FreeRTOS BSP i.MX 7Dual API Reference Manual

Freescale Semiconductor, Inc.

Document Number: FRTOS7DAPIRM
Rev. 0
August 2015

Contents

Contents

Contents

Contents

Section Number | Title | Page Number

Contents

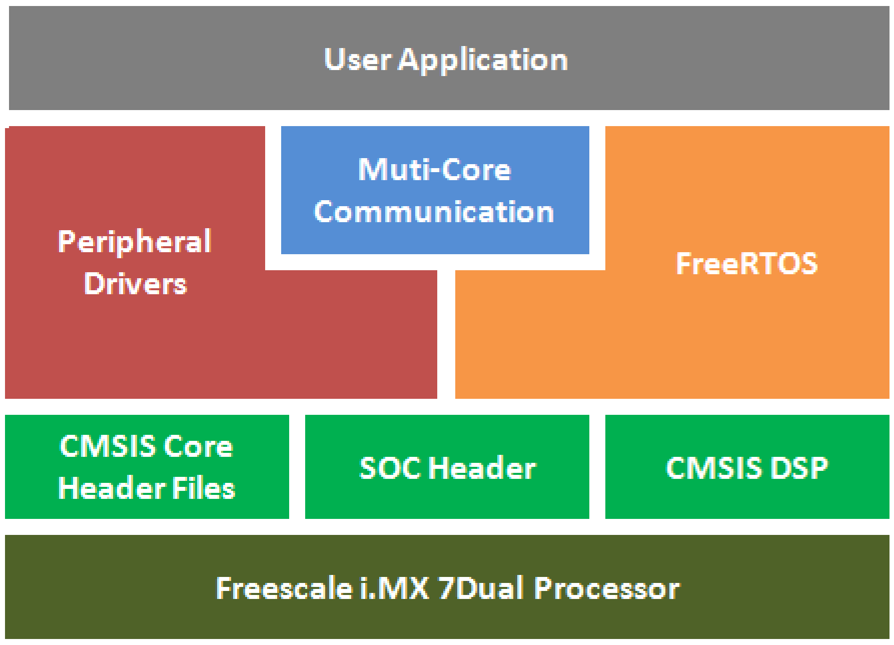
Section Number | Title | Page Number

Chapter 1
Introduction

The FreeRTOS BSP for i.MX 7Dual is a suite of robust peripheral drivers, FreeRTOS support, and multi-core communication mechanism designed to simplify and accelerate application development on Freescale i.MX 7Dual Processor.

Included in the FreeRTOS BSP for i.MX 7Dual is a full source code under a permissive open-source license for all demos, examples, middleware, and peripheral driver software.

The FreeRTOS BSP for i.MX 7Dual consists of the following runtime software components written in C:



- ARM® CMSIS Core, DSP standard libraries, and CMSIS-compliant device header files which provide direct access to the peripheral registers and bits.
- A set of peripheral drivers that provides a simple, stateless driver with an API encapsulating register access of the peripheral.
- A multicore communication mechanism (RPMsg) to decrease the developing difficulty of multicore communication.
- A FreeRTOS operating system to provide an event triggered preemptive scheduling RTOS.

The FreeRTOS BSP for i.MX 7Dual comes complete with software examples demonstrating the usage of the peripheral drivers, middleware, and FreeRTOS operating system. All examples are provided with projects for the following toolchains:

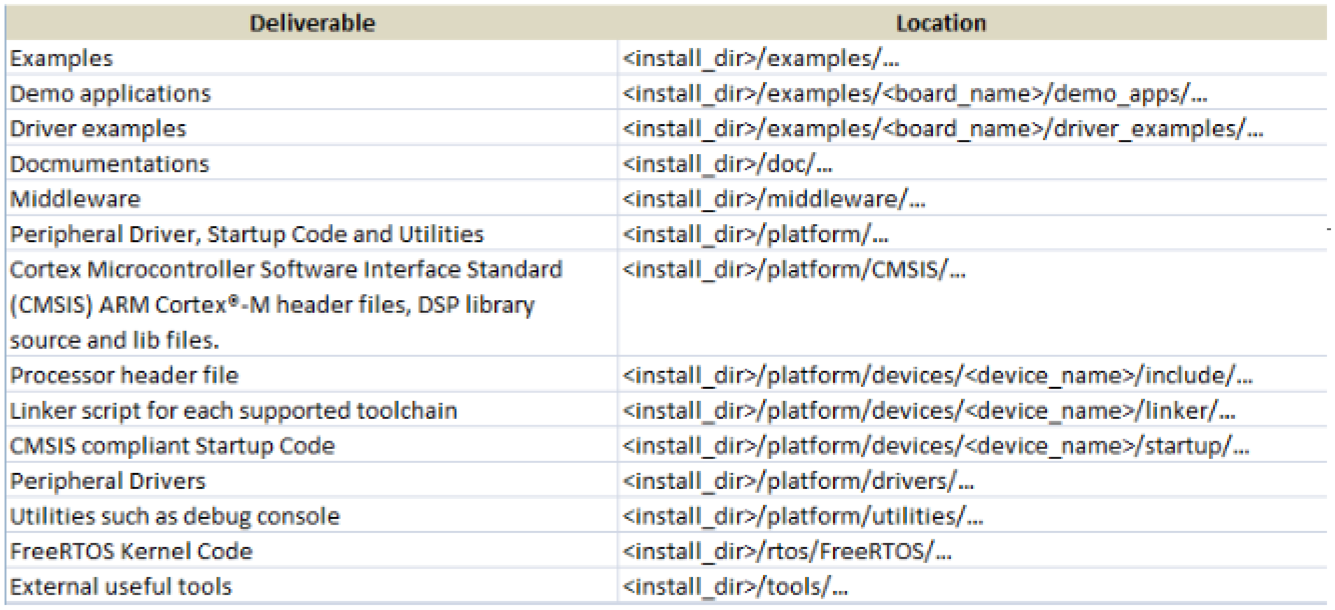
- DS-5
- GNU toolchain for ARM® Cortex® -M with Cmake build system

- IAR Embedded Workbench

The configurable items for each driver, at all levels, are encapsulated into C language data structures. Peripheral driver, board and FreeRTOS configuration is not fixed and can be changed according to the application.

The example applications demonstrate how to configure the drivers by passing configuration data to the APIs.

The organization of files in the FreeRTOS BSP for i.MX 7Dual release package is focused on ease-of-use. The FreeRTOS BSP for i.MX 7Dual folder hierarchy is organized at the top level with these folders:



Deliverable	Location
Examples	<install_dir>/examples/...
Demo applications	<install_dir>/examples/<board_name>/demo_apps/...
Driver examples	<install_dir>/examples/<board_name>/driver_examples/...
Docmumentations	<install_dir>/doc/...
Middleware	<install_dir>/middleware/...
Peripheral Driver, Startup Code and Utilities	<install_dir>/platform/...
Cortex Microcontroller Software Interface Standard (CMSIS) ARM Cortex®-M header files, DSP library source and lib files.	<install_dir>/platform/CMSIS/...
Processor header file	<install_dir>/platform/devices/<device_name>/include/...
Linker script for each supported toolchain	<install_dir>/platform/devices/<device_name>/linker/...
CMSIS compliant Startup Code	<install_dir>/platform/devices/<device_name>/startup/...
Peripheral Drivers	<install_dir>/platform/drivers/...
Utilities such as debug console	<install_dir>/platform/utilities/...
FreeRTOS Kernel Code	<install_dir>/rtos/FreeRTOS/...
External useful tools	<install_dir>/tools/...

The other sections of the document describes the API functions for Peripheral drivers.

