8.2.6 #define DEBUG_CONSOLE_DEVICE_TYPE_UART 1U

8.2.7 #define DEBUG_CONSOLE_DEVICE_TYPE_LPUART 2U

8.2.8 #define DEBUG_CONSOLE_DEVICE_TYPE_LPSCI 3U

8.2.9 #define DEBUG_CONSOLE_DEVICE_TYPE_USBCDC 4U

8.2.10 #define DEBUG_CONSOLE_DEVICE_TYPE_FLEXCOMM 5U

8.2.11 #define DEBUG_CONSOLE_DEVICE_TYPE_IUART 6U

8.2.12 #define DEBUG_CONSOLE_DEVICE_TYPE_VUSART 7U

8.2.13 #define DEBUG_CONSOLE_DEVICE_TYPE_MINI_USART 8U

8.2.14 #define DEBUG_CONSOLE_DEVICE_TYPE_SWO 9U

8.2.15 #define DEBUG_CONSOLE_DEVICE_TYPE_QSCI 10U

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

8.2.17 #define SDK_SIZEALIGN(*var, alignbytes*) ((unsigned int)((var) + ((alignbytes)-1U)) & (unsigned int)(∼(unsigned int)((alignbytes)-1U)))

Macro to change a value to a given size aligned value

8.3 Typedef Documentation

8.3.1 typedef int32_t status_t

8.4 Enumeration Type Documentation

8.4.1 enum _status_groups

Enumerator

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

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

kStatusGroup_HAL_RNG Group number for HAL RNG status codes.
kStatusGroup_HAL_I2S Group number for HAL I2S status codes.
kStatusGroup_HAL_ADC_SENSOR Group number for HAL ADC SENSOR status codes.
kStatusGroup_TIMERMANAGER Group number for TiMER MANAGER status codes.
kStatusGroup_SERIALMANAGER Group number for SERIAL MANAGER status codes.
kStatusGroup_LED Group number for LED status codes.
kStatusGroup_BUTTON Group number for BUTTON status codes.
kStatusGroup_EXTERN_EEPROM Group number for EXTERN EEPROM status codes.
kStatusGroup_SHELL Group number for SHELL status codes.
kStatusGroup_MEM_MANAGER Group number for MEM MANAGER status codes.
kStatusGroup_LIST Group number for List status codes.
kStatusGroup_OSA Group number for OSA status codes.
kStatusGroup_COMMON_TASK Group number for Common task status codes.
kStatusGroup_MSG Group number for messaging status codes.
kStatusGroup_SDK_OCOTP Group number for OCOTP status codes.
kStatusGroup_SDK_FLEXSPINOR Group number for FLEXSPINOR status codes.
kStatusGroup_CODEC Group number for codec status codes.
kStatusGroup_ASRC Group number for codec status ASRC.
kStatusGroup_OTFAD Group number for codec status codes.
kStatusGroup_SDIOSLV Group number for SDIOSLV status codes.
kStatusGroup_MECC Group number for MECC status codes.
kStatusGroup_ENET_QOS Group number for ENET_QOS status codes.
kStatusGroup_LOG Group number for LOG status codes.
kStatusGroup_I3CBUS Group number for I3CBUS status codes.
kStatusGroup_QSCI Group number for QSCI status codes.
kStatusGroup_SNT Group number for SNT status codes.
kStatusGroup_QUEUEDSPI Group number for QSPI status codes.
kStatusGroup_POWER_MANAGER Group number for POWER_MANAGER status codes.
kStatusGroup_IPED Group number for IPED status codes.
kStatusGroup_ELS_PKC Group number for ELS PKC status codes.
kStatusGroup_CSS_PKC Group number for CSS PKC status codes.
kStatusGroup_HOSTIF Group number for HOSTIF status codes.
kStatusGroup_CLIF Group number for CLIF status codes.
kStatusGroup_BMA Group number for BMA status codes.
kStatusGroup_NETC Group number for NETC status codes.

8.4.2 anonymous enum

Enumerator

kStatus_Success Generic status for Success.
kStatus_Fail Generic status for Fail.
kStatus_ReadOnly Generic status for read only failure.
kStatus_OutOfRange Generic status for out of range access.

kStatus_InvalidArgument Generic status for invalid argument check.
kStatus_Timeout Generic status for timeout.
kStatus_NoTransferInProgress Generic status for no transfer in progress.
kStatus_Busy Generic status for module is busy.
kStatus_NoData Generic status for no data is found for the operation.

8.5 Function Documentation

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

This is provided to support the dynamically allocated memory used in cache-able region.

Parameters

size	The length required to malloc.
alignbytes	The alignment size.

Return values

The	allocated memory.

8.5.2 void SDK_Free (void ∗ *ptr*)

Parameters

ptr	The memory to be release.

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

Please note that, this API uses while loop for delay, different run-time environments make the time not precise, if precise delay count was needed, please implement a new delay function with hardware timer.

Parameters

delayTime_us	Delay time in unit of microsecond.
coreClock_Hz	Core clock frequency with Hz.

8.5.4 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

8.5.5 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

8.5.6 static status_t EnableIRQWithPriority (IRQn_Type *interrupt,* uint8_t *priNum*) `[inline]`, `[static]`

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

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

Parameters

interrupt	The IRQ to Enable.
priNum	Priority number set to interrupt controller register.

Return values

kStatus_Success	Interrupt priority set successfully
kStatus_Fail	Failed to set the interrupt priority.

8.5.7 static status_t IRQ_SetPriority (IRQn_Type *interrupt,* uint8_t *priNum*) `[inline]`, `[static]`

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

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

Parameters

interrupt	The IRQ to set.
priNum	Priority number set to interrupt controller register.

Return values

kStatus_Success	Interrupt priority set successfully
kStatus_Fail	Failed to set the interrupt priority.

8.5.8 static status_t IRQ_ClearPendingIRQ (IRQn_Type *interrupt*) `[inline]`, `[static]`

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

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

Parameters

interrupt	The flag which IRQ to clear.

Return values

kStatus_Success	Interrupt priority set successfully
kStatus_Fail	Failed to set the interrupt priority.

8.5.9 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.

8.5.10 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().

Chapter 9
CTIMER: Standard counter/timers

9.1 Overview

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

9.2 Function groups

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

9.2.1 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.

9.2.2 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.

9.2.3 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.

9.2.4 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.

9.3 Typical use case

9.3.1 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

9.3.2 PWM output example

Set up a channel for PWM output. Refer to the driver examples codes located at <SDK_ROO-T>/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_external_match_t {
  kCTIMER_External_Match_0 = (1UL << 0),
  kCTIMER_External_Match_1 = (1UL << 1),
  kCTIMER_External_Match_2 = (1UL << 2),
  kCTIMER_External_Match_3 = (1UL << 3) }

List of external match.
- 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.
- uint32_t CTIMER_GetOutputMatchStatus (CTIMER_Type ∗base, uint32_t matchChannel)
Get the status of output match.
- 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.
- static void CTIMER_SetPrescale (CTIMER_Type ∗base, uint32_t prescale)
  Setup the timer prescale value.
- static uint32_t CTIMER_GetCaptureValue (CTIMER_Type ∗base, ctimer_capture_channel_t capture)
  Get capture channel value.
- static void CTIMER_EnableResetMatchChannel (CTIMER_Type ∗base, ctimer_match_t match, bool enable)
  Enable reset match channel.
- static void CTIMER_EnableStopMatchChannel (CTIMER_Type ∗base, ctimer_match_t match, bool enable)
  Enable stop match channel.
- static void CTIMER_EnableRisingEdgeCapture (CTIMER_Type ∗base, ctimer_capture_channel_t capture, bool enable)
  Enable capture channel rising edge.
- static void CTIMER_EnableFallingEdgeCapture (CTIMER_Type ∗base, ctimer_capture_channel_t capture, bool enable)
  Enable capture channel falling edge.

Driver version

- #define FSL_CTIMER_DRIVER_VERSION (MAKE_VERSION(2, 3, 1))
  Version 2.3.1.

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, const ctimer_match_t pwmPeriodChannel, ctimer_match_t matchChannel, uint32_t pwmPeriod, uint32_t pulsePeriod, bool enableInt)
  Configures the PWM signal parameters.
- status_t CTIMER_SetupPwm (CTIMER_Type ∗base, const ctimer_match_t pwmPeriodChannel, ctimer_match_t matchChannel, uint8_t dutyCyclePercent, uint32_t pwmFreq_Hz, uint32_t srcClock_Hz, bool enableInt)
  Configures the PWM signal parameters.
- 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, const ctimer_match_t pwmPeriodChannel, 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.

9.4 Data Structure Documentation

9.4.1 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

9.4.2 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.

9.5 Enumeration Type Documentation

9.5.1 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.

9.5.2 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.

9.5.3 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.

9.5.4 enum ctimer_external_match_t

Enumerator

kCTIMER_External_Match_0	External match 0.
kCTIMER_External_Match_1	External match 1.
kCTIMER_External_Match_2	External match 2.
kCTIMER_External_Match_3	External match 3.

9.5.5 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.

9.5.6 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.

9.5.7 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.

9.5.8 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

9.6 Function Documentation

9.6.1 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.

9.6.2 void CTIMER_Deinit (CTIMER_Type ∗ *base*)

Parameters

base	Ctimer peripheral base address

9.6.3 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.

9.6.4 status_t CTIMER_SetupPwmPeriod (CTIMER_Type ∗ *base,* const ctimer_match_t *pwmPeriodChannel,* 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 can manually assign the specified channel to set the PWM cycle.

Note

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

Parameters

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

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

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

Note

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

Parameters

base	Ctimer peripheral base address
pwmPeriod-Channel	Specify the channel to control the PWM period
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 will be generated.

9.6.6 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

9.6.7 void CTIMER_UpdatePwmDutycycle (CTIMER_Type ∗ *base,* const ctimer_match_t *pwmPeriodChannel,* ctimer_match_t *matchChannel,* uint8_t *dutyCyclePercent*)

Note

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

Parameters

base	Ctimer peripheral base address
pwmPeriod-Channel	Specify the channel to control the PWM period
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

9.6.8 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

9.6.9 uint32_t CTIMER_GetOutputMatchStatus (CTIMER_Type ∗ *base,* uint32_t *matchChannel*)

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

Parameters

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

Returns

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

9.6.10 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

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

Parameters

base	Ctimer peripheral base address.

Returns

return the timer count value.

9.6.12 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

9.6.13 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

9.6.14 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

9.6.15 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

9.6.16 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

9.6.17 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

9.6.18 static void CTIMER_StartTimer (CTIMER_Type ∗ *base*) `[inline]`, `[static]`

Parameters

base	Ctimer peripheral base address

9.6.19 static void CTIMER_StopTimer (CTIMER_Type ∗ *base*) `[inline]`, `[static]`

Parameters

base	Ctimer peripheral base address

9.6.20 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

9.6.21 static void CTIMER_SetPrescale (CTIMER_Type ∗ *base*, uint32_t *prescale*) `[inline]`, `[static]`

Specifies the maximum value for the Prescale Counter.

Parameters

base	Ctimer peripheral base address
prescale	Prescale value

9.6.22 static uint32_t CTIMER_GetCaptureValue (CTIMER_Type ∗ *base*, ctimer_capture_channel_t *capture*) [inline], [static]

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

Parameters

base	Ctimer peripheral base address
capture	Select capture channel

Returns

The timer count capture value.

9.6.23 static void CTIMER_EnableResetMatchChannel (CTIMER_Type ∗ *base*, ctimer_match_t *match*, bool *enable*) [inline], [static]

Set the specified match channel reset operation.

Parameters

base	Ctimer peripheral base address
match	match channel used
enable	Enable match channel reset operation.

9.6.24 static void CTIMER_EnableStopMatchChannel (CTIMER_Type ∗ *base*, ctimer_match_t *match*, bool *enable*) [inline], [static]

Set the specified match channel stop operation.

Parameters

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

9.6.25 static void CTIMER_EnableRisingEdgeCapture (CTIMER_Type ∗ *base*, ctimer_capture_channel_t *capture*, bool *enable*) [inline], [static]

Sets the specified capture channel for rising edge capture.

Parameters

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

9.6.26 static void CTIMER_EnableFallingEdgeCapture (CTIMER_Type ∗ *base*, ctimer_capture_channel_t *capture*, bool *enable*) [inline], [static]

Sets the specified capture channel for falling edge capture.

Parameters

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

Chapter 10
IAP: In Application Programming Driver

10.1 Overview

The MCUXpresso SDK provides a driver for the In Application Programming (IAP) module of MCU-Xpresso SDK devices.

10.2 Function groups

The driver provides a set of functions to call the on-chip in application programming interface. User code executing from on-chip RAM can call these functions to read information like part id; read and write flash, EEPROM and FAIM.

10.2.1 Basic operations

The function IAP_ReadPartID() reads the part id of the board.

The function IAP_ReadBootCodeVersion() reads the boot code Version.

The function IAP_ReadUniqueID() reads the unique id of the boards.

The function IAP_ReinvokeISP() reinvokes the ISP mode.

The function IAP_ReadFactorySettings() reads the factory settings.

10.2.2 Flash operations

The function IAP_PrepareSectorForWrite() prepares a sector for write or erase operation. Then, the function IAP_CopyRamToFlash() programs the flash memory.

The function IAP_EraseSector() erases a flash sector while the function IAP_ErasePage() erases a flash page.

The function IAP_BlankCheckSector() is used to blank check a sector or multiple sectors of on-chip flash memory.

The function IAP_Compare() is used to compare the memory contents at two locations. The user can compare several bytes (must be a multiple of 4) content in two different flash locations.

The function IAP_ReadFlashSignature() can get the 32-bits signature of the entire flash and the function IAP_ExtendedFlashSignatureRead() can calculate the signature of one or more flash pages.

10.2.3 EEPROM operations

The function IAP_ReadEEPROMPage() reads the 128 bytes content of an EEPROM page and IAP_Write-EEPROMPage() writes 128 bytes content in an EEPROM page

10.2.4 FAIM operations

The function IAP_ReadEEPROMPage() reads the 32 bits content of an FAIM page and IAP_WriteEEP-ROMPage() writes 32 bits content in an FAIM page

10.3 Typical use case

10.3.1 IAP Basic Operations

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

10.3.2 IAP Flash Operations

Refer to the driver example codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/iap/iap-_flash/

10.3.3 IAP EEPROM Operations

Refer to the driver example codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/iap/iap-_eeprom/

10.3.4 IAP FAIM Operations

Refer to the driver example codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/iap/iap-_faim/

Files

- file fsl_iap.h

Enumerations

- enum {
  kStatus_IAP_Success = kStatus_Success,
  kStatus_IAP_InvalidCommand = MAKE_STATUS(kStatusGroup_IAP, 1U),
  kStatus_IAP_SrcAddrError = MAKE_STATUS(kStatusGroup_IAP, 2U),
  kStatus_IAP_DstAddrError,
  kStatus_IAP_SrcAddrNotMapped,
  kStatus_IAP_DstAddrNotMapped,
  kStatus_IAP_CountError,
  kStatus_IAP_InvalidSector,
  kStatus_IAP_SectorNotblank = MAKE_STATUS(kStatusGroup_IAP, 8U),
  kStatus_IAP_NotPrepared,
  kStatus_IAP_CompareError,
  kStatus_IAP_Busy = MAKE_STATUS(kStatusGroup_IAP, 11U),
  kStatus_IAP_ParamError,
  kStatus_IAP_AddrError = MAKE_STATUS(kStatusGroup_IAP, 13U),
  kStatus_IAP_AddrNotMapped = MAKE_STATUS(kStatusGroup_IAP, 14U),
  kStatus_IAP_NoPower = MAKE_STATUS(kStatusGroup_IAP, 24U),
  kStatus_IAP_NoClock = MAKE_STATUS(kStatusGroup_IAP, 27U),
  kStatus_IAP_ReinvokeISPConfig = MAKE_STATUS(kStatusGroup_IAP, 0x1CU) }

  iap status codes.
- enum _iap_commands {
  kIapCmd_IAP_ReadFactorySettings = 40U,
  kIapCmd_IAP_PrepareSectorforWrite = 50U,
  kIapCmd_IAP_CopyRamToFlash = 51U,
  kIapCmd_IAP_EraseSector = 52U,
  kIapCmd_IAP_BlankCheckSector = 53U,
  kIapCmd_IAP_ReadPartId = 54U,
  kIapCmd_IAP_Read_BootromVersion = 55U,
  kIapCmd_IAP_Compare = 56U,
  kIapCmd_IAP_ReinvokeISP = 57U,
  kIapCmd_IAP_ReadUid = 58U,
  kIapCmd_IAP_ErasePage = 59U,
  kIapCmd_IAP_ReadSignature = 70U,
  kIapCmd_IAP_ExtendedReadSignature = 73U,
  kIapCmd_IAP_ReadEEPROMPage = 80U,
  kIapCmd_IAP_WriteEEPROMPage = 81U }

  iap command codes.
- enum _flash_access_time { ,
  kFlash_IAP_TwoSystemClockTime = 1U,
  kFlash_IAP_ThreeSystemClockTime = 2U }

  Flash memory access time.

Driver version

- #define **FSL_IAP_DRIVER_VERSION** (MAKE_VERSION(2, 0, 7))

Basic operations

- status_t IAP_ReadPartID (uint32_t ∗partID)
  Read part identification number.
- status_t IAP_ReadBootCodeVersion (uint32_t ∗bootCodeVersion)
  Read boot code version number.
- void IAP_ReinvokeISP (uint8_t ispType, uint32_t ∗status)
  Reinvoke ISP.
- status_t IAP_ReadUniqueID (uint32_t ∗uniqueID)
  Read unique identification.

10.4 Enumeration Type Documentation

10.4.1 anonymous enum

Enumerator

kStatus_IAP_Success Api is executed successfully.
kStatus_IAP_InvalidCommand Invalid command.
kStatus_IAP_SrcAddrError Source address is not on word boundary.
kStatus_IAP_DstAddrError Destination address is not on a correct boundary.
kStatus_IAP_SrcAddrNotMapped Source address is not mapped in the memory map.
kStatus_IAP_DstAddrNotMapped Destination address is not mapped in the memory map.
kStatus_IAP_CountError Byte count is not multiple of 4 or is not a permitted value.
kStatus_IAP_InvalidSector Sector/page number is invalid or end sector/page number is greater than start sector/page number.
kStatus_IAP_SectorNotblank One or more sectors are not blank.
kStatus_IAP_NotPrepared Command to prepare sector for write operation has not been executed.
kStatus_IAP_CompareError Destination and source memory contents do not match.
kStatus_IAP_Busy Flash programming hardware interface is busy.
kStatus_IAP_ParamError Insufficient number of parameters or invalid parameter.
kStatus_IAP_AddrError Address is not on word boundary.
kStatus_IAP_AddrNotMapped Address is not mapped in the memory map.
kStatus_IAP_NoPower Flash memory block is powered down.
kStatus_IAP_NoClock Flash memory block or controller is not clocked.
kStatus_IAP_ReinvokeISPConfig Reinvoke configuration error.

10.4.2 enum _iap_commands

Enumerator

kIapCmd_IAP_ReadFactorySettings Read the factory settings.
kIapCmd_IAP_PrepareSectorforWrite Prepare Sector for write.
kIapCmd_IAP_CopyRamToFlash Copy RAM to flash.
kIapCmd_IAP_EraseSector Erase Sector.
kIapCmd_IAP_BlankCheckSector Blank check sector.

kIapCmd_IAP_ReadPartId	Read part id.
kIapCmd_IAP_Read_BootromVersion	Read bootrom version.
kIapCmd_IAP_Compare	Compare.
kIapCmd_IAP_ReinvokeISP	Reinvoke ISP.
kIapCmd_IAP_ReadUid	Read Uid.
kIapCmd_IAP_ErasePage	Erase Page.
kIapCmd_IAP_ReadSignature	Read Signature.
kIapCmd_IAP_ExtendedReadSignature	Extended Read Signature.
kIapCmd_IAP_ReadEEPROMPage	Read EEPROM page.
kIapCmd_IAP_WriteEEPROMPage	Write EEPROM page.

10.4.3 enum _flash_access_time

Enumerator

kFlash_IAP_TwoSystemClockTime	1 system clock flash access time
kFlash_IAP_ThreeSystemClockTime	2 system clock flash access time

10.5 Function Documentation

10.5.1 status_t IAP_ReadPartID (uint32_t ∗ *partID*)

This function is used to read the part identification number.

Parameters

partID	Address to store the part identification number.

Return values

kStatus_IAP_Success	Api has been executed successfully.

10.5.2 status_t IAP_ReadBootCodeVersion (uint32_t ∗ *bootCodeVersion*)

This function is used to read the boot code version number.

Parameters

bootCode-Version	Address to store the boot code version.

Return values

kStatus_IAP_Success	Api has been executed successfully.

note Boot code version is two 32-bit words. Word 0 is the major version, word 1 is the minor version.

10.5.3 void IAP_ReinvokeISP (uint8_t *ispType,* uint32_t ∗ *status*)

This function is used to invoke the boot loader in ISP mode. It maps boot vectors and configures the peripherals for ISP.

Parameters

ispType	ISP type selection.
status	store the possible status.

Return values

kStatus_IAP_ReinvokeIS-PConfig	reinvoke configuration error.

note The error response will be returned when IAP is disabled or an invalid ISP type selection appears. The call won't return unless an error occurs, so there can be no status code.

10.5.4 status_t IAP_ReadUniqueID (uint32_t ∗ *uniqueID*)

This function is used to read the unique id.

Parameters

uniqueID	store the uniqueID.

Return values

kStatus_IAP_Success	Api has been executed successfully.

Chapter 11
LPC_ACOMP: Analog comparator Driver

11.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Analog comparator (LPC_ACOMP) 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/acomp/acomp-_basic

11.2.2 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 }
  The ACOMP ladder reference voltage.
- enum acomp_interrupt_enable_t {
  kACOMP_InterruptsFallingEdgeEnable = 0U,
  kACOMP_InterruptsRisingEdgeEnable = 1U,
  kACOMP_InterruptsBothEdgesEnable = 2U }
  The ACOMP interrupts enable.

- enum acomp_hysteresis_selection_t {
  kACOMP_HysteresisNoneSelection = 0U,
  kACOMP_Hysteresis5MVSelection = 1U,
  kACOMP_Hysteresis10MVSelection = 2U,
  kACOMP_Hysteresis20MVSelection = 3U }
  The ACOMP hysteresis selection.

Driver version

- #define FSL_ACOMP_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))
  ACOMP driver version 2.1.0.

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.

11.3 Data Structure Documentation

11.3.1 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.

Field Documentation

(1) bool acomp_config_t::enableSyncToBusClk

If false, Comparator output is used directly.

(2) acomp_hysteresis_selection_t acomp_config_t::hysteresisSelection

11.3.2 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.

Field Documentation

(1) uint8_t acomp_ladder_config_t::ladderValue

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

(2) acomp_ladder_reference_voltage_t acomp_ladder_config_t::referenceVoltage

11.4 Macro Definition Documentation

11.4.1 #define FSL_ACOMP_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

11.5 Enumeration Type Documentation

11.5.1 enum acomp_ladder_reference_voltage_t

Enumerator

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

11.5.2 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.

11.5.3 enum acomp_hysteresis_selection_t

Enumerator

kACOMP_HysteresisNoneSelection	None (the output will switch as the voltages cross).

kACOMP_Hysteresis5MVSelection	5mV.
kACOMP_Hysteresis10MVSelection	10mV.
kACOMP_Hysteresis20MVSelection	20mV.

11.6 Function Documentation

11.6.1 void ACOMP_Init (ACOMP_Type ∗ *base,* const acomp_config_t ∗ *config*)

Parameters

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

11.6.2 void ACOMP_Deinit (ACOMP_Type ∗ *base*)

Parameters

base	ACOMP peripheral base address.

11.6.3 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

config	Pointer to the configuration structure.

11.6.4 void ACOMP_EnableInterrupts (ACOMP_Type ∗ *base,* acomp_interrupt_enable_t *enable*)

Parameters

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

11.6.5 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.

11.6.6 static void ACOMP_ClearInterruptsStatusFlags (ACOMP_Type ∗ *base*) `[inline]`, `[static]`

Parameters

base	ACOMP peripheral base address.

11.6.7 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.

11.6.8 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.

11.6.9 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.

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

12.1 Overview

The MCUXpresso SDK provides a peripheral driver for the 12-bit Successive Approximation (SAR) Analog-to-Digital Converter (ADC) 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/fsl_-adc

12.2.2 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_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, 5, 2))
ADC driver version 2.5.2.

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 hardware self-calibration.
- static void ADC_EnableTemperatureSensor (ADC_Type ∗base, bool enable)
  Enable the internal temperature sensor measurement.

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_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.

12.3 Data Structure Documentation

12.3.1 struct adc_config_t

Data Fields

- uint32_t clockDividerNumber
  This field is only available when using kADC_ClockSynchronousMode for "clockMode" field.

Field Documentation

(1) 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.

12.3.2 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 launch 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.

Field Documentation

(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.

(2) uint32_t adc_conv_seq_config_t::triggerMask

The available range is 6-bit.

(3) adc_trigger_polarity_t adc_conv_seq_config_t::triggerPolarity

(4) bool adc_conv_seq_config_t::enableSyncBypass

(5) bool adc_conv_seq_config_t::enableSingleStep

(6) adc_seq_interrupt_mode_t adc_conv_seq_config_t::interruptMode

12.3.3 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.

Field Documentation

(1) uint32_t adc_result_info_t::result

(2) adc_threshold_compare_status_t adc_result_info_t::thresholdCompareStatus

(3) adc_threshold_crossing_status_t adc_result_info_t::thresholdCorssingStatus

(4) uint32_t adc_result_info_t::channelNumber

(5) bool adc_result_info_t::overrunFlag

12.4 Macro Definition Documentation

12.4.1 #define FSL_ADC_DRIVER_VERSION (MAKE_VERSION(2, 5, 2))

12.5 Enumeration Type Documentation

12.5.1 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.

12.5.2 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.

12.5.3 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).

12.5.4 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.

12.5.5 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.

12.5.6 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.

12.5.7 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.

12.5.8 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.

12.5.9 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.

12.5.10 enum adc_tempsensor_common_mode_t

Enumerator

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

12.5.11 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.

12.6 Function Documentation

12.6.1 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.

12.6.2 void ADC_Deinit (ADC_Type ∗ *base*)

Parameters

base	ADC peripheral base address.

12.6.3 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.

12.6.4 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 clock frequency that ADC operates at.

Return values

true	Calibration succeed.
false	Calibration failed.

12.6.5 static void ADC_EnableTemperatureSensor (ADC_Type ∗ *base,* bool *enable*) [inline], [static]

When enabling the internal temperature sensor measurement, the channel 0 would be connected to internal sensor instead of external pin.

Parameters

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

12.6.6 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.

12.6.7 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.

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

Parameters

base	ADC peripheral base address.

12.6.9 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.

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

Parameters

base	ADC peripheral bass address.

12.6.11 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.

12.6.12 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.

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

Parameters

base	ADC peripheral base address.

12.6.14 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.

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

Parameters

base	ADC peripheral bass address.

12.6.16 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.

12.6.17 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.