Chapter 2
Architectural Overview

This chapter provides the architectural overview for the FreeRTOS BSP for i.MX 7Dual. It describes each layer within the architecture and its associated components.

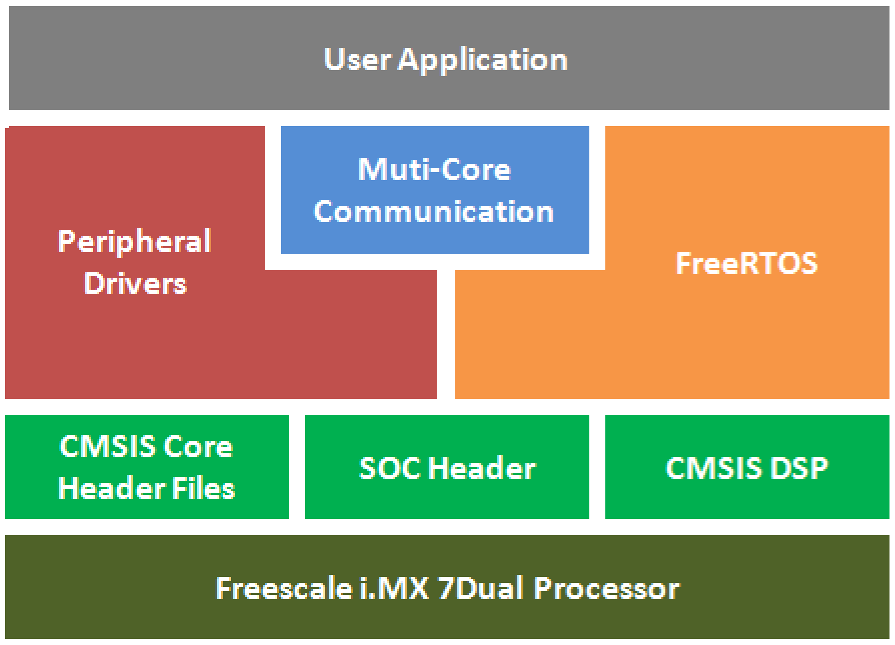
Overview

The FreeRTOS BSP for i.MX 7Dual architecture consists of six key components listed below.

1. The ARM Cortex Microcontroller Software Interface Standard (CMSIS) core-compliant device specific header files, SoC Header, and CMSIS DSP libraries.
2. Peripheral Drivers
3. Real-time Operating System (RTOS) —— FreeRTOS OS
4. Board-specific configuration
5. Multicore communication mechanism integrate with FreeRTOS BSP for i.MX 7Dual
6. Applications based on the FreeRTOS BSP architecture

This image shows how each component stacks up.

The FreeRTOS BSP consists of these runtime software components written in C:



i.MX Processor header files

The FreeRTOS BSP contains CMSIS-compliant device-specific header files which provide direct access to the i.MX Processor peripheral registers. Each supported i.MX device in FreeRTOS BSP 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 an access to the peripheral registers through pointers and predefined masks.

Along with the SoC header files, the FreeRTOS BSP also includes common CMSIS header files for the ARM Cortex-M core and DSP library from the latest CMSIS release. The CMSIS DSP library source code is also included for reference. These files and the above mentioned header files can all be found in the FreeRTOS BSP platform/CMSIS directory.

Peripheral Drivers

The FreeRTOS BSP Peripheral Drivers consists of low-level drivers for the i.MX Series Processor on-chip peripherals. The main goal of this part is to abstract the hardware peripheral register accesses into a set of stateless basic functional operations. The Peripheral Driver itself can be used to build application-specific logic or as building blocks for use case-driven high-level Peripheral Drivers. It primarily focuses on the functional control, configuration, and realization of basic peripheral operations. The Peripheral Driver hides register access details and various processor peripheral instantiation differences so that, either an application or high-level Peripheral Drivers, can be abstracted from the low-level hardware details. Therefore, the hardware peripheral must be accessed through a Peripheral Driver.

The Peripheral Drivers cover the most useful basic APIs for embedded system developers and provide an easy-to-use initialization functions. The initialization functions initialization data structure consist of all the necessary parameters to bring Peripherals into use. For example, the UART Driver initialization data structure includes the baud Rate, data bits number, number of stop bits, parity error check mode and data transfer direction. Essentially, the Peripheral Driver functional boundary is limited by the peripheral itself. There is one Peripheral Driver for each peripheral and the Peripheral Driver only accesses the features available within the peripheral. In addition, the Peripheral Driver does not define interrupt service routine entries or support interrupt handling. These tasks must be handled by a high-level Peripheral Driver or by the user application.

The Peripheral Drivers can be found in the platform/drivers directory.

Design Guidelines

This section summarizes the design guidelines that were used to develop the Peripheral Drivers. It is meant for information purposes and provides more details on the make-up of the Peripheral drivers. As previously stated, the main goal of the Peripheral Driver is to abstract the hardware details and provide a set of easy-to-use low-level drivers. The Peripheral Driver itself can be used directly by the user application or as building blocks of high-level transaction drivers. The Peripheral Driver is mainly focused on individual functional primitives and makes no assumption of use cases. The Peripheral Driver APIs follow a naming convention that is consistent from one peripheral to the next. This is a summary of the design guidelines used when developing the Peripheral Driver drivers:

- There is a dedicated Peripheral Driver for each individual peripheral.
- Each Peripheral Driver has an initialization function to put the peripheral into a ready to use state.
- Each Peripheral Driver has an enable and disable function to enable or disable the peripheral module.
- The Peripheral Driver does not have an internal operation context and should not dynamically allocate memory.
- The Peripheral Driver must remain stateless.

- The Peripheral Driver does not depend on any other software entities or invoke any functions from other peripherals.

The CMSIS startup code contains the vector table with the ISR entry names which are defined as weak references. The Peripheral Driver ISR entry names match the names defined in the vector table such that these newly-defined ISR entries in the driver replace the weak references defined in the vector table. There is no dependency on the location of the vector table. However, if the vector table is re-located, the ISR entry names should remain consistent.

To use the Peripheral Driver to directly build an interrupt-driven application or a high-level driver, define the ISR entries to service needed interrupts. The ISR entry names have to match the names of the ISR entry names provided in the CMSIS startup code vector table.

Demo Applications

The Demo Applications provided in the FreeRTOS BSP provide examples, which show how to build user applications using the FreeRTOS BSP framework. The Demo Applications can be found in the FreeRTOS BSP top-level demo directory. The FreeRTOS BSP includes two types of demo applications:

- Demo Applications that demonstrate the usage of the Peripheral Drivers.
- Demo Applications that provide a reference design based on the features available on the target i.-MX Processor and its evaluation boards. This demo is targeted to highlight a certain feature of the SoC for its intended usage, and/or to provide turnkey references using the FreeRTOS BSP peripheral driver library with other integrated software components, such as RPMsg .

Board Configuration

The FreeRTOS BSP drivers make no assumption on board-specific configurations nor do the drivers configure pin muxing, which are part of the board-specific configuration. The FreeRTOS BSP provides board-configuration files that configure pin muxing for applications, clock gating for on-chip peripherals, Resource Domain Controller setting, FreeRTOS feature customize setting and functions that can be called before driver initialization:

- Pin Muxing configuration is used to set the IOMUX connection between dedicated pins and peripherals.
- Clock settings of the board configuration vary from demo to demo. The clock for peripheral is off by default and should be opened in the demo/application's hardware_init file.
- Resource Domain Controller setting varies from demo to demo. The peripheral used by ARM® Cortex® -M4 core can be set as monopolized or shared with ARM® Cortex® -A7 Core. The RDC configuration is also located in the hardware_init file of certain demo/application.
- FreeRTOS feature customize setting include setting to maximize kernel performance and functionality or to minimize code size.
- Board Configuration includes useful functions, such as the debug console initialization, and so on.