12.6.18 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.

12.6.19 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.

12.6.20 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.

12.6.21 static void ADC_SetChannelWithThresholdPair0 (ADC_Type * *base,* uint32_t *channelMask*) `[inline]`, `[static]`

Parameters

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

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

Parameters

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

12.6.23 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.

12.6.24 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.

12.6.25 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.

12.6.26 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.

12.6.27 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 13
CRC: Cyclic Redundancy Check Driver

13.1 Overview

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

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

13.2 CRC Driver Initialization and Configuration

CRC_Init() function enables the clock for the CRC module in the LPC SYSCON block and fully (re-)configures the CRC module according to configuration structure. It also starts checksum computation by writing the seed.

The seed member of the configuration structure is the initial checksum for which new data can be added to. When starting new checksum computation, the seed should be set to the initial checksum per the C-RC protocol specification. For continued checksum operation, the seed should be set to the intermediate checksum value as obtained from previous calls to CRC_GetConfig() function. After CRC_Init(), one or multiple CRC_WriteData() calls follow to update checksum with data, then CRC_Get16bitResult() or CRC_Get32bitResult() follows to read the result. CRC_Init() can be called as many times as required, which allows for runtime changes of the CRC protocol.

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

CRC_Deinit() function disables clock to the CRC module.

CRC_Reset() performs hardware reset of the CRC module.

13.3 CRC Write Data

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

CRC_WriteSeed() Write seed (initial checksum) to CRC module.

13.4 CRC Get Checksum

The CRC_Get16bitResult() or CRC_Get32bitResult() function is used to read the CRC module checksum register. The bit reverse and 1's complement operations are already applied to the result if previously configured. Use CRC_GetConfig() function to get the actual checksum without bit reverse and 1's complement applied so it can be used as seed when resuming calculation later.

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

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

CRC_Init() / CRC_WriteData() / CRC_GetConfig() to get intermediate checksum to be used as seed value in future.

CRC_Init() / CRC_WriteData() / ... / CRC_WriteData() / CRC_GetConfig() to get intermediate checksum.

13.5 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() or CRC_GetConfig()

Should be protected by RTOS mutex to protect CRC module against concurrent accesses from different tasks. For 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/crcRefer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/crc

Files

- file fsl_crc.h

Data Structures

- struct crc_config_t
  CRC protocol configuration. More...

Macros

- #define CRC_DRIVER_USE_CRC16_CCITT_FALSE_AS_DEFAULT 1
  Default configuration structure filled by CRC_GetDefaultConfig().

Enumerations

- enum crc_polynomial_t {
  kCRC_Polynomial_CRC_CCITT = 0U,
  kCRC_Polynomial_CRC_16 = 1U,
  kCRC_Polynomial_CRC_32 = 2U }
  CRC polynomials to use.

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_Reset (CRC_Type ∗base)
  resets CRC peripheral module.
- void CRC_WriteSeed (CRC_Type ∗base, uint32_t seed)
  Write seed to CRC peripheral module.
- void CRC_GetDefaultConfig (crc_config_t ∗config)
  Loads default values to CRC protocol configuration structure.
- void CRC_GetConfig (CRC_Type ∗base, crc_config_t ∗config)
  Loads actual values configured in CRC peripheral to CRC protocol configuration structure.
- void CRC_WriteData (CRC_Type ∗base, const uint8_t ∗data, size_t dataSize)
  Writes data to the CRC module.
- static uint32_t CRC_Get32bitResult (CRC_Type ∗base)
  Reads 32-bit checksum from the CRC module.
- static uint16_t CRC_Get16bitResult (CRC_Type ∗base)
  Reads 16-bit checksum from the CRC module.

Driver version

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

13.6 Data Structure Documentation

13.6.1 struct crc_config_t

This structure holds the configuration for the CRC protocol.

Data Fields

- crc_polynomial_t polynomial
  CRC polynomial.
- bool reverseIn
  Reverse bits on input.
- bool complementIn
  Perform 1's complement on input.
- bool reverseOut
  Reverse bits on output.
- bool complementOut
  Perform 1's complement on output.
- uint32_t seed
  Starting checksum value.

Field Documentation

(1) crc_polynomial_t crc_config_t::polynomial

(2) bool crc_config_t::reverseIn

(3) bool crc_config_t::complementIn

(4) bool crc_config_t::reverseOut

(5) bool crc_config_t::complementOut

(6) uint32_t crc_config_t::seed

13.7 Macro Definition Documentation

13.7.1 #define FSL_CRC_DRIVER_VERSION (MAKE_VERSION(2, 1, 1))

Version 2.1.1.

Current version: 2.1.1

Change log:

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

13.7.2 #define CRC_DRIVER_USE_CRC16_CCITT_FALSE_AS_DEFAULT 1

Uses CRC-16/CCITT-FALSE as default.

13.8 Enumeration Type Documentation

13.8.1 enum crc_polynomial_t

Enumerator

kCRC_Polynomial_CRC_CCITT	x^16+x^12+x^5+1
kCRC_Polynomial_CRC_16	x^16+x^15+x^2+1
kCRC_Polynomial_CRC_32	x^32+x^26+x^23+x^22+x^16+x^12+x^11+x^10+x^8+x^7+x^5+x^4+x^2+x+

13.9 Function Documentation

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

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

Parameters

base	CRC peripheral address.
config	CRC module configuration structure.

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

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

Parameters

base	CRC peripheral address.

13.9.3 void CRC_Reset (CRC_Type ∗ *base*)

Parameters

base	CRC peripheral address.

13.9.4 void CRC_WriteSeed (CRC_Type ∗ *base,* uint32_t *seed*)

Parameters

base	CRC peripheral address.
seed	CRC Seed value.

13.9.5 void CRC_GetDefaultConfig (crc_config_t ∗ *config*)

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

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

Parameters

config	CRC protocol configuration structure

13.9.6 void CRC_GetConfig (CRC_Type ∗ *base,* crc_config_t ∗ *config*)

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

Parameters

base	CRC peripheral address.
config	CRC protocol configuration structure

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

Writes input data buffer bytes to CRC data register.

Parameters

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

13.9.8 static uint32_t CRC_Get32bitResult (CRC_Type ∗ *base*) `[inline]`, `[static]`

Reads CRC data register.

Parameters

base	CRC peripheral address.

Returns

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

13.9.9 static uint16_t CRC_Get16bitResult (CRC_Type * *base*) [inline], [static]

Reads CRC data register.

Parameters

base	CRC peripheral address.

Returns

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

Chapter 14
DAC: 10-bit Digital To Analog Converter Driver

14.1 Overview

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

14.2 Typical use case

14.2.1 Polling Configuration

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

14.2.2 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_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, 2))
  DAC driver version 2.0.2.

14.3 Data Structure Documentation

14.3.1 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.

Field Documentation

(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.

14.4 Macro Definition Documentation

14.4.1 #define LPC_DAC_DRIVER_VERSION (MAKE_VERSION(2, 0, 2))

14.5 Enumeration Type Documentation

14.5.1 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 350u-A. This allows a maximum update rate of 400 kHz.

14.6 Function Documentation

14.6.1 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.

14.6.2 void DAC_Deinit (DAC_Type ∗ *base*)

Parameters

base	DAC peripheral base address.

14.6.3 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".

14.6.4 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.

14.6.5 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.

14.6.6 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.

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

Parameters

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

14.6.8 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 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, 7))
  LPC GPIO driver version.

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, 7))

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
I2C: Inter-Integrated Circuit Driver

16.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Inter-Integrated Circuit (I2C) module of MC-UXpresso 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 I2-C 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_MasterTransfer-NonBlocking() set up the interrupt non-blocking transfer. When the transfer completes, the upper layer is notified through a callback function with the status.

16.2 Typical use case

16.2.1 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

16.2.2 Master Operation in DMA transactional method

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

16.2.3 Slave Operation in functional method

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

16.2.4 Slave Operation in interrupt transactional method

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

Modules

- I2C Driver
- I2C Master Driver
- I2C Slave Driver

16.3 I2C Driver

16.3.1 Overview

Files

- file fsl_i2c.h

Macros

- #define I2C_RETRY_TIMES 0U /∗ Define to zero means keep waiting until the flag is assert/deassert. ∗/
  Retry times for waiting flag.
- #define I2C_STAT_MSTCODE_IDLE (0)
  Master Idle State Code.
- #define I2C_STAT_MSTCODE_RXREADY (1UL)
  Master Receive Ready State Code.
- #define I2C_STAT_MSTCODE_TXREADY (2UL)
  Master Transmit Ready State Code.
- #define I2C_STAT_MSTCODE_NACKADR (3UL)
  Master NACK by slave on address State Code.
- #define I2C_STAT_MSTCODE_NACKDAT (4UL)
  Master NACK by slave on data State Code.

Enumerations

- enum {
  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_StartStopError = MAKE_STATUS(kStatusGroup_LPC_I2C, 8),
  kStatus_I2C_UnexpectedState = MAKE_STATUS(kStatusGroup_LPC_I2C, 9),
  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, 1, 0))
  I2C driver version.

16.3.2 Macro Definition Documentation

16.3.2.1 #define FSL_I2C_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

16.3.2.2 #define I2C_RETRY_TIMES 0U /∗ Define to zero means keep waiting until the flag is assert/deassert. ∗/

16.3.3 Enumeration Type Documentation

16.3.3.1 anonymous enum

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_StartStopError	Start and stop error.
kStatus_I2C_UnexpectedState	Unexpected state.
kStatus_I2C_Addr_Nak	NAK received during the address probe.
kStatus_I2C_Timeout	Timeout polling status flags.

16.4 I2C Master Driver

16.4.1 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.

16.4.2 Data Structure Documentation

16.4.2.1 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.

Field Documentation

(1) bool i2c_master_config_t::enableMaster

(2) uint32_t i2c_master_config_t::baudRate_Bps

(3) bool i2c_master_config_t::enableTimeout

16.4.2.2 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.

Field Documentation

(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.

(2) uint16_t i2c_master_transfer_t::slaveAddress

(3) i2c_direction_t i2c_master_transfer_t::direction

(4) uint32_t i2c_master_transfer_t::subaddress

Transferred MSB first.

(5) size_t i2c_master_transfer_t::subaddressSize

Maximum size is 4 bytes.

(6) void∗ i2c_master_transfer_t::data

(7) size_t i2c_master_transfer_t::dataSize

16.4.2.3 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.

Field Documentation

(1) uint8_t i2c_master_handle_t::state

(2) uint32_t i2c_master_handle_t::remainingBytes

(3) uint8_t∗ i2c_master_handle_t::buf

(4) i2c_master_transfer_t i2c_master_handle_t::transfer

(5) i2c_master_transfer_callback_t i2c_master_handle_t::completionCallback

(6) void∗ i2c_master_handle_t::userData

16.4.3 Typedef Documentation

16.4.3.1 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.

16.4.4 Enumeration Type Documentation

16.4.4.1 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.

16.4.4.2 enum i2c_direction_t

Enumerator

kI2C_Write Master transmit.
kI2C_Read Master receive.

16.4.4.3 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.

16.4.4.4 enum _i2c_transfer_states

16.4.5 Function Documentation

16.4.5.1 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.

16.4.5.2 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.

16.4.5.3 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.

16.4.5.4 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.

16.4.5.5 static void I2C_MasterReset (I2C_Type ∗ *base*) `[inline]`, `[static]`

Restores the I2C master peripheral to reset conditions.

Parameters

base	The I2C peripheral base address.

16.4.5.6 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.

16.4.5.7 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

16.4.5.8 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.

16.4.5.9 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.

16.4.5.10 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.

16.4.5.11 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.

16.4.5.12 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.

16.4.5.13 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.

16.4.5.14 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.

16.4.5.15 status_t I2C_MasterStop (I2C_Type ∗ *base*)

Return values

kStatus_Success	Successfully send the stop signal.
kStatus_I2C_Timeout	Send stop signal failed, timeout.

16.4.5.16 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.

16.4.5.17 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.

16.4.5.18 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.

16.4.5.19 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.

16.4.5.20 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.

16.4.5.21 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.

16.4.5.22 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	

16.4.5.23 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.

16.4.5.24 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.
i2cHandle	Pointer to the I2C master driver handle i2c_master_handle_t.

16.5 I2C Slave Driver

16.5.1 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.

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.

16.5.2 Data Structure Documentation

16.5.2.1 struct i2c_slave_address_t

Data Fields

- uint8_t address
  7-bit Slave address SLVADR.
- bool addressDisable
  Slave address disable SADISABLE.

Field Documentation

(1) uint8_t i2c_slave_address_t::address

(2) bool i2c_slave_address_t::addressDisable

16.5.2.2 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.

Field Documentation

(1) i2c_slave_address_t i2c_slave_config_t::address0

(2) i2c_slave_address_t i2c_slave_config_t::address1

(3) i2c_slave_address_t i2c_slave_config_t::address2

(4) i2c_slave_address_t i2c_slave_config_t::address3

(5) i2c_slave_address_qual_mode_t i2c_slave_config_t::qualMode

(6) uint8_t i2c_slave_config_t::qualAddress

(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)

(8) bool i2c_slave_config_t::enableSlave

16.5.2.3 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.

Field Documentation

(1) i2c_slave_handle_t∗ i2c_slave_transfer_t::handle

(2) i2c_slave_transfer_event_t i2c_slave_transfer_t::event

(3) uint8_t i2c_slave_transfer_t::receivedAddress

7-bits plus R/nW bit0

(4) uint32_t i2c_slave_transfer_t::eventMask

(5) size_t i2c_slave_transfer_t::transferredCount

(6) status_t i2c_slave_transfer_t::completionStatus

Only applies for kI2C_SlaveCompletionEvent.

16.5.2.4 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.