These board-configuration files can be found in the examples/<board_name> directory.

FreeRTOS Operating System

The FreeRTOS BSP drivers are designed to work with or without an RTOS. The FreeRTOS OS provides a common set of service routines for drivers, integrated software solutions, and upper-level applications.

Middleware integration

FreeRTOS BSP also provides a foundation for software stacks and other middleware. The FreeRTOS BSP integrates other third party enablement software stacks, such as RPMsg and other middleware, to offer a complete, easy-to-use, software development kit for the i.MX Processor users.

Memory Division

This section discuss the FreeRTOS BSP demo/example build target location in the memory map and how to build an application/demo for other storage devices.

The demos/examples in FreeRTOS BSP are built for on-chip Tightly Coupled Memory (TCM) in the A-RM Cortex-M4 core. Running at TCM gives the application the best performance. However, the TCML used to store application firmware has a 32 KB capacity limit. To overcome this drawback, the user should move the application to a different position in the memory map.

To build a demo/example firmware for other storage devices, the user doesn't need to change the source code of the demo/example. Only the linker script of the demo/example should be changed. The linker script for i.MX processor device is located in the <install_dir>/platform/devices/<device_-name>/linker/<toolchain>. When making modifications, note the following:

- The modified memory space is a legal space to boot the ARM Cortex-M4 Core.
- The modified memory space is not occupied by the ARM Cortex-A7 Core.

Chapter 3
Analog-to-Digital Convert (ADC)

3.1 Overview

The FreeRTOS BSP provides a driver for the Analog-to-Digital Converter (ADC) block of i.MX devices.

Modules

- ADC driver

3.2 ADC driver

3.2.1 Overview

This section describes the programming interface of the ADC driver (platform/drivers/inc/adc_imx7d.h). The Analog-to-Digital Converter (ADC) peripheral driver configures the ADC (12-bit Analog-to-Digital Converter). It handles initialization and configuration of a 12-bit ADC module.

3.2.2 ADC Driver model building

ADC driver has three parts:

- Basic Converter - This part handles the mechanism that converts the external analog voltage to a digital value. API functions configure the converter.
- Channel Mux - Multiple channels share the converter in each ADC instance because of the time division multiplexing. However, the converter can only handle one channel at a time. To get the value of an indicated channel, the channel mux should be set to the connection between an indicated pad and the converter's input. The conversion value during this period is for the channel only. API functions configure the channel.
- Advance Feature Group - The advanced feature group covers optional features for applications. These features includes some that are already implemented by hardware, such as the hardware average, hardware compare, different power, and speed mode. APIs configure the advanced features. Although these features are optional, they are recommended to ensure that the ADC performs better, especially for calibration.

3.2.3 ADC Initialization

To initialize the ADC driver, prepare a configuration structure and populate it with an available configuration. API functions are designed for typical use cases and facilitate populating the structure.

1. Use the ADC_Init() function to set ADC module sample rate and enablement of the level-shifter.

   ```
   void ADC_Init(ADC_Type* base, adc_init_config_t* initConfig)
   ```

2. Use the ADC_LogicChInit() function to initialize ADC Logic channel.It can set input channel,convert rate and hardware averge number.

   ```
   void ADC_LogicChInit(ADC_Type* base, uint8_t logicCh,
         adc_logic_ch_init_config_t* chInitConfig)
   ```

3. Choose the ADC operation mode with ADC conversion control functions or ADC comparer control functions.

   ```
   void ADC_SetConvertCmd(ADC_Type* base, uint8_t logicCh, bool enable)
   void ADC_TriggerSingleConvert(ADC_Type* base, uint8_t logicCh)
   void ADC_SetCmpMode(ADC_Type* base, uint8_t logicCh, uint8_t cmpMode)
   ```

4. Finally, set an interrupt mode with interrupt and flag control functions or DMA and FIFO control functions.

```
void ADC_SetIntCmd(ADC_Type* base, uint32_t intSource, bool enable)
void ADC_SetIntSigCmd(ADC_Type* base, uint32_t intSignal, bool enable)
void ADC_SetDmaCmd(ADC_Type* base, bool enable)
```

5. For a low-power performance, use ADC low-power control functions.

```
void ADC_SetClockDownCmd(ADC_Type* base, bool clockDown)
void ADC_SetPowerDownCmd(ADC_Type* base, bool powerDown)
```

3.2.4 ADC Get Result

Use ADC_GetConvertResult() to get results.

```
uint16_t ADC_GetConvertResult(ADC_Type* base, uint8_t logicCh)
```

Data Structures

- struct adc_init_config_t
  ADC module initialize structure. More...
- struct adc_logic_ch_init_config_t
  ADC logic channel initialize structure. More...

Enumerations

- enum _adc_logic_ch_selection {
  adcLogicChA = 0x0,
  adcLogicChB = 0x1,
  adcLogicChC = 0x2,
  adcLogicChD = 0x3,
  adcLogicChSW = 0x4 }
  ADC logic channel selection enumeration.
- enum _adc_average_number {
  adcAvgNum4 = 0x0,
  adcAvgNum8 = 0x1,
  adcAvgNum16 = 0x2,
  adcAvgNum32 = 0x3 }
  ADC hardware average number enumeration.
- enum _adc_compare_mode {
  adcCmpModeDisable = 0x0,
  adcCmpModeGreaterThanLow = 0x1,
  adcCmpModeLessThanLow = 0x2,
  adcCmpModeInInterval = 0x3,
  adcCmpModeGreaterThanHigh = 0x5,
  adcCmpModeLessThanHigh = 0x6,
  adcCmpModeOutOffInterval = 0x7 }
  ADC build-in comparer work mode configuration enumeration.

- enum _adc_interrupt
  This enumeration contains the settings for all of the ADC interrupt configurations.
- enum _adc_status_flag
  Flag for ADC interrupt/DMA status check or polling status.

ADC Module Initialization and Configuration functions.

- void ADC_Init (ADC_Type ∗base, adc_init_config_t ∗initConfig)
  Initialize ADC to reset state and initialize with initialize structure.
- void ADC_Deinit (ADC_Type ∗base)
  This function reset ADC module register content to its default value.
- static void ADC_LevelShifterEnable (ADC_Type ∗base)
  This function Enable ADC module build-in Level Shifter.
- static void ADC_LevelShifterDisable (ADC_Type ∗base)
  This function Disable ADC module build-in Level Shifter to save power.
- void ADC_SetSampleRate (ADC_Type ∗base, uint32_t sampleRate)
  This function is used to set ADC module sample rate.

ADC Low power control functions.

- void ADC_SetClockDownCmd (ADC_Type ∗base, bool clockDown)
  This function is used to stop all digital part power.
- void ADC_SetPowerDownCmd (ADC_Type ∗base, bool powerDown)
  This function is used to power down ADC analogue core.

ADC Convert Channel Initialization and Configuration functions.

- void ADC_LogicChInit (ADC_Type ∗base, uint8_t logicCh, adc_logic_ch_init_config_t ∗chInitConfig)
  Initialize ADC Logic channel with initialize structure.
- void ADC_LogicChDeinit (ADC_Type ∗base, uint8_t logicCh)
  Reset target ADC logic channel registers to default value.
- void ADC_SelectInputCh (ADC_Type ∗base, uint8_t logicCh, uint8_t inputCh)
  Select input channel for target logic channel.
- void ADC_SetConvertRate (ADC_Type ∗base, uint8_t logicCh, uint32_t convertRate)
  Set ADC conversion rate of target logic channel.
- void ADC_SetAverageCmd (ADC_Type ∗base, uint8_t logicCh, bool enable)
  Set work state of hardware average feature of target logic channel.
- void ADC_SetAverageNum (ADC_Type ∗base, uint8_t logicCh, uint8_t avgNum)
  Set hardware average number of target logic channel.