Field Documentation

(1) volatile i2c_slave_transfer_t i2c_slave_handle_t::transfer

(2) volatile bool i2c_slave_handle_t::isBusy

(3) volatile i2c_slave_fsm_t i2c_slave_handle_t::slaveFsm

(4) i2c_slave_transfer_callback_t i2c_slave_handle_t::callback

(5) void∗ i2c_slave_handle_t::userData

16.5.3 Typedef Documentation

16.5.3.1 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.

16.5.3.2 typedef void(∗ i2c_isr_t)(I2C_Type ∗base, void ∗i2cHandle)

16.5.4 Enumeration Type Documentation

16.5.4.1 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).

16.5.4.2 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.

16.5.4.3 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.

16.5.4.4 enum i2c_slave_bus_speed_t

16.5.4.5 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.

16.5.5 Function Documentation

16.5.5.1 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.

16.5.5.2 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.

16.5.5.3 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.

16.5.5.4 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.

16.5.5.5 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.

16.5.5.6 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.

16.5.5.7 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).

16.5.5.8 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).

16.5.5.9 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.

16.5.5.10 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 k-I2C_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.

16.5.5.11 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 k-I2C_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.

16.5.5.12 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 k-I2C_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.

16.5.5.13 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_SlaveAddressMatch-Event.

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.

16.5.5.14 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	

16.5.5.15 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.

16.5.5.16 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.
i2cHandle	Pointer to i2c_slave_handle_t structure which stores the transfer state.

Chapter 17
IOCON: I/O pin configuration

17.1 Overview

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

17.2 Function groups

17.2.1 Pin mux set

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

17.2.2 Pin mux set

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

17.3 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, 2))
  IOCON driver version 2.0.2.

17.4 Data Structure Documentation

17.4.1 struct iocon_group_t

17.5 Macro Definition Documentation

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

17.6 Function Documentation

17.6.1 __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

17.6.2 __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 18
SPI: Serial Peripheral Interface Driver

18.1 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. S-PI functional operation groups provide the functional API set.

Transactional APIs are transaction target high level APIs. Transactional APIs can be used to enable the peripheral and in the application if the code size and performance of transactional APIs satisfy the requirements. If the code size and performance are a critical requirement, see the transactional A-PI implementation and write a custom code. All transactional APIs use the spi_handle_t as the first parameter. Initialize the handle by calling the SPI_MasterTransferCreateHandle() or SPI_SlaveTransfer-CreateHandle() API.

Transactional APIs support asynchronous transfer. This means that the functions SPI_MasterTransferNon-Blocking() and SPI_SlaveTransferNonBlocking() set up the interrupt for data transfer. When the transfer completes, the upper layer is notified through a callback function with the kStatus_SPI_Idle status.

18.2 Typical use case

18.2.1 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

18.3 SPI Driver

18.3.1 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 (0x00U)
  SPI dummy transfer data, the data is sent while txBuff is NULL.
- #define SPI_RETRY_TIMES 0U /∗ Define to zero means keep waiting until the flag is assert/deassert. ∗/
  Retry times for waiting flag.

Typedefs

- typedef spi_master_handle_t spi_slave_handle_t
  Slave handle 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,
  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_data_width_t {
  kSPI_Data4Bits = 3,
  kSPI_Data5Bits = 4,
  kSPI_Data6Bits = 5,
  kSPI_Data7Bits = 6,
  kSPI_Data8Bits = 7,
  kSPI_Data9Bits = 8,
  kSPI_Data10Bits = 9,
  kSPI_Data11Bits = 10,
  kSPI_Data12Bits = 11,
  kSPI_Data13Bits = 12,
  kSPI_Data14Bits = 13,
  kSPI_Data15Bits = 14,
  kSPI_Data16Bits = 15 }
  Transfer data width.
- enum {
  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,
  kStatus_SPI_Timeout = MAKE_STATUS(kStatusGroup_LPC_MINISPI, 4) }
  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, 6))
  SPI driver version.

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.

18.3.2 Data Structure Documentation

18.3.2.1 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.

Field Documentation

(1) uint8_t spi_delay_config_t::preDelay

(2) uint8_t spi_delay_config_t::postDelay

(3) uint8_t spi_delay_config_t::frameDelay

(4) uint8_t spi_delay_config_t::transferDelay

18.3.2.2 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.

Field Documentation

(1) spi_delay_config_t spi_master_config_t::delayConfig

18.3.2.3 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.

18.3.2.4 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.

Field Documentation

(1) uint32_t spi_transfer_t::configFlags

18.3.2.5 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.

Field Documentation

(1) uint32_t spi_master_handle_t::lastCommand

18.3.3 Macro Definition Documentation

18.3.3.1 #define FSL_SPI_DRIVER_VERSION (MAKE_VERSION(2, 0, 6))

18.3.3.2 #define SPI_DUMMYDATA (0x00U)

18.3.3.3 #define SPI_RETRY_TIMES 0U /∗ Define to zero means keep waiting until the flag is assert/deassert. ∗/

18.3.4 Enumeration Type Documentation

18.3.4.1 enum _spi_xfer_option

Enumerator

kSPI_EndOfFrame Add delay at the end of each frame(the last clk edge).
kSPI_EndOfTransfer Re-assert the CS signal after transfer finishes to deselect slave.
kSPI_ReceiveIgnore Ignore the receive data.

18.3.4.2 enum spi_shift_direction_t

Enumerator

kSPI_MsbFirst Data transfers start with most significant bit.
kSPI_LsbFirst Data transfers start with least significant bit.

18.3.4.3 enum spi_clock_polarity_t

Enumerator

kSPI_ClockPolarityActiveHigh Active-high SPI clock (idles low).
kSPI_ClockPolarityActiveLow Active-low SPI clock (idles high).

18.3.4.4 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.

18.3.4.5 enum spi_ssel_t

Enumerator

kSPI_Ssel0Assert Slave select 0.

18.3.4.6 enum spi_data_width_t

Enumerator

kSPI_Data4Bits	4 bits data width
kSPI_Data5Bits	5 bits data width
kSPI_Data6Bits	6 bits data width
kSPI_Data7Bits	7 bits data width
kSPI_Data8Bits	8 bits data width
kSPI_Data9Bits	9 bits data width
kSPI_Data10Bits	10 bits data width
kSPI_Data11Bits	11 bits data width
kSPI_Data12Bits	12 bits data width
kSPI_Data13Bits	13 bits data width
kSPI_Data14Bits	14 bits data width
kSPI_Data15Bits	15 bits data width
kSPI_Data16Bits	16 bits data width

18.3.4.7 anonymous enum

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.
kStatus_SPI_Timeout	SPI Timeout polling status flags.

18.3.4.8 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.

18.3.4.9 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.

18.3.5 Function Documentation

18.3.5.1 uint32_t SPI_GetInstance (SPI_Type ∗ *base*)

18.3.5.2 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

18.3.5.3 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_Master-GetDefaultConfig(). 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.

18.3.5.4 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

18.3.5.5 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

18.3.5.6 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

18.3.5.7 static void SPI_Enable (SPI_Type ∗ *base*, bool *enable*) `[inline]`, `[static]`

Parameters

base	SPI base pointer
enable	or disable (true = enable, false = disable)

18.3.5.8 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.

18.3.5.9 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.

18.3.5.10 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

18.3.5.11 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

18.3.5.12 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.

18.3.5.13 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.

18.3.5.14 static void SPI_WriteData (SPI_Type ∗ *base*, uint16_t *data*) [inline], [static]

Parameters

base	SPI base pointer
data	needs to be write.

18.3.5.15 static void SPI_WriteConfigFlags (SPI_Type ∗ *base*, uint32_t *configFlags*) [inline], [static]

Parameters

base	SPI base pointer
configFlags	control command needs to be written.

18.3.5.16 void SPI_WriteDataWithConfigFlags (SPI_Type ∗ *base*, uint16_t *data*, uint32_t *configFlags*)

Parameters

base	SPI base pointer
data	value needs to be written.
configFlags	control command needs to be written.

18.3.5.17 static uint32_t SPI_ReadData (SPI_Type ∗ *base*) [inline], [static]

Parameters

base	SPI base pointer

Returns

Data in the register.

18.3.5.18 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.

18.3.5.19 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.

18.3.5.20 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.
kStatus_SPI_Timeout	The transfer timed out and was aborted.

18.3.5.21 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.

18.3.5.22 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.

18.3.5.23 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.

18.3.5.24 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.

18.3.5.25 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.

18.3.5.26 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.

18.3.5.27 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.

18.3.5.28 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.

18.3.5.29 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.

18.3.5.30 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 19
USART: Universal Asynchronous Receiver/Transmitter Driver

19.1 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.

19.2 Typical use case

19.2.1 USART Send/receive using a polling method

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

19.2.2 USART Send/receive using an interrupt method

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

19.2.3 USART Receive using the ringbuffer feature

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

19.2.4 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

19.3 USART Driver

19.3.1 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.
- #define UART_RETRY_TIMES 0U
  Retry times for waiting flag.

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 {
  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,
  kStatus_USART_Timeout = MAKE_STATUS(kStatusGroup_LPC_USART, 12) }

Error codes for the USART driver.
- 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_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_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) }
USART status flags.

Driver version

- #define FSL_USART_DRIVER_VERSION (MAKE_VERSION(2, 5, 0))
  USART driver version.

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.
- status_t 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.

19.3.2 Data Structure Documentation

19.3.2.1 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.
- bool enableHardwareFlowControl
  Enable hardware control RTS/CTS.
- 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.

Field Documentation

(1) bool usart_config_t::enableRx

(2) bool usart_config_t::enableTx

(3) bool usart_config_t::enableContinuousSCLK

(4) usart_sync_mode_t usart_config_t::syncMode

(5) usart_clock_polarity_t usart_config_t::clockPolarity

19.3.2.2 struct usart_transfer_t

Data Fields

- size_t dataSize
  The byte count to be transfer.
- uint8_t ∗ data
  The buffer of data to be transfer.
- uint8_t ∗ rxData
  The buffer to receive data.
- const uint8_t ∗ txData
  The buffer of data to be sent.

Field Documentation

(1) uint8_t∗ usart_transfer_t::data

(2) uint8_t∗ usart_transfer_t::rxData

(3) const uint8_t∗ usart_transfer_t::txData

(4) size_t usart_transfer_t::dataSize

19.3.2.3 struct _usart_handle

Data Fields

- const uint8_t ∗volatile txData
  Address of remaining data to send.
- volatile size_t txDataSize
  Size of the remaining data to send.
- size_t txDataSizeAll
  Size of the data to send out.
- uint8_t ∗volatile rxData
  Address of remaining data to receive.
- volatile size_t rxDataSize
  Size of the remaining data to receive.
- size_t rxDataSizeAll
  Size of the data to receive.

- uint8_t ∗ rxRingBuffer
  Start address of the receiver ring buffer.
- size_t rxRingBufferSize
  Size of the ring buffer.
- volatile uint16_t rxRingBufferHead
  Index for the driver to store received data into ring buffer.
- volatile uint16_t rxRingBufferTail
  Index for the user to get data from the ring buffer.
- 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.

Field Documentation

(1) const uint8_t∗ volatile usart_handle_t::txData

(2) volatile size_t usart_handle_t::txDataSize

(3) size_t usart_handle_t::txDataSizeAll

(4) uint8_t∗ volatile usart_handle_t::rxData

(5) volatile size_t usart_handle_t::rxDataSize

(6) size_t usart_handle_t::rxDataSizeAll

(7) uint8_t∗ usart_handle_t::rxRingBuffer

(8) size_t usart_handle_t::rxRingBufferSize

(9) volatile uint16_t usart_handle_t::rxRingBufferHead

(10) volatile uint16_t usart_handle_t::rxRingBufferTail

(11) usart_transfer_callback_t usart_handle_t::callback

(12) void∗ usart_handle_t::userData

(13) volatile uint8_t usart_handle_t::txState

19.3.3 Macro Definition Documentation

19.3.3.1 #define FSL_USART_DRIVER_VERSION (MAKE_VERSION(2, 5, 0))

19.3.3.2 #define FSL_SDK_ENABLE_USART_DRIVER_TRANSACTIONAL_APIS 1

1 for enable, 0 for disable.

19.3.3.3 #define FSL_SDK_USART_DRIVER_ENABLE_BAUDRATE_AUTO_GENERATE 1

1 for enable, 0 for disable

19.3.3.4 #define UART_RETRY_TIMES 0U

Defining to zero means to keep waiting for the flag until it is assert/deassert.

19.3.4 Typedef Documentation

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

19.3.5 Enumeration Type Documentation

19.3.5.1 anonymous enum

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.
kStatus_USART_Timeout	USART times out.

19.3.5.2 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.

19.3.5.3 enum usart_sync_mode_t

Enumerator

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

19.3.5.4 enum usart_stop_bit_count_t

Enumerator

kUSART_OneStopBit	One stop bit.
kUSART_TwoStopBit	Two stop bits.

19.3.5.5 enum usart_data_len_t

Enumerator

kUSART_7BitsPerChar	Seven bit mode.
kUSART_8BitsPerChar	Eight bit mode.

19.3.5.6 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.

19.3.5.7 enum _usart_interrupt_enable

Enumerator

kUSART_RxReadyInterruptEnable	Receive ready interrupt.
kUSART_TxReadyInterruptEnable	Transmit ready 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_AllInterruptEnable	All interrupt.

19.3.5.8 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.

19.3.6 Function Documentation

19.3.6.1 uint32_t USART_GetInstance (USART_Type ∗ *base*)

19.3.6.2 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

19.3.6.3 void USART_Deinit (USART_Type ∗ *base*)

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

Parameters

base	USART peripheral base address.

19.3.6.4 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; usart-Config->stopBitCount = kUSART_OneStopBit; usartConfig->bitCountPerChar = kUSART_8BitsPer-Char; usartConfig->loopback = false; usartConfig->enableTx = false; usartConfig->enableRx = false; ...

Parameters

config	Pointer to configuration structure.

19.3.6.5 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.

19.3.6.6 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.

19.3.6.7 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.

19.3.6.8 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.

19.3.6.9 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.

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

This function returns the enabled USART interrupts.

Parameters

base	USART peripheral base address.

19.3.6.11 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.

19.3.6.12 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.

19.3.6.13 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.

19.3.6.14 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.

19.3.6.15 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.

19.3.6.16 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.

19.3.6.17 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.

19.3.6.18 status_t 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.

Return values

kStatus_USART_Timeout	Transmission timed out and was aborted.
kStatus_Success	Successfully wrote all data.

19.3.6.19 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_USART_Timeout	Transmission timed out and was aborted.
kStatus_Success	Successfully received all data.

19.3.6.20 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.

19.3.6.21 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.

19.3.6.22 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.

19.3.6.23 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.

19.3.6.24 size_t USART_TransferGetRxRingBufferLength (usart_handle_t ∗ *handle*)

Parameters

handle	USART handle pointer.

Returns

Length of received data in RX ring buffer.

19.3.6.25 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.

19.3.6.26 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_NoTransferIn-Progress	No send in progress.
kStatus_InvalidArgument	Parameter is invalid.
kStatus_Success	Get successfully through the parameter `count`;

19.3.6.27 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.

19.3.6.28 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.

19.3.6.29 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`;

19.3.6.30 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 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, 3))
  Version 2.0.3.

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.
- static void MRT_ReleaseChannel (MRT_Type ∗base, mrt_chnl_t channel)
  Release the channel when the timer is using the multi-task mode.

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. Count can contain the LOAD bit, which control the force load feature.

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

20.6.15 static void MRT_ReleaseChannel (MRT_Type * *base,* mrt_chnl_t *channel*) `[inline]`, `[static]`

In multi-task mode, the INUSE flags allow more control over when MRT channels are released for further use. The user can hold on to a channel acquired by calling MRT_GetIdleChannel() for as long as it is needed and release it by calling this function. This removes the need to ask for an available channel for every use.

Parameters

base	Multi-Rate timer peripheral base address
channel	Timer channel number.

Chapter 21
PINT: Pin Interrupt and Pattern Match Driver

21.1 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.

21.2 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.

21.2.1 Pin Interrupt use case

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

21.2.2 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,
  kPINT_SecPatternMatchInp0Src = 0U,
  kPINT_SecPatternMatchInp1Src = 1U }
  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, 11))

21.3 Typedef Documentation

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

21.4 Enumeration Type Documentation

21.4.1 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.

21.4.2 enum pint_pin_int_t

Enumerator

kPINT_PinInt0	Pin Interrupt 0.

21.4.3 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.
kPINT_SecPatternMatchInp0Src	Input source 0.
kPINT_SecPatternMatchInp1Src	Input source 1.

21.4.4 enum pint_pmatch_bslice_t

Enumerator

kPINT_PatternMatchBSlice0	Bit slice 0.

21.4.5 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.

21.5 Function Documentation

21.5.1 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.	

21.5.2 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.	

21.5.3 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.	

21.5.4 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.	

21.5.5 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.

21.5.6 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.	

21.5.7 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.

21.5.8 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.	

21.5.9 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.

21.5.10 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.	

21.5.11 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.

21.5.12 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.	

21.5.13 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.

21.5.14 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.	

21.5.15 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.

21.5.16 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.	

21.5.17 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.	

21.5.18 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.

21.5.19 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.

21.5.20 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.

21.5.21 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.	

21.5.22 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.	

21.5.23 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.	

21.5.24 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.	

21.5.25 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.	

21.5.26 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.	

21.5.27 void PINT_Deinit (PINT_Type * *base*)

This function disables the PINT clock.

Parameters

base	Base address of the PINT peripheral.

Return values

None.	

21.5.28 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.
pintIdx	pin index.

Return values

None.	

21.5.29 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.
pintIdx	pin index.

Return values

None.	

Chapter 22
PLU: Programmable Logic Unit

22.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Programmable Logic Unit module of MCUXpresso SDK devices.

22.2 Function groups

The PLU driver supports the creation of small combinatorial and/or sequential logic networks including simple state machines.

22.2.1 Initialization and de-initialization

The function PLU_Init() enables the PLU clock and reset the module.

The function PIT_Deinit() gates the PLU clock.

22.2.2 Set input/output source and Truth Table

The function PLU_SetLutInputSource() sets the input source for the LUT element.

The function PLU_SetOutputSource() sets output source of the PLU module.

The function PLU_SetLutTruthTable() sets the truth table for the LUT element.

22.2.3 Read current Output State

The function PLU_ReadOutputState() reads the current state of the 8 designated PLU Outputs.

22.2.4 Wake-up/Interrupt Control

The function PLU_EnableWakeIntRequest() enables the wake-up/interrupt request on a PLU output pin with a optional configuration to eliminate the glitches. The function PLU_GetDefaultWakeIntConfig() gets the default configuration which can be used in a case with a given PLU_CLKIN.

The function PLU_LatchInterrupt() latches the interrupt and it can be cleared by function PLU_ClearLatchedInterrupt().

22.3 Typical use case

22.3.1 PLU combination example

Create a simple combinatorial logic network to control the LED. Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/plu/combination

Enumerations

- enum plu_lut_index_t {
  kPLU_LUT_0 = 0U,
  kPLU_LUT_1 = 1U,
  kPLU_LUT_2 = 2U,
  kPLU_LUT_3 = 3U,
  kPLU_LUT_4 = 4U,
  kPLU_LUT_5 = 5U,
  kPLU_LUT_6 = 6U,
  kPLU_LUT_7 = 7U,
  kPLU_LUT_8 = 8U,
  kPLU_LUT_9 = 9U,
  kPLU_LUT_10 = 10U,
  kPLU_LUT_11 = 11U,
  kPLU_LUT_12 = 12U,
  kPLU_LUT_13 = 13U,
  kPLU_LUT_14 = 14U,
  kPLU_LUT_15 = 15U,
  kPLU_LUT_16 = 16U,
  kPLU_LUT_17 = 17U,
  kPLU_LUT_18 = 18U,
  kPLU_LUT_19 = 19U,
  kPLU_LUT_20 = 20U,
  kPLU_LUT_21 = 21U,
  kPLU_LUT_22 = 22U,
  kPLU_LUT_23 = 23U,
  kPLU_LUT_24 = 24U,
  kPLU_LUT_25 = 25U }

  Index of LUT.
- enum plu_lut_in_index_t {
  kPLU_LUT_IN_0 = 0U,
  kPLU_LUT_IN_1 = 1U,
  kPLU_LUT_IN_2 = 2U,
  kPLU_LUT_IN_3 = 3U,
  kPLU_LUT_IN_4 = 4U }

  Inputs of LUT.
- enum plu_lut_input_source_t {

kPLU_LUT_IN_SRC_PLU_IN_0 = 0U,
kPLU_LUT_IN_SRC_PLU_IN_1 = 1U,
kPLU_LUT_IN_SRC_PLU_IN_2 = 2U,
kPLU_LUT_IN_SRC_PLU_IN_3 = 3U,
kPLU_LUT_IN_SRC_PLU_IN_4 = 4U,
kPLU_LUT_IN_SRC_PLU_IN_5 = 5U,
kPLU_LUT_IN_SRC_LUT_OUT_0 = 6U,
kPLU_LUT_IN_SRC_LUT_OUT_1 = 7U,
kPLU_LUT_IN_SRC_LUT_OUT_2 = 8U,
kPLU_LUT_IN_SRC_LUT_OUT_3 = 9U,
kPLU_LUT_IN_SRC_LUT_OUT_4 = 10U,
kPLU_LUT_IN_SRC_LUT_OUT_5 = 11U,
kPLU_LUT_IN_SRC_LUT_OUT_6 = 12U,
kPLU_LUT_IN_SRC_LUT_OUT_7 = 13U,
kPLU_LUT_IN_SRC_LUT_OUT_8 = 14U,
kPLU_LUT_IN_SRC_LUT_OUT_9 = 15U,
kPLU_LUT_IN_SRC_LUT_OUT_10 = 16U,
kPLU_LUT_IN_SRC_LUT_OUT_11 = 17U,
kPLU_LUT_IN_SRC_LUT_OUT_12 = 18U,
kPLU_LUT_IN_SRC_LUT_OUT_13 = 19U,
kPLU_LUT_IN_SRC_LUT_OUT_14 = 20U,
kPLU_LUT_IN_SRC_LUT_OUT_15 = 21U,
kPLU_LUT_IN_SRC_LUT_OUT_16 = 22U,
kPLU_LUT_IN_SRC_LUT_OUT_17 = 23U,
kPLU_LUT_IN_SRC_LUT_OUT_18 = 24U,
kPLU_LUT_IN_SRC_LUT_OUT_19 = 25U,
kPLU_LUT_IN_SRC_LUT_OUT_20 = 26U,
kPLU_LUT_IN_SRC_LUT_OUT_21 = 27U,
kPLU_LUT_IN_SRC_LUT_OUT_22 = 28U,
kPLU_LUT_IN_SRC_LUT_OUT_23 = 29U,
kPLU_LUT_IN_SRC_LUT_OUT_24 = 30U,
kPLU_LUT_IN_SRC_LUT_OUT_25 = 31U,
kPLU_LUT_IN_SRC_FLIPFLOP_0 = 32U,
kPLU_LUT_IN_SRC_FLIPFLOP_1 = 33U,
kPLU_LUT_IN_SRC_FLIPFLOP_2 = 34U,
kPLU_LUT_IN_SRC_FLIPFLOP_3 = 35U }

Available sources of LUT input.

- enum plu_output_index_t {
kPLU_OUTPUT_0 = 0U,
kPLU_OUTPUT_1 = 1U,
kPLU_OUTPUT_2 = 2U,
kPLU_OUTPUT_3 = 3U,
kPLU_OUTPUT_4 = 4U,
kPLU_OUTPUT_5 = 5U,
kPLU_OUTPUT_6 = 6U,

kPLU_OUTPUT_7 = 7U }

PLU output multiplexer registers.

- enum plu_output_source_t {
kPLU_OUT_SRC_LUT_0 = 0U,
kPLU_OUT_SRC_LUT_1 = 1U,
kPLU_OUT_SRC_LUT_2 = 2U,
kPLU_OUT_SRC_LUT_3 = 3U,
kPLU_OUT_SRC_LUT_4 = 4U,
kPLU_OUT_SRC_LUT_5 = 5U,
kPLU_OUT_SRC_LUT_6 = 6U,
kPLU_OUT_SRC_LUT_7 = 7U,
kPLU_OUT_SRC_LUT_8 = 8U,
kPLU_OUT_SRC_LUT_9 = 9U,
kPLU_OUT_SRC_LUT_10 = 10U,
kPLU_OUT_SRC_LUT_11 = 11U,
kPLU_OUT_SRC_LUT_12 = 12U,
kPLU_OUT_SRC_LUT_13 = 13U,
kPLU_OUT_SRC_LUT_14 = 14U,
kPLU_OUT_SRC_LUT_15 = 15U,
kPLU_OUT_SRC_LUT_16 = 16U,
kPLU_OUT_SRC_LUT_17 = 17U,
kPLU_OUT_SRC_LUT_18 = 18U,
kPLU_OUT_SRC_LUT_19 = 19U,
kPLU_OUT_SRC_LUT_20 = 20U,
kPLU_OUT_SRC_LUT_21 = 21U,
kPLU_OUT_SRC_LUT_22 = 22U,
kPLU_OUT_SRC_LUT_23 = 23U,
kPLU_OUT_SRC_LUT_24 = 24U,
kPLU_OUT_SRC_LUT_25 = 25U,
kPLU_OUT_SRC_FLIPFLOP_0 = 26U,
kPLU_OUT_SRC_FLIPFLOP_1 = 27U,
kPLU_OUT_SRC_FLIPFLOP_2 = 28U,
kPLU_OUT_SRC_FLIPFLOP_3 = 29U }

  Available sources of PLU output.

Driver version

- #define FSL_PLU_DRIVER_VERSION (MAKE_VERSION(2, 2, 1))

  Version 2.2.1.

Initialization and deinitialization

- void PLU_Init (PLU_Type ∗base)

  Enable the PLU clock and reset the module.
- void PLU_Deinit (PLU_Type ∗base)

  Gate the PLU clock.

Set input/output source and Truth Table

- static void PLU_SetLutInputSource (PLU_Type ∗base, plu_lut_index_t lutIndex, plu_lut_in_index_t lutInIndex, plu_lut_input_source_t inputSrc)
  Set Input source of LUT.
- static void PLU_SetOutputSource (PLU_Type ∗base, plu_output_index_t outputIndex, plu_output_source_t outputSrc)
  Set Output source of PLU.
- static void PLU_SetLutTruthTable (PLU_Type ∗base, plu_lut_index_t lutIndex, uint32_t truthTable)
  Set Truth Table of LUT.

Read current Output State

- static uint32_t PLU_ReadOutputState (PLU_Type ∗base)
  Read the current state of the 8 designated PLU Outputs.