ADC Conversion Control functions.

- void ADC_SetConvertCmd (ADC_Type ∗base, uint8_t logicCh, bool enable)
  Set continuous convert work mode of target logic channel.

- void ADC_TriggerSingleConvert (ADC_Type ∗base, uint8_t logicCh)
  Trigger single time convert on target logic channel.
- uint16_t ADC_GetConvertResult (ADC_Type ∗base, uint8_t logicCh)
  Get 12-bit length right aligned convert result.

ADC Comparer Control functions.

- void ADC_SetCmpMode (ADC_Type ∗base, uint8_t logicCh, uint8_t cmpMode)
  Set the work mode of ADC module build-in comparer on target logic channel.
- void ADC_SetCmpHighThres (ADC_Type ∗base, uint8_t logicCh, uint16_t threshold)
  Set ADC module build-in comparer high threshold on target logic channel.
- void ADC_SetCmpLowThres (ADC_Type ∗base, uint8_t logicCh, uint16_t threshold)
  Set ADC module build-in comparer low threshold on target logic channel.
- void ADC_SetAutoDisableCmd (ADC_Type ∗base, uint8_t logicCh, bool enable)
  Set the working mode of ADC module auto disable feature on target logic channel.

Interrupt and Flag control functions.

- void ADC_SetIntCmd (ADC_Type ∗base, uint32_t intSource, bool enable)
  Enables or disables ADC interrupt requests.
- void ADC_SetIntSigCmd (ADC_Type ∗base, uint32_t intSignal, bool enable)
  Enables or disables ADC interrupt flag when interrupt condition met.
- static uint32_t ADC_GetStatusFlag (ADC_Type ∗base, uint32_t flags)
  Gets the ADC status flag state.
- static void ADC_ClearStatusFlag (ADC_Type ∗base, uint32_t flags)
  Clear one or more ADC status flag state.

DMA & FIFO control functions.

- void ADC_SetDmaReset (ADC_Type ∗base, bool active)
  Set the reset state of ADC internal DMA part.
- void ADC_SetDmaCmd (ADC_Type ∗base, bool enable)
  Set the work mode of ADC DMA part.
- void ADC_SetDmaFifoCmd (ADC_Type ∗base, bool enable)
  Set the work mode of ADC DMA FIFO part.
- static void ADC_SetDmaCh (ADC_Type ∗base, uint32_t logicCh)
  Select the logic channel that uses the DMA transfer.
- static void ADC_SetDmaWatermark (ADC_Type ∗base, uint32_t watermark)
  Set the DMA request trigger watermark.
- static uint32_t ADC_GetFifoData (ADC_Type ∗base)
  Get the convert result from DMA FIFO.
- static bool ADC_IsFifoFull (ADC_Type ∗base)
  Get the DMA FIFO full status.
- static bool ADC_IsFifoEmpty (ADC_Type ∗base)
  Get the DMA FIFO empty status.
- static uint8_t ADC_GetFifoEntries (ADC_Type ∗base)
  Get the entries number in DMA FIFO.

3.2.5 Data Structure Documentation

3.2.5.1 struct adc_init_config_t

Data Fields

- uint32_t sampleRate
  The desired ADC sample rate.
- bool levelShifterEnable
  The level shifter module configuration(Enable to power on ADC module).

3.2.5.1.0.1 Field Documentation

3.2.5.1.0.1.1 uint32_t adc_init_config_t::sampleRate

3.2.5.1.0.1.2 bool adc_init_config_t::levelShifterEnable

3.2.5.2 struct adc_logic_ch_init_config_t

Data Fields

- uint8_t inputChannel
  The logic channel to be set.
- bool coutinuousEnable
  Continuous sample mode enable configuration.
- uint32_t convertRate
  The continuous rate when continuous sample enabled.
- bool averageEnable
  Hardware average enable configuration.
- uint8_t averageNumber
  The average number for hardware average function.

3.2.5.2.0.2 Field Documentation

3.2.5.2.0.2.1 uint8_t adc_logic_ch_init_config_t::inputChannel

3.2.5.2.0.2.2 bool adc_logic_ch_init_config_t::coutinuousEnable

3.2.5.2.0.2.3 uint32_t adc_logic_ch_init_config_t::convertRate

3.2.5.2.0.2.4 bool adc_logic_ch_init_config_t::averageEnable

3.2.5.2.0.2.5 uint8_t adc_logic_ch_init_config_t::averageNumber

3.2.6 Enumeration Type Documentation

3.2.6.1 enum _adc_logic_ch_selection

Enumerator

adcLogicChA ADC Logic Channel A.
adcLogicChB ADC Logic Channel B.
adcLogicChC ADC Logic Channel C.
adcLogicChD ADC Logic Channel D.
adcLogicChSW ADC Logic Channel Software.

3.2.6.2 enum _adc_average_number

Enumerator

adcAvgNum4 ADC Hardware Average Number is set to 4.
adcAvgNum8 ADC Hardware Average Number is set to 8.
adcAvgNum16 ADC Hardware Average Number is set to 16.
adcAvgNum32 ADC Hardware Average Number is set to 32.

3.2.6.3 enum _adc_compare_mode

Enumerator

adcCmpModeDisable ADC build-in comparator is disabled.
adcCmpModeGreaterThanLow ADC build-in comparator is triggered when sample value greater than low threshold.
adcCmpModeLessThanLow ADC build-in comparator is triggered when sample value less than low threshold.
adcCmpModeInInterval ADC build-in comparator is triggered when sample value in interval between low and high threshold.
adcCmpModeGreaterThanHigh ADC build-in comparator is triggered when sample value greater than high threshold.

adcCmpModeLessThanHigh ADC build-in comparator is triggered when sample value less than high threshold.

adcCmpModeOutOffInterval ADC build-in comparator is triggered when sample value out of interval between low and high threshold.

3.2.7 Function Documentation

3.2.7.1 void ADC_Init (ADC_Type ∗ *base*, adc_init_config_t ∗ *initConfig*)

Parameters

base	ADC base pointer.
initConfig	ADC initialize structure.

3.2.7.2 void ADC_Deinit (ADC_Type ∗ *base*)

Parameters

base	ADC base pointer.

3.2.7.3 static void ADC_LevelShifterEnable (ADC_Type ∗ *base*) [inline], [static]

```
For iMX7D, Level Shifter should always be enabled.
User can disable Level Shifter to save power.
```

Parameters

base	ADC base pointer.

3.2.7.4 static void ADC_LevelShifterDisable (ADC_Type ∗ *base*) [inline], [static]

Parameters

base	ADC base pointer.

3.2.7.5 void ADC_SetSampleRate (ADC_Type ∗ *base,* uint32_t *sampleRate*)

Parameters

base	ADC base pointer.
sampleRate	Desired ADC sample rate.

3.2.7.6 void ADC_SetClockDownCmd (ADC_Type ∗ *base,* bool *clockDown*)

Parameters

base	ADC base pointer.
clockDown	- true: Clock down. • false: Clock running.

3.2.7.7 void ADC_SetPowerDownCmd (ADC_Type ∗ *base,* bool *powerDown*)

```
Before entering into stop-mode, power down ADC analogue core first.
```

Parameters

base	ADC base pointer.
powerDown	- true: Power down the ADC analogue core. • false: Do not power down the ADC analogue core.

3.2.7.8 void ADC_LogicChInit (ADC_Type ∗ *base,* uint8_t *logicCh,* adc_logic_ch_init_config_t ∗ *chInitConfig*)

Parameters

base	ADC base pointer.
logicCh	ADC module logic channel selection (refer to _adc_logic_ch_selection enumeration).
chInitConfig	ADC logic channel initialize structure.

3.2.7.9 void ADC_LogicChDeinit (ADC_Type ∗ *base,* uint8_t *logicCh*)

Parameters

base	ADC base pointer.
logicCh	ADC module logic channel selection (refer to _adc_logic_ch_selection enumeration).

3.2.7.10 void ADC_SelectInputCh (ADC_Type ∗ *base,* uint8_t *logicCh,* uint8_t *inputCh*)

Parameters

base	ADC base pointer.
logicCh	ADC module logic channel selection (refer to _adc_logic_ch_selection enumeration).
inputCh	Input channel selection for target logic channel(vary from 0 to 15).

3.2.7.11 void ADC_SetConvertRate (ADC_Type ∗ *base,* uint8_t *logicCh,* uint32_t *convertRate*)

Parameters

base	ADC base pointer.
logicCh	ADC module logic channel selection(refer to _adc_logic_ch_selection enumeration).
convertRate	ADC conversion rate in Hz.

3.2.7.12 void ADC_SetAverageCmd (ADC_Type ∗ *base,* uint8_t *logicCh,* bool *enable*)

Parameters

base	ADC base pointer.
logicCh	ADC module logic channel selection (refer to _adc_logic_ch_selection enumeration).
enable	- true: Enable hardware average. • false: Disable hardware average.

3.2.7.13 void ADC_SetAverageNum (ADC_Type ∗ *base,* uint8_t *logicCh,* uint8_t *avgNum*)

Parameters

base	ADC base pointer.
logicCh	ADC module logic channel selection (refer to _adc_logic_ch_selection enumeration).
avgNum	hardware average number(should select from _adc_average_number enumeration).

3.2.7.14 void ADC_SetConvertCmd (ADC_Type ∗ *base,* uint8_t *logicCh,* bool *enable*)

Parameters

base	ADC base pointer.
logicCh	ADC module logic channel selection (refer to _adc_logic_ch_selection enumeration).
enable	- true: Enable continuous convert. • false: Disable continuous convert.

3.2.7.15 void ADC_TriggerSingleConvert (ADC_Type ∗ *base,* uint8_t *logicCh*)

Parameters

base	ADC base pointer.
logicCh	ADC module logic channel selection (refer to _adc_logic_ch_selection enumeration).

3.2.7.16 uint16_t ADC_GetConvertResult (ADC_Type ∗ *base,* uint8_t *logicCh*)

Parameters

base	ADC base pointer.
logicCh	ADC module logic channel selection(refer to _adc_logic_ch_selection enumeration).

Returns

convert result on target logic channel.

3.2.7.17 void ADC_SetCmpMode (ADC_Type ∗ *base*, uint8_t *logicCh*, uint8_t *cmpMode*)

Parameters

base	ADC base pointer.
logicCh	ADC module logic channel selection (refer to _adc_logic_ch_selection enumeration).
cmpMode	Comparer work mode selected from _adc_compare_mode enumeration.

3.2.7.18 void ADC_SetCmpHighThres (ADC_Type ∗ *base*, uint8_t *logicCh*, uint16_t *threshold*)

Parameters

base	ADC base pointer.
logicCh	ADC module logic channel selection (refer to _adc_logic_ch_selection enumeration).
threshold	Comparer threshold in 12-bit unsigned int formate.

3.2.7.19 void ADC_SetCmpLowThres (ADC_Type ∗ *base*, uint8_t *logicCh*, uint16_t *threshold*)

Parameters

base	ADC base pointer.

logicCh	ADC module logic channel selection (refer to _adc_logic_ch_selection enumeration).
threshold	Comparer threshold in 12-bit unsigned int formate.

3.2.7.20 void ADC_SetAutoDisableCmd (ADC_Type ∗ *base,* uint8_t *logicCh,* bool *enable*)

```
This feature can disable continuous conversion when CMP condition matched.
```

Parameters

base	ADC base pointer.
logicCh	ADC module logic channel selection (refer to _adc_logic_ch_selection enumeration).
enable	- true: Enable Auto Disable feature. • false: Disable Auto Disable feature.

3.2.7.21 void ADC_SetIntCmd (ADC_Type ∗ *base,* uint32_t *intSource,* bool *enable*)

Parameters

base	ADC base pointer.
intSource	ADC interrupt sources to configuration.
enable	Pass true to enable interrupt, false to disable.

3.2.7.22 void ADC_SetIntSigCmd (ADC_Type ∗ *base,* uint32_t *intSignal,* bool *enable*)

Parameters

base	ADC base pointer.
intSignal	ADC interrupt signals to configuration.
intSignal	Should be select from _adc_interrupt enumeration.

3.2.7.23 static uint32_t ADC_GetStatusFlag (ADC_Type ∗ *base,* uint32_t *flags*) `[inline]`, `[static]`

Parameters

base	ADC base pointer.
flags	ADC status flag mask defined in _adc_status_flag enumeration.

Returns

ADC status, each bit represents one status flag

3.2.7.24 static void ADC_ClearStatusFlag (ADC_Type ∗ *base*, uint32_t *flags*) [inline], [static]

Parameters

base	ADC base pointer.
flags	ADC status flag mask defined in _adc_status_flag enumeration.

3.2.7.25 void ADC_SetDmaReset (ADC_Type ∗ *base*, bool *active*)

Parameters

base	ADC base pointer.
active	- true :Reset the DMA and DMA FIFO return to its reset value. • false :de-active DMA reset.

3.2.7.26 void ADC_SetDmaCmd (ADC_Type ∗ *base*, bool *enable*)

Parameters

base	ADC base pointer.
enable	- true :Enable DMA, the data in DMA FIFO should move by SDMA. • false :Disable DMA, the data in DMA FIFO can only move by CPU.

3.2.7.27 void ADC_SetDmaFifoCmd (ADC_Type ∗ *base*, bool *enable*)

Parameters

base	ADC base pointer.
enable	- true :Enable DMA FIFO. • false :Disable DMA FIFO.

3.2.7.28 static void ADC_SetDmaCh (ADC_Type ∗ *base,* uint32_t *logicCh*) `[inline]`, `[static]`

Parameters

base	ADC base pointer.
logicCh	ADC module logic channel selection (refer to _adc_logic_ch_selection enumeration).

3.2.7.29 static void ADC_SetDmaWatermark (ADC_Type ∗ *base,* uint32_t *watermark*) `[inline]`, `[static]`

Parameters

base	ADC base pointer.
watermark	DMA request trigger watermark.

3.2.7.30 static uint32_t ADC_GetFifoData (ADC_Type ∗ *base*) `[inline]`, `[static]`

```
Data position:
    DMA_FIFO_DATA1(27~16bits)
    DMA_FIFO_DATA0(11~0bits)
```

Parameters

base	ADC base pointer.

Returns

Get 2 ADC transfer result from DMA FIFO.

3.2.7.31 static bool ADC_IsFifoFull (ADC_Type ∗ *base*) `[inline]`, `[static]`

Parameters

base	ADC base pointer.

Returns

- true: DMA FIFO full
  - false: DMA FIFO not full

3.2.7.32 static bool ADC_IsFifoEmpty (ADC_Type ∗ *base*) [inline], [static]

Parameters

base	ADC base pointer.

Returns

- true: DMA FIFO empty
  - false: DMA FIFO not empty

3.2.7.33 static uint8_t ADC_GetFifoEntries (ADC_Type ∗ *base*) [inline], [static]

Parameters

base	ADC base pointer.

Returns

The numbers of data in DMA FIFO.