22.4 Enumeration Type Documentation

22.4.1 enum plu_lut_index_t

Enumerator

kPLU_LUT_0	5-input Look-up Table 0
kPLU_LUT_1	5-input Look-up Table 1
kPLU_LUT_2	5-input Look-up Table 2
kPLU_LUT_3	5-input Look-up Table 3
kPLU_LUT_4	5-input Look-up Table 4
kPLU_LUT_5	5-input Look-up Table 5
kPLU_LUT_6	5-input Look-up Table 6
kPLU_LUT_7	5-input Look-up Table 7
kPLU_LUT_8	5-input Look-up Table 8
kPLU_LUT_9	5-input Look-up Table 9
kPLU_LUT_10	5-input Look-up Table 10
kPLU_LUT_11	5-input Look-up Table 11
kPLU_LUT_12	5-input Look-up Table 12
kPLU_LUT_13	5-input Look-up Table 13
kPLU_LUT_14	5-input Look-up Table 14
kPLU_LUT_15	5-input Look-up Table 15
kPLU_LUT_16	5-input Look-up Table 16
kPLU_LUT_17	5-input Look-up Table 17
kPLU_LUT_18	5-input Look-up Table 18
kPLU_LUT_19	5-input Look-up Table 19
kPLU_LUT_20	5-input Look-up Table 20
kPLU_LUT_21	5-input Look-up Table 21
kPLU_LUT_22	5-input Look-up Table 22
kPLU_LUT_23	5-input Look-up Table 23

kPLU_LUT_24	5-input Look-up Table 24
kPLU_LUT_25	5-input Look-up Table 25

22.4.2 enum plu_lut_in_index_t

5 input present for each LUT.

Enumerator

kPLU_LUT_IN_0	LUT input 0.
kPLU_LUT_IN_1	LUT input 1.
kPLU_LUT_IN_2	LUT input 2.
kPLU_LUT_IN_3	LUT input 3.
kPLU_LUT_IN_4	LUT input 4.

22.4.3 enum plu_lut_input_source_t

Enumerator

kPLU_LUT_IN_SRC_PLU_IN_0	Select PLU input 0 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_PLU_IN_1	Select PLU input 1 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_PLU_IN_2	Select PLU input 2 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_PLU_IN_3	Select PLU input 3 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_PLU_IN_4	Select PLU input 4 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_PLU_IN_5	Select PLU input 5 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_LUT_OUT_0	Select LUT output 0 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_LUT_OUT_1	Select LUT output 1 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_LUT_OUT_2	Select LUT output 2 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_LUT_OUT_3	Select LUT output 3 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_LUT_OUT_4	Select LUT output 4 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_LUT_OUT_5	Select LUT output 5 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_LUT_OUT_6	Select LUT output 6 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_LUT_OUT_7	Select LUT output 7 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_LUT_OUT_8	Select LUT output 8 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_LUT_OUT_9	Select LUT output 9 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_LUT_OUT_10	Select LUT output 10 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_LUT_OUT_11	Select LUT output 11 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_LUT_OUT_12	Select LUT output 12 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_LUT_OUT_13	Select LUT output 13 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_LUT_OUT_14	Select LUT output 14 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_LUT_OUT_15	Select LUT output 15 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_LUT_OUT_16	Select LUT output 16 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_LUT_OUT_17	Select LUT output 17 to be connected to LUTn Input x.

kPLU_LUT_IN_SRC_LUT_OUT_18	Select LUT output 18 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_LUT_OUT_19	Select LUT output 19 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_LUT_OUT_20	Select LUT output 20 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_LUT_OUT_21	Select LUT output 21 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_LUT_OUT_22	Select LUT output 22 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_LUT_OUT_23	Select LUT output 23 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_LUT_OUT_24	Select LUT output 24 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_LUT_OUT_25	Select LUT output 25 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_FLIPFLOP_0	Select Flip-Flops state 0 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_FLIPFLOP_1	Select Flip-Flops state 1 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_FLIPFLOP_2	Select Flip-Flops state 2 to be connected to LUTn Input x.
kPLU_LUT_IN_SRC_FLIPFLOP_3	Select Flip-Flops state 3 to be connected to LUTn Input x.

22.4.4 enum plu_output_index_t

Enumerator

kPLU_OUTPUT_0	PLU OUTPUT 0.
kPLU_OUTPUT_1	PLU OUTPUT 1.
kPLU_OUTPUT_2	PLU OUTPUT 2.
kPLU_OUTPUT_3	PLU OUTPUT 3.
kPLU_OUTPUT_4	PLU OUTPUT 4.
kPLU_OUTPUT_5	PLU OUTPUT 5.
kPLU_OUTPUT_6	PLU OUTPUT 6.
kPLU_OUTPUT_7	PLU OUTPUT 7.

22.4.5 enum plu_output_source_t

Enumerator

kPLU_OUT_SRC_LUT_0	Select LUT0 output to be connected to PLU output.
kPLU_OUT_SRC_LUT_1	Select LUT1 output to be connected to PLU output.
kPLU_OUT_SRC_LUT_2	Select LUT2 output to be connected to PLU output.
kPLU_OUT_SRC_LUT_3	Select LUT3 output to be connected to PLU output.
kPLU_OUT_SRC_LUT_4	Select LUT4 output to be connected to PLU output.
kPLU_OUT_SRC_LUT_5	Select LUT5 output to be connected to PLU output.
kPLU_OUT_SRC_LUT_6	Select LUT6 output to be connected to PLU output.
kPLU_OUT_SRC_LUT_7	Select LUT7 output to be connected to PLU output.
kPLU_OUT_SRC_LUT_8	Select LUT8 output to be connected to PLU output.
kPLU_OUT_SRC_LUT_9	Select LUT9 output to be connected to PLU output.
kPLU_OUT_SRC_LUT_10	Select LUT10 output to be connected to PLU output.
kPLU_OUT_SRC_LUT_11	Select LUT11 output to be connected to PLU output.

kPLU_OUT_SRC_LUT_12	Select LUT12 output to be connected to PLU output.
kPLU_OUT_SRC_LUT_13	Select LUT13 output to be connected to PLU output.
kPLU_OUT_SRC_LUT_14	Select LUT14 output to be connected to PLU output.
kPLU_OUT_SRC_LUT_15	Select LUT15 output to be connected to PLU output.
kPLU_OUT_SRC_LUT_16	Select LUT16 output to be connected to PLU output.
kPLU_OUT_SRC_LUT_17	Select LUT17 output to be connected to PLU output.
kPLU_OUT_SRC_LUT_18	Select LUT18 output to be connected to PLU output.
kPLU_OUT_SRC_LUT_19	Select LUT19 output to be connected to PLU output.
kPLU_OUT_SRC_LUT_20	Select LUT20 output to be connected to PLU output.
kPLU_OUT_SRC_LUT_21	Select LUT21 output to be connected to PLU output.
kPLU_OUT_SRC_LUT_22	Select LUT22 output to be connected to PLU output.
kPLU_OUT_SRC_LUT_23	Select LUT23 output to be connected to PLU output.
kPLU_OUT_SRC_LUT_24	Select LUT24 output to be connected to PLU output.
kPLU_OUT_SRC_LUT_25	Select LUT25 output to be connected to PLU output.
kPLU_OUT_SRC_FLIPFLOP_0	Select Flip-Flops state(0) to be connected to PLU output.
kPLU_OUT_SRC_FLIPFLOP_1	Select Flip-Flops state(1) to be connected to PLU output.
kPLU_OUT_SRC_FLIPFLOP_2	Select Flip-Flops state(2) to be connected to PLU output.
kPLU_OUT_SRC_FLIPFLOP_3	Select Flip-Flops state(3) to be connected to PLU output.

22.5 Function Documentation

22.5.1 void PLU_Init (PLU_Type ∗ *base*)

Note

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

Parameters

base	PLU peripheral base address

22.5.2 void PLU_Deinit (PLU_Type ∗ *base*)

Parameters

base	PLU peripheral base address

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

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

Parameters

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

22.5.4 static void PLU_SetOutputSource (PLU_Type ∗ *base*, plu_output_index_t *outputIndex*, plu_output_source_t *outputSrc*) `[inline]`, `[static]`

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

Parameters

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

22.5.5 static void PLU_SetLutTruthTable (PLU_Type ∗ *base*, plu_lut_index_t *lutIndex*, uint32_t *truthTable*) `[inline]`, `[static]`

Parameters

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

22.5.6 static uint32_t PLU_ReadOutputState (PLU_Type ∗ *base*) `[inline]`, `[static]`

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

Parameters

base	PLU peripheral base address.

Returns

Current PLU output state value.

Chapter 23
SWM: Switch Matrix Module

23.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Switch Matrix Module (SWM) module of MCUXpresso SDK devices.

23.2 SWM: Switch Matrix Module

23.2.1 SWM Operations

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.

swm connections

- enum swm_fixed_port_pin_type_t {
  kSWM_PLU_INPUT0_PortPin_P0_0 = 0x00U,
  kSWM_PLU_INPUT0_PortPin_P0_8 = 0x01U,
  kSWM_PLU_INPUT0_PortPin_P0_17 = 0x02U,
  kSWM_PLU_INPUT1_PortPin_P0_1 = 0x00U,
  kSWM_PLU_INPUT1_PortPin_P0_9 = 0x01U,
  kSWM_PLU_INPUT1_PortPin_P0_18 = 0x02U,
  kSWM_PLU_INPUT2_PortPin_P0_2 = 0x00U,
  kSWM_PLU_INPUT2_PortPin_P0_10 = 0x01U,
  kSWM_PLU_INPUT2_PortPin_P0_19 = 0x02U,
  kSWM_PLU_INPUT3_PortPin_P0_3 = 0x00U,
  kSWM_PLU_INPUT3_PortPin_P0_11 = 0x01U,
  kSWM_PLU_INPUT3_PortPin_P0_20 = 0x02U,
  kSWM_PLU_INPUT4_PortPin_P0_4 = 0x00U,
  kSWM_PLU_INPUT4_PortPin_P0_12 = 0x01U,
  kSWM_PLU_INPUT4_PortPin_P0_21 = 0x02U,
  kSWM_PLU_INPUT5_PortPin_P0_5 = 0x00U,
  kSWM_PLU_INPUT5_PortPin_P0_13 = 0x01U,
  kSWM_PLU_INPUT5_PortPin_P0_22 = 0x02U,
  kSWM_PLU_OUT0_PortPin_P0_7 = 0x00U,
  kSWM_PLU_OUT0_PortPin_P0_14 = 0x01U,
  kSWM_PLU_OUT0_PortPin_P0_23 = 0x02U,
  kSWM_PLU_OUT1_PortPin_P0_8 = 0x00U,
  kSWM_PLU_OUT1_PortPin_P0_15 = 0x01U,
  kSWM_PLU_OUT1_PortPin_P0_24 = 0x02U,
  kSWM_PLU_OUT2_PortPin_P0_9 = 0x00U,
  kSWM_PLU_OUT2_PortPin_P0_16 = 0x01U,
  kSWM_PLU_OUT2_PortPin_P0_25 = 0x02U,
  kSWM_PLU_OUT3_PortPin_P0_10 = 0x00U,
  kSWM_PLU_OUT3_PortPin_P0_17 = 0x01U,
  kSWM_PLU_OUT3_PortPin_P0_26 = 0x02U,
  kSWM_PLU_OUT4_PortPin_P0_11 = 0x00U,
  kSWM_PLU_OUT4_PortPin_P0_18 = 0x01U,
  kSWM_PLU_OUT4_PortPin_P0_27 = 0x02U,
  kSWM_PLU_OUT5_PortPin_P0_12 = 0x00U,
  kSWM_PLU_OUT5_PortPin_P0_19 = 0x01U,
  kSWM_PLU_OUT5_PortPin_P0_28 = 0x02U,
  kSWM_PLU_OUT6_PortPin_P0_13 = 0x00U,
  kSWM_PLU_OUT6_PortPin_P0_20 = 0x01U,
  kSWM_PLU_OUT6_PortPin_P0_29 = 0x02U,
  kSWM_PLU_OUT7_PortPin_P0_14 = 0x00U,
  kSWM_PLU_OUT7_PortPin_P0_21 = 0x01U,
  kSWM_PLU_OUT7_PortPin_P0_30 = 0x02U }

SWM pinassignfixed_port_pin number.

- 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_Reset = 0xffU }

  SWM port_pin number.
- enum swm_select_fixed_movable_t {

kSWM_PLU_INPUT0 = 0U,
kSWM_PLU_INPUT1 = 1U,
kSWM_PLU_INPUT2 = 2U,
kSWM_PLU_INPUT3 = 3U,
kSWM_PLU_INPUT4 = 4U,
kSWM_PLU_INPUT5 = 5U,
kSWM_PLU_OUT0 = 6U,
kSWM_PLU_OUT1 = 7U,
kSWM_PLU_OUT2 = 8U,
kSWM_PLU_OUT3 = 9U,
kSWM_PLU_OUT4 = 10U,
kSWM_PLU_OUT5 = 11U,
kSWM_PLU_OUT6 = 12U,
kSWM_PLU_OUT7 = 13U,
kSWM_PINASSINGNFIXED_MOVABLE_NUM_FUNCS = 14U }

SWM pinassignfixed movable selection.