Chapter 4
Clock Control Module (CCM)

4.1 Overview

The FreeRTOS BSP provides a driver for the Clock Control Module (CCM) block of i.MX devices.

Modules

- CCM Analog driver
- CCM driver

4.2 CCM Analog driver

4.2.1 Overview

The chapter describes the programming interface of the CCM Analog driver (platform/drivers/inc/ccm_-analog_imx7d.h). The Clock Control Module (CCM) Analog part provides the PLL and PFD control. The CCM Analog driver provides a set of APIs to access the control registers, including these services:

- PLL power, gate, lock status, and output frequency
- PFD gate, stable, fraction, and output frequency

4.2.2 PLL power, gate, lock status and output frequency

PLL uses OSC as an external reference clock. To use CCM PLL, ensure that the reference clock is set correctly.

PLL can be powered up/down by the POWERDOWN bit in the PLL register. If no peripheral is running with the clock derived from this PLL, the PLL can be powered down to save power. Use the CCM_ANALOG_PowerUpPll() and CCM_ANALOG_PowerDownPll() functions for this purpose.

PLL can be bypassed and peripherals can use PLL as a clock source to get the OSC frequency in bypassed mode. This is a legacy method for i.MX series. However it's not recommended for the i.MX 7Dual because all peripherals can directly select the OSC as a clock source and it doesn't make sense to force the PLL bypass. Use the CCM_ANALOG_SetPllBypass() and CCM_ANALOG_IsPllBypassed() functions to set and get the status of the PLL bypass mode.

After power up, the PLL clock is still not available for use. The PLL clock functions CCM_ANALOG_-EnablePllClock() and CCM_ANALOG_DisablePllClock() can be used to control the clock output.

After enabling the PLL, check whether the PLL is locked before it is used by peripherals by using the CCM_ANALOG_IsPllLocked() function.

Some clock gates allow/forbid the PLL clock outputting to the system. CCM_ANALOG_EnablePfdClock() and CCM_ANALOG_DisablePfdClock() functions in the PFD control API can provide such control.

To help getting the current system PLL frequency easier, CCM_ANALOG_GetSysPllFreq() is provided to get the clock frequency in hertz.

4.2.3 PFD gate, stable, fraction and output frequency

The system PLL is equipped with the Phase Fractional Dividers(PFD) to generate the additional frequencies required by the different functional blocks.

CCM_ANALOG_EnablePfdClock() and CCM_ANALOG_DisablePfdClock() are used to allow clock outputting to functional blocks or not. In addition to the PFD clocks, some system PLL's clock output is also controlled by these functions.

After enabling the PFD clock, we need to make sure if the PFD clock is stable before getting it used by peripherals. Call CCM_ANALOG_IsPfdStable() to get the status.

To get different frequencies, fractions are needed. Call CCM_ANALOG_SetPfdFrac() to set fraction to get your required frequency. Call CCM_ANALOG_GetPfdFrac() to get current setting of fraction.

To help getting current PFD frequency easier, CCM_ANALOG_GetPfdFreq() is provided to get PFD clock frequency in HZ.

Enumerations

- enum _ccm_analog_pll_control
  PLL control names for PLL power/bypass/lock operations.
- enum _ccm_analog_pll_clock
  PLL clock names for clock enable/disable settings.
- enum _ccm_analog_pfd_clkgate
  PFD gate names for clock gate settings, clock source is system PLL(PLL_480)
- enum _ccm_analog_pfd_frac
  PFD fraction names for clock fractional divider operations.
- enum _ccm_analog_pfd_stable
  PFD stable names for clock stable query.

CCM Analog PLL Operations

- static void CCM_ANALOG_PowerUpPll (CCM_ANALOG_Type ∗base, uint32_t pllControl)
  Power up PLL.
- static void CCM_ANALOG_PowerDownPll (CCM_ANALOG_Type ∗base, uint32_t pllControl)
  Power down PLL.
- static void CCM_ANALOG_SetPllBypass (CCM_ANALOG_Type ∗base, uint32_t pllControl, bool bypass)
  PLL bypass setting.
- static bool CCM_ANALOG_IsPllBypassed (CCM_ANALOG_Type ∗base, uint32_t pllControl)
  Check if PLL is bypassed.
- static bool CCM_ANALOG_IsPllLocked (CCM_ANALOG_Type ∗base, uint32_t pllControl)
  Check if PLL clock is locked.
- static void CCM_ANALOG_EnablePllClock (CCM_ANALOG_Type ∗base, uint32_t pllClock)
  Enable PLL clock.
- static void CCM_ANALOG_DisablePllClock (CCM_ANALOG_Type ∗base, uint32_t pllClock)
  Disable PLL clock.
- uint32_t CCM_ANALOG_GetSysPllFreq (CCM_ANALOG_Type ∗base)
  Get System PLL (PLL_480) clock frequency.

CCM Analog PFD Operations

- static void CCM_ANALOG_EnablePfdClock (CCM_ANALOG_Type ∗base, uint32_t pfdClkGate)
  Enable PFD clock.

- static void CCM_ANALOG_DisablePfdClock (CCM_ANALOG_Type ∗base, uint32_t pfdClkGate)
  Disable PFD clock.
- static bool CCM_ANALOG_IsPfdStable (CCM_ANALOG_Type ∗base, uint32_t pfdStable)
  Check if PFD clock is stable.
- static void CCM_ANALOG_SetPfdFrac (CCM_ANALOG_Type ∗base, uint32_t pfdFrac, uint32_t value)
  Set PFD clock fraction.
- static uint32_t CCM_ANALOG_GetPfdFrac (CCM_ANALOG_Type ∗base, uint32_t pfdFrac)
  Get PFD clock fraction.
- uint32_t CCM_ANALOG_GetPfdFreq (CCM_ANALOG_Type ∗base, uint32_t pfdFrac)
  Get PFD clock frequency.

4.2.4 Enumeration Type Documentation

4.2.4.1 enum _ccm_analog_pll_control

These constants define the PLL control names for PLL power/bypass/lock operations.

0:15 : REG offset to CCM_ANALOG_BASE in bytes

16:20 : Power down bit shift

4.2.4.2 enum _ccm_analog_pll_clock

These constants define the PLL clock names for PLL clock enable/disable operations.

0:15 : REG offset to CCM_ANALOG_BASE in bytes

16:20 : Clock enable bit shift

4.2.4.3 enum _ccm_analog_pfd_clkgate

These constants define the PFD gate names for PFD clock enable/disable operations.

0:15 : REG offset to CCM_ANALOG_BASE in bytes

16:20 : Clock gate bit shift

4.2.4.4 enum _ccm_analog_pfd_frac

These constants define the PFD fraction names for PFD fractional divider operations.

0:15 : REG offset to CCM_ANALOG_BASE in bytes

16:20 : Fraction bits shift

4.2.4.5 enum _ccm_analog_pfd_stable

These constants define the PFD stable names for clock stable query.