- 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_SCLK = 7U,
kSWM_SPI0_SCK = 8U,
kSWM_SPI0_MOSI = 9U,
kSWM_SPI0_MISO = 10U,
kSWM_SPI0_SSEL0 = 11U,
kSWM_SPI0_SSEL1 = 12U,
kSWM_T0_CAP_CHN0 = 13U,
kSWM_T0_CAP_CHN1 = 14U,
kSWM_T0_CAP_CHN2 = 15U,
kSWM_T0_MAT_CHN0 = 16U,
kSWM_T0_MAT_CHN1 = 17U,
kSWM_T0_MAT_CHN2 = 18U,
kSWM_T0_MAT_CHN3 = 19U,
kSWM_I2C0_SDA = 20U,
kSWM_I2C0_SCL = 21U,
kSWM_ACMP_OUT = 22U,
kSWM_CLKOUT = 23U,
kSWM_GPIO_INT_BMAT = 24U,
kSWM_LVLSHFT_IN0 = 25U,
kSWM_LVLSHFT_IN1 = 26U,
kSWM_LVLSHFT_OUT0 = 27U,
kSWM_LVLSHFT_OUT1 = 28U,
kSWM_I2C1_SDA = 29U,
kSWM_I2C1_SCL = 30U,
kSWM_PLU_CLKIN_IN = 31U,
kSWM_CAPT_X0 = 32U,
kSWM_CAPT_X1 = 33U,
kSWM_CAPT_X2 = 34U,
kSWM_CAPT_X3 = 35U,
kSWM_CAPT_X4 = 36U,
kSWM_CAPT_YL = 37U,
kSWM_CAPT_YH = 38U,
kSWM_MOVABLE_NUM_FUNCS = 39U }

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_SWCLK = SWM_PINENABLE0_SWCLK_MASK,
kSWM_SWDIO = SWM_PINENABLE0_SWDIO_MASK,
kSWM_RESETN = SWM_PINENABLE0_RESETN_MASK,
kSWM_CLKIN = SWM_PINENABLE0_CLKIN_MASK,
kSWM_WKCLKIN = SWM_PINENABLE0_WKCLKIN_MASK,
kSWM_VDDCMP = SWM_PINENABLE0_VDDCMP_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_ACMP_INPUT5 = SWM_PINENABLE0_ACMP_I5_MASK,
kSWM_DAC_OUT0 = SWM_PINENABLE0_DACOUT0_MASK,
kSWM_FIXEDPIN_NUM_FUNCS = (int)0x80000001U }
SWM fixed pin selection.

Driver version

- #define FSL_SWM_DRIVER_VERSION (MAKE_VERSION(2, 1, 2))
  LPC SWM driver version.

23.3 Macro Definition Documentation

23.3.1 #define FSL_SWM_DRIVER_VERSION (MAKE_VERSION(2, 1, 2))

23.4 Enumeration Type Documentation

23.4.1 enum swm_fixed_port_pin_type_t

Enumerator

kSWM_PLU_INPUT0_PortPin_P0_0	port_pin number P0_0.
kSWM_PLU_INPUT0_PortPin_P0_8	port_pin number P0_8.
kSWM_PLU_INPUT0_PortPin_P0_17	port_pin number P0_17.
kSWM_PLU_INPUT1_PortPin_P0_1	port_pin number P0_1.

kSWM_PLU_INPUT1_PortPin_P0_9 port_pin number P0_9.
kSWM_PLU_INPUT1_PortPin_P0_18 port_pin number P0_18.
kSWM_PLU_INPUT2_PortPin_P0_2 port_pin number P0_2.
kSWM_PLU_INPUT2_PortPin_P0_10 port_pin number P0_10.
kSWM_PLU_INPUT2_PortPin_P0_19 port_pin number P0_19.
kSWM_PLU_INPUT3_PortPin_P0_3 port_pin number P0_3.
kSWM_PLU_INPUT3_PortPin_P0_11 port_pin number P0_11.
kSWM_PLU_INPUT3_PortPin_P0_20 port_pin number P0_20.
kSWM_PLU_INPUT4_PortPin_P0_4 port_pin number P0_4.
kSWM_PLU_INPUT4_PortPin_P0_12 port_pin number P0_12.
kSWM_PLU_INPUT4_PortPin_P0_21 port_pin number P0_21.
kSWM_PLU_INPUT5_PortPin_P0_5 port_pin number P0_5.
kSWM_PLU_INPUT5_PortPin_P0_13 port_pin number P0_13.
kSWM_PLU_INPUT5_PortPin_P0_22 port_pin number P0_22.
kSWM_PLU_OUT0_PortPin_P0_7 port_pin number P0_7.
kSWM_PLU_OUT0_PortPin_P0_14 port_pin number P0_14.
kSWM_PLU_OUT0_PortPin_P0_23 port_pin number P0_23.
kSWM_PLU_OUT1_PortPin_P0_8 port_pin number P0_8.
kSWM_PLU_OUT1_PortPin_P0_15 port_pin number P0_15.
kSWM_PLU_OUT1_PortPin_P0_24 port_pin number P0_24.
kSWM_PLU_OUT2_PortPin_P0_9 port_pin number P0_9.
kSWM_PLU_OUT2_PortPin_P0_16 port_pin number P0_16.
kSWM_PLU_OUT2_PortPin_P0_25 port_pin number P0_25.
kSWM_PLU_OUT3_PortPin_P0_10 port_pin number P0_10.
kSWM_PLU_OUT3_PortPin_P0_17 port_pin number P0_17.
kSWM_PLU_OUT3_PortPin_P0_26 port_pin number P0_26.
kSWM_PLU_OUT4_PortPin_P0_11 port_pin number P0_11.
kSWM_PLU_OUT4_PortPin_P0_18 port_pin number P0_18.
kSWM_PLU_OUT4_PortPin_P0_27 port_pin number P0_27.
kSWM_PLU_OUT5_PortPin_P0_12 port_pin number P0_12.
kSWM_PLU_OUT5_PortPin_P0_19 port_pin number P0_19.
kSWM_PLU_OUT5_PortPin_P0_28 port_pin number P0_28.
kSWM_PLU_OUT6_PortPin_P0_13 port_pin number P0_13.
kSWM_PLU_OUT6_PortPin_P0_20 port_pin number P0_20.
kSWM_PLU_OUT6_PortPin_P0_29 port_pin number P0_29.
kSWM_PLU_OUT7_PortPin_P0_14 port_pin number P0_14.
kSWM_PLU_OUT7_PortPin_P0_21 port_pin number P0_21.
kSWM_PLU_OUT7_PortPin_P0_30 port_pin number P0_30.

23.4.2 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_Reset	port_pin reset number.

23.4.3 enum swm_select_fixed_movable_t

Enumerator

kSWM_PLU_INPUT0	Movable function as PLU_INPUT0.
kSWM_PLU_INPUT1	Movable function as PLU_INPUT1.
kSWM_PLU_INPUT2	Movable function as PLU_INPUT2.
kSWM_PLU_INPUT3	Movable function as PLU_INPUT3.
kSWM_PLU_INPUT4	Movable function as PLU_INPUT4.
kSWM_PLU_INPUT5	Movable function as PLU_INPUT5.
kSWM_PLU_OUT0	Movable function as PLU_OUT0.

kSWM_PLU_OUT1	Movable function as PLU_OUT1.
kSWM_PLU_OUT2	Movable function as PLU_OUT2.
kSWM_PLU_OUT3	Movable function as PLU_OUT3.
kSWM_PLU_OUT4	Movable function as PLU_OUT4.
kSWM_PLU_OUT5	Movable function as PLU_OUT5.
kSWM_PLU_OUT6	Movable function as PLU_OUT6.
kSWM_PLU_OUT7	Movable function as PLU_OUT7.
kSWM_PINASSINGNFIXED_MOVABLE_NUM_FUNCS	Movable function number.

23.4.4 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_SCLK	Movable function as USART1_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_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_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_I2C0_SDA	Movable function as I2C0_SDA.
kSWM_I2C0_SCL	Movable function as I2C0_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_LVLSHFT_IN0	Movable function as LVLSHFT_IN0.
kSWM_LVLSHFT_IN1	Movable function as LVLSHFT_IN1.
kSWM_LVLSHFT_OUT0	Movable function as LVLSHFT_OUT0.
kSWM_LVLSHFT_OUT1	Movable function as LVLSHFT_OUT1.
kSWM_I2C1_SDA	Movable function as I2C1_SDA.
kSWM_I2C1_SCL	Movable function as I2C1_SCL.

kSWM_PLU_CLKIN_IN Movable function as PLU_CLKIN_IN.
kSWM_CAPT_X0 Movable function as CAPT_X0.
kSWM_CAPT_X1 Movable function as CAPT_X1.
kSWM_CAPT_X2 Movable function as CAPT_X2.
kSWM_CAPT_X3 Movable function as CAPT_X3.
kSWM_CAPT_X4 Movable function as CAPT_X4.
kSWM_CAPT_YL Movable function as CAPT_YL.
kSWM_CAPT_YH Movable function as CAPT_YH.
kSWM_MOVABLE_NUM_FUNCS Movable function number.

23.4.5 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_SWCLK Fixed-pin function as SWCLK.
kSWM_SWDIO Fixed-pin function as SWDIO.
kSWM_RESETN Fixed-pin function as RESETN.
kSWM_CLKIN Fixed-pin function as CLKIN.
kSWM_WKCLKIN Fixed-pin function as WKCLKIN.
kSWM_VDDCMP Fixed-pin function as VDDCMP.
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_ACMP_INPUT5 Fixed-pin function as ACMP_INPUT5.
kSWM_DAC_OUT0 Fixed-pin function as DACOUT0.
kSWM_FIXEDPIN_NUM_FUNCS Fixed-pin function number.

23.5 Function Documentation

23.5.1 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.

23.5.2 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 24
SYSCON: System Configuration

24.1 Overview

The MCUXpresso SDK provides a peripheral clock and power driver for the SYSCON module of MCU-Xpresso 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.

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

Driver version

- #define FSL_SYSON_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))
  Group syscon driver version for SDK.

24.2 Macro Definition Documentation

24.2.1 #define FSL_SYSON_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

Version 2.0.1.

24.3 Enumeration Type Documentation

24.3.1 enum syscon_connection_t

Enumerator

kSYSCON_GpioPort0Pin0ToPintsel Pin Interrupt.

24.4 Function Documentation

24.4.1 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 25
WKT: Self-wake-up Timer

25.1 Overview

The MCUXpresso SDK provides a driver for the Self-wake-up Timer (WKT) of MCUXpresso SDK devices.

25.2 Function groups

The WKT driver supports operating the module as a time counter.

25.2.1 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.

25.2.2 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.

25.2.3 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.

25.2.4 Status

Provides functions to get and clear the WKT status flags.

25.3 Typical use case

25.3.1 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, 2))
  Version 2.0.2.

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.

25.4 Data Structure Documentation

25.4.1 struct wkt_config_t

Data Fields

- wkt_clock_source_t clockSource
  External or internal clock source select.

25.5 Enumeration Type Documentation

25.5.1 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, LP-OSCEN bit of DPDCTRL register must be enabled.
kWKT_ExternalClockSource WKT clock sourced from the Low power clock Use this clock, WA-KECLKPAD_DISABLE bit of DPDCTRL register must be enabled.

25.5.2 enum wkt_status_flags_t

Enumerator

kWKT_AlarmFlag Alarm flag.

25.6 Function Documentation

25.6.1 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.

25.6.2 void WKT_Deinit (WKT_Type ∗ *base*)

Parameters

base	WKT peripheral base address

25.6.3 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

25.6.4 static uint32_t WKT_GetCounterValue (WKT_Type ∗ *base*) [inline], [static]

Parameters

base	WKT peripheral base address

25.6.5 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

25.6.6 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

25.6.7 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

25.6.8 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 26
WWDT: Windowed Watchdog Timer Driver

26.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Watchdog module (WDOG) of MCUXpresso SDK devices.

26.2 Function groups

26.2.1 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.

26.2.2 Status

Provides functions to get and clear the WWDT status.

26.2.3 Interrupt

Provides functions to enable/disable WWDT interrupts and get current enabled interrupts.

26.2.4 Watch dog Refresh

The function WWDT_Refresh() feeds the WWDT.

26.3 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, 9))
  Defines WWDT driver version.

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.

26.4 Data Structure Documentation

26.4.1 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.

Field Documentation

(1) uint32_t wwdt_config_t::warningValue

Set this to 0 for no warning

(2) uint32_t wwdt_config_t::clockFreq_Hz

26.5 Macro Definition Documentation

26.5.1 #define FSL_WWDT_DRIVER_VERSION (MAKE_VERSION(2, 1, 9))

26.6 Enumeration Type Documentation

26.6.1 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.

26.7 Function Documentation

26.7.1 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

26.7.2 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

26.7.3 void WWDT_Deinit (WWDT_Type ∗ *base*)

This function shuts down the WWDT.

Parameters

base	WWDT peripheral base address

26.7.4 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

26.7.5 static void WWDT_Disable (WWDT_Type ∗ *base*) [inline], [static]

Deprecated Do not use this function. It will be deleted in next release version, for once the bit field of WDEN written with a 1, it can not be re-written with a 0.

This function write value into WWDT_MOD register to disable the WWDT.

Parameters

base	WWDT peripheral base address

26.7.6 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

26.7.7 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

26.7.8 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.

26.7.9 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.

26.7.10 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.