0:15 : REG offset to CCM_ANALOG_BASE in bytes

16:20 : Stable bit shift

4.2.5 Function Documentation

4.2.5.1 static void CCM_ANALOG_PowerUpPll (CCM_ANALOG_Type $*$ *base,* uint32_t *pllControl*) `[inline]`, `[static]`

Parameters

base	CCM_ANALOG base pointer.
pllControl	PLL control name (see _ccm_analog_pll_control enumeration)

4.2.5.2 static void CCM_ANALOG_PowerDownPll (CCM_ANALOG_Type $*$ *base,* uint32_t *pllControl*) `[inline]`, `[static]`

Parameters

base	CCM_ANALOG base pointer.
pllControl	PLL control name (see _ccm_analog_pll_control enumeration)

4.2.5.3 static void CCM_ANALOG_SetPllBypass (CCM_ANALOG_Type $*$ *base,* uint32_t *pllControl,* bool *bypass*) `[inline]`, `[static]`

Parameters

base	CCM_ANALOG base pointer.
pllControl	PLL control name (see _ccm_analog_pll_control enumeration)
bypass	Bypass the PLL (true: bypass, false: not bypass)

4.2.5.4 static bool CCM_ANALOG_IsPllBypassed (CCM_ANALOG_Type $*$ *base,* uint32_t *pllControl*) `[inline]`, `[static]`

Parameters

base	CCM_ANALOG base pointer.
pllControl	PLL control name (see _ccm_analog_pll_control enumeration)

Returns

PLL bypass status (true: bypassed, false: not bypassed)

4.2.5.5 static bool CCM_ANALOG_IsPllLocked (CCM_ANALOG_Type ∗ *base,* uint32_t *pllControl*) [inline], [static]

Parameters

base	CCM_ANALOG base pointer.
pllControl	PLL control name (see _ccm_analog_pll_control enumeration)

Returns

PLL lock status (true: locked, false: not locked)

4.2.5.6 static void CCM_ANALOG_EnablePllClock (CCM_ANALOG_Type ∗ *base,* uint32_t *pllClock*) [inline], [static]

Parameters

base	CCM_ANALOG base pointer.
pllClock	PLL clock name (see _ccm_analog_pll_clock enumeration)

4.2.5.7 static void CCM_ANALOG_DisablePllClock (CCM_ANALOG_Type ∗ *base,* uint32_t *pllClock*) [inline], [static]

Parameters

base	CCM_ANALOG base pointer.
pllClock	PLL clock name (see _ccm_analog_pll_clock enumeration)

4.2.5.8 uint32_t CCM_ANALOG_GetSysPllFreq (CCM_ANALOG_Type ∗ *base*)

Parameters

base	CCM_ANALOG base pointer.

Returns

System PLL clock frequency in HZ

4.2.5.9 static void CCM_ANALOG_EnablePfdClock (CCM_ANALOG_Type ∗ *base*, uint32_t *pfdClkGate*) `[inline]`, `[static]`

Parameters

base	CCM_ANALOG base pointer.
pfdClkGate	PFD clock gate (see _ccm_analog_pfd_clkgate enumeration)

4.2.5.10 static void CCM_ANALOG_DisablePfdClock (CCM_ANALOG_Type ∗ *base*, uint32_t *pfdClkGate*) `[inline]`, `[static]`

Parameters

base	CCM_ANALOG base pointer.
pfdClkGate	PFD clock gate (see _ccm_analog_pfd_clkgate enumeration)

4.2.5.11 static bool CCM_ANALOG_IsPfdStable (CCM_ANALOG_Type ∗ *base*, uint32_t *pfdStable*) `[inline]`, `[static]`

Parameters

base	CCM_ANALOG base pointer.
pfdStable	PFD stable identifier (see _ccm_analog_pfd_stable enumeration)

Returns

PFD clock stable status (true: stable, false: not stable)

4.2.5.12 static void CCM_ANALOG_SetPfdFrac (CCM_ANALOG_Type ∗ *base,* uint32_t *pfdFrac,* uint32_t *value*) `[inline]`, `[static]`

Parameters

base	CCM_ANALOG base pointer.
pfdFrac	PFD clock fraction (see _ccm_analog_pfd_frac enumeration)
value	PFD clock fraction value

4.2.5.13 static uint32_t CCM_ANALOG_GetPfdFrac (CCM_ANALOG_Type ∗ *base,* uint32_t *pfdFrac*) `[inline]`, `[static]`

Parameters

base	CCM_ANALOG base pointer.
pfdFrac	PFD clock fraction (see _ccm_analog_pfd_frac enumeration)

Returns

PFD clock fraction value

4.2.5.14 uint32_t CCM_ANALOG_GetPfdFreq (CCM_ANALOG_Type ∗ *base,* uint32_t *pfdFrac*)

Parameters

base	CCM_ANALOG base pointer.
pfdFrac	PFD clock fraction (see _ccm_analog_pfd_frac enumeration)

Returns

PFD clock frequency in HZ

4.3 CCM driver

4.3.1 Overview

The chapter describes the programming interface of the CCM driver (platform/drivers/inc/ccm_imx7d.h). The Clock Control Module (CCM) provides clock routing, divider, and gate control. The CCM driver provides a set of APIs to access these control registers, including these services:

- Clock routing
- Clock divider
- Clock gate

4.3.2 Clock routing

Every CPU, bus, and peripheral can select one out of maximum 8 clock roots as the clock source. The clock root varies from OSC, PLL, PFD to external sources. Each CPU, bus or peripheral might have different sources to select. Use CCM_SetRootMux() and CCM_GetRootMux() functions to set and get clock source. Before that, the clock node (which CPU, bus or peripheral) should be decided, and enumeration *ccm_root_control used for this purpose. The clock source is also enumerated by _ccm_rootmux*<node> and every clock node has its own definition.

To make the clock source work, one additional operation is needed. Use CCM_EnableRoot() function to make the selection effective and CCM_DisableRoot() function to break the routing. To check whether the clock source is effective, use CCM_IsRootEnabled().

4.3.3 Clock divider

The clock root often has a high frequency, and, in many use cases, it needs to be divided before routing to the functional block. That's why the divider is introduced. Bus and peripheral clock slice in i.MX 7Dual have pre and post dividers and CPU slice has only post divider. Use CCM_SetRootDivider() function to set the dividers and CCM_GetRootDivider() function to get the current divider setting.

In many use cases, clock routing and divider needs to be set at once. To facilitate such usage, use the CCM_UpdateRoot() function.

4.3.4 Clock gate

After clock routing and divider are properly set, the final step to output clock to functional block is opening the gate. CCM_ControlGate() function controls the gate status. Clock root and gate are not mapped one-on-one, which means that some gates can be set from one clock root to different states.

Clock gate has 4 states:

1. Domain clocks not needed
2. Domain clocks needed when in RUN

3. Domain clocks needed when in RUN and WAIT
4. Domain clocks needed all the time
   In use cases 2 and 3, the gate closes automatically when the CPU runs into a certain power mode. Choose the gate from _ccm_ccgr_gate enumeration and set the state defined by enum _ccm_gate_-value.
   Besides the gate operation on the CPU, bus and peripheral clock root, CCM_ControlGate() function can also be used to control the PLL gate. Assign the gate with _ccm_pll_gate enumeration to achieve the same functionality.

Enumerations

- enum _ccm_root_control
  Root control names for root clock setting.
- enum _ccm_rootmux_m4
  Clock source enumeration for ARM Cortex-M4 core.
- enum _ccm_rootmux_axi
  Clock source enumeration for AXI bus.
- enum _ccm_rootmux_ahb
  Clock source enumeration for AHB bus.
- enum _ccm_rootmux_ipg
  Clock source enumeration for IPG bus.
- enum _ccm_rootmux_qspi
  Clock source enumeration for QSPI peripheral.
- enum _ccm_rootmux_can
  Clock source enumeration for CAN peripheral.
- enum _ccm_rootmux_ecspi
  Clock source enumeration for ECSPI peripheral.
- enum _ccm_rootmux_i2c
  Clock source enumeration for I2C peripheral.
- enum _ccm_rootmux_uart
  Clock source enumeration for UART peripheral.
- enum _ccm_rootmux_ftm
  Clock source enumeration for FlexTimer peripheral.
- enum _ccm_rootmux_gpt
  Clock source enumeration for GPT peripheral.
- enum _ccm_rootmux_wdog
  Clock source enumeration for WDOG peripheral.
- enum _ccm_pll_gate
  CCM PLL gate control.
- enum _ccm_ccgr_gate
  CCM CCGR gate control.
- enum _ccm_gate_value {
  ccmClockNotNeeded = 0x0U,
  ccmClockNeededRun = 0x1111U,
  ccmClockNeededRunWait = 0x2222U,
  ccmClockNeededAll = 0x3333U }
  CCM gate control value.