26.7.11 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 27
Debug Console Lite

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

specifier	Description

- Support a format specifier for SCANF following this prototype " %[*][width][length]specifier", which is explained below

*	Description
An optional starting asterisk indicates that the data is to be read from the stream but ignored. In other words, it is not stored in the corresponding argument.	

width	Description
This specifies the maximum number of characters to be read in the current reading operation.	

length	Description
hh	The argument is interpreted as a signed character or unsigned character (only applies to integer specifiers: i, d, o, u, x, and X).
h	The argument is interpreted as a short integer or unsigned short integer (only applies to integer specifiers: i, d, o, u, x, and X).
l	The argument is interpreted as a long integer or unsigned long integer for integer specifiers (i, d, o, u, x, and X) and as a wide character or wide character string for specifiers c and s.
ll	The argument is interpreted as a long long integer or unsigned long long integer for integer specifiers (i, d, o, u, x, and X) and as a wide character or wide character string for specifiers c and s.
L	The argument is interpreted as a long double (only applies to floating point specifiers: e, E, f, g, and G).
j or z or t	Not supported

specifier	Qualifying Input	Type of argument
c	Single character: Reads the next character. If a width different from 1 is specified, the function reads width characters and stores them in the successive locations of the array passed as argument. No null character is appended at the end.	char ∗
i	Integer: : Number optionally preceded with a + or - sign	int ∗
d	Decimal integer: Number optionally preceded with a + or - sign	int ∗
a, A, e, E, f, F, g, G	Floating point: Decimal number containing a decimal point, optionally preceded by a + or - sign and optionally followed by the e or E character and a decimal number. Two examples of valid entries are -732.103 and 7.12e4	float ∗
o	Octal Integer:	int ∗
s	String of characters. This reads subsequent characters until a white space is found (white space characters are considered to be blank, newline, and tab).	char ∗
u	Unsigned decimal integer.	unsigned int ∗

The debug console has its own printf/scanf/putchar/getchar functions which are defined in the header file.

```
int DbgConsole_Printf(const char *fmt_s, ...);
int DbgConsole_Putchar(int ch);
int DbgConsole_Scanf(char *fmt_ptr, ...);
int DbgConsole_Getchar(void);
```

This utility supports selecting toolchain’s printf/scanf or the MCUXpresso SDK printf/scanf.

```
#if SDK_DEBUGCONSOLE == DEBUGCONSOLE_DISABLE /* Disable debug console */
#define PRINTF
#define SCANF
#define PUTCHAR
#define GETCHAR
#elif SDK_DEBUGCONSOLE == DEBUGCONSOLE_REDIRECT_TO_SDK /* Select printf, scanf, putchar, getchar of SDK
```

```
      version. */
#define PRINTF DbgConsole_Printf
#define SCANF DbgConsole_Scanf
#define PUTCHAR DbgConsole_Putchar
#define GETCHAR DbgConsole_Getchar
#elif SDK_DEBUGCONSOLE == DEBUGCONSOLE_REDIRECT_TO_TOOLCHAIN /* Select printf, scanf, putchar, getchar of
      toolchain. */
#define PRINTF printf
#define SCANF scanf
#define PUTCHAR putchar
#define GETCHAR getchar
#endif /* SDK_DEBUGCONSOLE */
```

27.2.3 SDK_DEBUGCONSOLE and SDK_DEBUGCONSOLE_UART

There are two macros SDK_DEBUGCONSOLE and SDK_DEBUGCONSOLE_UART added to configure PRINTF and low level output perihperal.

- The macro SDK_DEBUGCONSOLE is used for forntend. Whether debug console redirect to toolchain or SDK or disabled, it decides which is the frontend of the debug console, Tool chain or SDK. The fucntion can be set by the macro SDK_DEBUGCONSOLE.
- The macro SDK_DEBUGCONSOLE_UART is used for backend. It is use to decide whether provide low level IO implementation to toolchain printf and scanf. For example, within MCU-Xpresso, if the macro SDK_DEBUGCONSOLE_UART is defined, __sys_write and __sys_readc will be used when __REDLIB__ is defined; _write and _read will be used in other cases.The macro does not specifically refer to the perihpheral "UART". It refers to the external perihperal UART. So if the macro SDK_DEBUGCONSOLE_UART is not defined when tool-chain printf is calling, the semihosting will be used.

The following the matrix show the effects of SDK_DEBUGCONSOLE and SDK_DEBUGCONSOLE_-UART on PRINTF and printf. The green mark is the default setting of the debug console.

SDK_DEBUGCONSOLE	SDK_DEBUGCONSOLE_UART	PRINTF	printf
DEBUGCONSOLE_-REDIRECT_TO_SDK	defined	UART	UART
DEBUGCONSOLE_-REDIRECT_TO_SDK	undefined	UART	semihost
DEBUGCONSOLE_-REDIRECT_TO_TO-OLCHAIN	defined	UART	UART
DEBUGCONSOLE_-REDIRECT_TO_TO-OLCHAIN	undefined	semihost	semihost
DEBUGCONSOLE_-DISABLE	defined	No ouput	UART
DEBUGCONSOLE_-DISABLE	undefined	No ouput	semihost

SDK_DEBUGCONSOLE	SDK_DEBUGCONSOLE_UART	PRINTF	printf

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

Macros

- #define DEBUGCONSOLE_REDIRECT_TO_TOOLCHAIN 0U
  Definition select redirect toolchain printf, scanf to uart or not.
- #define DEBUGCONSOLE_REDIRECT_TO_SDK 1U
  Select SDK version printf, scanf.
- #define DEBUGCONSOLE_DISABLE 2U
  Disable debugconsole function.
- #define SDK_DEBUGCONSOLE DEBUGCONSOLE_REDIRECT_TO_SDK
  Definition to select sdk or toolchain printf, scanf.
- #define PRINTF_FLOAT_ENABLE 0U
  Definition to printf the float number.
- #define SCANF_FLOAT_ENABLE 0U
  Definition to scanf the float number.
- #define PRINTF_ADVANCED_ENABLE 0U
  Definition to support advanced format specifier for printf.
- #define SCANF_ADVANCED_ENABLE 0U
  Definition to support advanced format specifier for scanf.
- #define PRINTF DbgConsole_Printf
  Definition to select redirect toolchain printf, scanf to uart or not.

Initialization

- status_t DbgConsole_Init (uint8_t instance, uint32_t baudRate, serial_port_type_t device, uint32_t clkSrcFreq)
  Initializes the peripheral used for debug messages.
- status_t DbgConsole_Deinit (void)
  De-initializes the peripheral used for debug messages.
- status_t DbgConsole_EnterLowpower (void)
  Prepares to enter low power consumption.
- status_t DbgConsole_ExitLowpower (void)
  Restores from low power consumption.
- int DbgConsole_Printf (const char ∗fmt_s,...)
  Writes formatted output to the standard output stream.
- int DbgConsole_Vprintf (const char ∗fmt_s, va_list formatStringArg)
  Writes formatted output to the standard output stream.
- int DbgConsole_Putchar (int ch)
  Writes a character to stdout.
- int DbgConsole_Scanf (char ∗fmt_s,...)
  Reads formatted data from the standard input stream.
- int DbgConsole_Getchar (void)
  Reads a character from standard input.

27.4 Macro Definition Documentation

27.4.1 #define DEBUGCONSOLE_REDIRECT_TO_TOOLCHAIN 0U

Select toolchain printf and scanf.

27.4.2 #define DEBUGCONSOLE_REDIRECT_TO_SDK 1U

27.4.3 #define DEBUGCONSOLE_DISABLE 2U

27.4.4 #define SDK_DEBUGCONSOLE DEBUGCONSOLE_REDIRECT_TO_SDK

27.4.5 #define PRINTF_FLOAT_ENABLE 0U

27.4.6 #define SCANF_FLOAT_ENABLE 0U

27.4.7 #define PRINTF_ADVANCED_ENABLE 0U

27.4.8 #define SCANF_ADVANCED_ENABLE 0U

27.4.9 #define PRINTF DbgConsole_Printf

if SDK_DEBUGCONSOLE defined to 0,it represents select toolchain printf, scanf. if SDK_DEBUGCO-NSOLE defined to 1,it represents select SDK version printf, scanf. if SDK_DEBUGCONSOLE defined to 2,it represents disable debugconsole function.

27.5 Function Documentation

27.5.1 status_t DbgConsole_Init (uint8_t *instance,* uint32_t *baudRate,* serial_port_type_t *device,* uint32_t *clkSrcFreq*)

Call this function to enable debug log messages to be output via the specified peripheral, frequency of peripheral source clock, and base address at the specified baud rate. After this function has returned, stdout and stdin are connected to the selected peripheral.

Parameters

instance	The instance of the module.If the device is kSerialPort_Uart, the instance is UART peripheral instance. The UART hardware peripheral type is determined by UAR-T adapter. For example, if the instance is 1, if the lpuart_adapter.c is added to the current project, the UART periheral is LPUART1. If the uart_adapter.c is added to the current project, the UART periheral is UART1.
baudRate	The desired baud rate in bits per second.
device	Low level device type for the debug console, can be one of the following. • kSerialPort_Uart.
clkSrcFreq	Frequency of peripheral source clock.

Returns

Indicates whether initialization was successful or not.

Return values

kStatus_Success	Execution successfully
kStatus_Fail	Execution failure

27.5.2 status_t DbgConsole_Deinit (void)

Call this function to disable debug log messages to be output via the specified peripheral base address and at the specified baud rate.

Returns

Indicates whether de-initialization was successful or not.

27.5.3 status_t DbgConsole_EnterLowpower (void)

This function is used to prepare to enter low power consumption.

Returns

Indicates whether de-initialization was successful or not.

27.5.4 status_t DbgConsole_ExitLowpower (void)

This function is used to restore from low power consumption.

Returns

Indicates whether de-initialization was successful or not.

27.5.5 int DbgConsole_Printf (const char ∗ *fmt_s,* ...)

Call this function to write a formatted output to the standard output stream.

Parameters

fmt_s	Format control string.

Returns

Returns the number of characters printed or a negative value if an error occurs.

27.5.6 int DbgConsole_Vprintf (const char ∗ *fmt_s,* va_list *formatStringArg*)

Call this function to write a formatted output to the standard output stream.

Parameters

fmt_s	Format control string.
formatString-Arg	Format arguments.

Returns

Returns the number of characters printed or a negative value if an error occurs.

27.5.7 int DbgConsole_Putchar (int *ch*)

Call this function to write a character to stdout.

Parameters

ch	Character to be written.

Returns

Returns the character written.

27.5.8 int DbgConsole_Scanf (char ∗ *fmt_s*, ...)

Call this function to read formatted data from the standard input stream.

Parameters

fmt_s	Format control string.

Returns

Returns the number of fields successfully converted and assigned.

27.5.9 int DbgConsole_Getchar (void)

Call this function to read a character from standard input.

Returns

Returns the character read.

27.6 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.

27.6.1 Guide Semihosting for IAR

NOTE: After the setting both "printf" and "scanf" are available for debugging.

Step 1: Setting up the environment

1. To set debugger options, choose Project>Options. In the Debugger category, click the Setup tab.
2. Select Run to main and click OK. This ensures that the debug session starts by running the main function.
3. The project is now ready to be built.

Step 2: Building the project

1. Compile and link the project by choosing Project>Make or F7.
2. Alternatively, click the Make button on the tool bar. The Make command compiles and links those files that have been modified.

Step 3: Starting semihosting

1. Choose "Semihosting_IAR" project -> "Options" -> "Debugger" -> "J-Link/J-Trace".
2. Choose tab "J-Link/J-Trace" -> "Connection" tab -> "SWD".
3. Choose tab "General Options" -> "Library Configurations", select Semihosted, select Via semihosting. Please Make sure the SDK_DEBUGCONSOLE_UART is not defined in project settings.
4. Start the project by choosing Project>Download and Debug.
5. Choose View>Terminal I/O to display the output from the I/O operations.

27.6.2 Guide Semihosting for Keil µVision

NOTE: Semihosting is not support by MDK-ARM, use the retargeting functionality of MDK-ARM instead.

27.6.3 Guide Semihosting for MCUXpresso IDE

Step 1: Setting up the environment

1. To set debugger options, choose Project>Properties. select the setting category.
2. Select Tool Settings, unfold MCU C Compile.
3. Select Preprocessor item.
4. Set SDK_DEBUGCONSOLE=0, if set SDK_DEBUGCONSOLE=1, the log will be redirect to the UART.

Step 2: Building the project

1. Compile and link the project.

Step 3: Starting semihosting

1. Download and debug the project.
2. When the project runs successfully, the result can be seen in the Console window.

Semihosting can also be selected through the "Quick settings" menu in the left bottom window, Quick settings->SDK Debug Console->Semihost console.

27.6.4 Guide Semihosting for ARMGCC

Step 1: Setting up the environment

1. Turn on "J-LINK GDB Server" -> Select suitable "Target device" -> "OK".
2. Turn on "PuTTY". Set up as follows.
   - "Host Name (or IP address)" : localhost
   - "Port" :2333
   - "Connection type" : Telet.
   - Click "Open".
3. Increase "Heap/Stack" for GCC to 0x2000:

Add to "CMakeLists.txt"

SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE} --defsym=__stack_size__=0x2000")

SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --defsym=__stack_size__=0x2000")

SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --defsym=__heap_size__=0x2000")

SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE} --defsym=__heap_size__=0x2000")

Step 2: Building the project

1. Change "CMakeLists.txt":
   Change "SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE} –specs=nano.specs")"
   to "SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE} –specs=rdimon.specs")"
   Replace paragraph
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -fno-common")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -ffunction-sections")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -fdata-sections")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -ffreestanding")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -fno-builtin")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -mthumb")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -mapcs")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -Xlinker")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --gc-sections")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -Xlinker")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -static")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -Xlinker")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -z")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -Xlinker")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} muldefs")
   To
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --specs=rdimon.specs ")
   Remove
   target_link_libraries(semihosting_ARMGCC.elf debug nosys)
2. Run "build_debug.bat" to build project

Step 3: Starting semihosting

1. Download the image and set as follows.

```
cd D:\mcu-sdk-2.0-origin\boards\twrk64f120m\driver_examples\semihosting\armgcc\debug
d:
C:\PROGRA~2\GNUTOO~1\4BD65~1.920\bin\arm-none-eabi-gdb.exe
target remote localhost:2331
monitor reset
monitor semihosting enable
monitor semihosting thumbSWI 0xAB
monitor semihosting IOClient 1
monitor flash device = MK64FN1M0xxx12
load semihosting_ARMGCC.elf
monitor reg pc = (0x00000004)
monitor reg sp = (0x00000000)
continue
```

2. After the setting, press "enter". The PuTTY window now shows the printf() output.

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.

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, Freescale, the Freescale logo, Kinetis, Processor Expert, and Tower are trademarks of NXP B.V. All other product or service names are the property of their respective owners. Arm, Cortex, Keil, Mbed, Mbed Enabled, and Vision 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.

© 2021 NXP B.V.