CCM Root Setting

- static void CCM_SetRootMux (CCM_Type ∗base, uint32_t ccmRoot, uint32_t mux)
  Set clock root mux.
- static uint32_t CCM_GetRootMux (CCM_Type ∗base, uint32_t ccmRoot)
  Get clock root mux.
- static void CCM_EnableRoot (CCM_Type ∗base, uint32_t ccmRoot)
  Enable clock root.
- static void CCM_DisableRoot (CCM_Type ∗base, uint32_t ccmRoot)
  Disable clock root.
- static bool CCM_IsRootEnabled (CCM_Type ∗base, uint32_t ccmRoot)
  Check whether clock root is enabled.
- void CCM_SetRootDivider (CCM_Type ∗base, uint32_t ccmRoot, uint32_t pre, uint32_t post)
  Set root clock divider.
- void CCM_GetRootDivider (CCM_Type ∗base, uint32_t ccmRoot, uint32_t ∗pre, uint32_t ∗post)
  Get root clock divider.
- void CCM_UpdateRoot (CCM_Type ∗base, uint32_t ccmRoot, uint32_t mux, uint32_t pre, uint32_t post)
  Update clock root in one step, for dynamical clock switching.

CCM Gate Control

- static void CCM_ControlGate (CCM_Type ∗base, uint32_t ccmGate, uint32_t control)
  Set PLL or CCGR gate control.

4.3.5 Enumeration Type Documentation

4.3.5.1 enum _ccm_gate_value

Enumerator

ccmClockNotNeeded Clock always disabled.
ccmClockNeededRun Clock enabled when CPU is running.
ccmClockNeededRunWait Clock enabled when CPU is running or in WAIT mode.
ccmClockNeededAll Clock always enabled.

4.3.6 Function Documentation

4.3.6.1 static void CCM_SetRootMux (CCM_Type ∗ *base,* uint32_t *ccmRoot,* uint32_t *mux*) `[inline]`, `[static]`

Parameters

base	CCM base pointer.
ccmRoot	Root control (see _ccm_root_control enumeration)
mux	Root mux value (see _ccm_rootmux_xxx enumeration)

4.3.6.2 static uint32_t CCM_GetRootMux (CCM_Type ∗ *base,* uint32_t *ccmRoot*) `[inline]`, `[static]`

Parameters

base	CCM base pointer.
ccmRoot	Root control (see _ccm_root_control enumeration)

Returns

root mux value (see _ccm_rootmux_xxx enumeration)

4.3.6.3 static void CCM_EnableRoot (CCM_Type ∗ *base,* uint32_t *ccmRoot*) `[inline]`, `[static]`

Parameters

base	CCM base pointer.
ccmRoot	Root control (see _ccm_root_control enumeration)

4.3.6.4 static void CCM_DisableRoot (CCM_Type ∗ *base,* uint32_t *ccmRoot*) `[inline]`, `[static]`

Parameters

base	CCM base pointer.
ccmRoot	Root control (see _ccm_root_control enumeration)

4.3.6.5 static bool CCM_IsRootEnabled (CCM_Type ∗ *base,* uint32_t *ccmRoot*) `[inline]`, `[static]`

Parameters

base	CCM base pointer.
ccmRoot	Root control (see _ccm_root_control enumeration)

Returns

CCM root enabled or not (true: enabled, false: disabled)

4.3.6.6 void CCM_SetRootDivider (CCM_Type $*$ *base,* uint32_t *ccmRoot,* uint32_t *pre,* uint32_t *post*)

Parameters

base	CCM base pointer.
ccmRoot	Root control (see _ccm_root_control enumeration)
pre	Pre divider value (0-7, divider=n+1)
post	Post divider value (0-63, divider=n+1)

4.3.6.7 void CCM_GetRootDivider (CCM_Type $*$ *base,* uint32_t *ccmRoot,* uint32_t $*$ *pre,* uint32_t $*$ *post*)

Parameters

base	CCM base pointer.
ccmRoot	Root control (see _ccm_root_control enumeration)
pre	Pointer to pre divider value store address
post	Pointer to post divider value store address

4.3.6.8 void CCM_UpdateRoot (CCM_Type $*$ *base,* uint32_t *ccmRoot,* uint32_t *mux,* uint32_t *pre,* uint32_t *post*)

Parameters

base	CCM base pointer.
ccmRoot	Root control (see _ccm_root_control enumeration)
root	mux value (see _ccm_rootmux_xxx enumeration)
pre	Pre divider value (0-7, divider=n+1)
post	Post divider value (0-63, divider=n+1)

4.3.6.9 static void CCM_ControlGate (CCM_Type ∗ *base,* uint32_t *ccmGate,* uint32_t *control*) `[inline]`, `[static]`

Parameters

base	CCM base pointer.
ccmGate	Gate control (see _ccm_pll_gate and _ccm_ccgr_gate enumeration)
control	Gate control value (see _ccm_gate_value)

Chapter 5
Enhanced Configurable Serial Peripheral Interface (ECSPI)

5.1 Overview

The FreeRTOS BSP provides a driver for the Enhanced Configurable Serial Peripheral Interface (ECSPI) of i.MX devices.

Modules

- ECSPI driver

5.2 ECSPI driver

5.2.1 Overview

This chapter describes the programming interface of the ECSPI driver (platform/drivers/inc/ecspi.h). The Enhanced Configurable Serial Peripheral Interface (ECSPI) chapter provides data transfer with DMA and interrupt mode. The ECSPI driver provides a set of APIs to access these registers, including these features:

- Data send and receive
- DMA management
- Interrupt management

5.2.2 SPI Initialization

To initialize the ECSPI module, call the ECSPI_Init() function and pass the instance of ECSPI and an initialization structure. For example, to use the ECSPI1 module, pass the ECSPI1 base pointer and a pointer pointing to the ecspi_init_t structure.

Call the ECSPI Initialization function ECSPI_Init() functions for initialization and configuration. First, configure the ecspi_init_t structure as needed. Then, call ECSPI_Init() function to complete the initialization. To set parameters not included in the ecspi_init_t structure, the driver provides a set of APIs to meet the requirements.

The following is an example of the ECSPI module initialization:

```
#define BOARD_ECSPI_MASTER_BASEADDR      ECSPI2
#define ECSPI_MASTER_BURSTLENGTH         (7)
#define BOARD_ECSPI_MASTER_CHANNEL       ecspiSelectChannel0
#define ECSPI_MASTER_STARTMODE           (0)

    // Configure the initialization structure.
    // Include clockRate, baudRate, mode, burstLength, channelSelect, clockPhase, clockPolarity,
       ecspiAutoStart
    // user can configure master and slave as needed.
    ecspi_init_t ecspiMasterInitConfig = {
        .clockRate = get_ecspi_clock_freq(BOARD_ECSPI_MASTER_BASEADDR),
        .baudRate = 500000,
        .mode = ecspiMasterMode,
        .burstLength = ECSPI_MASTER_BURSTLENGTH,
        .channelSelect = BOARD_ECSPI_MASTER_CHANNEL,
        .clockPhase = ecspiClockPhaseSecondEdge,
        .clockPolarity = ecspiClockPolarityActiveHigh,
        .ecspiAutoStart = ECSPI_MASTER_STARTMODE
    };

    // Initialize ECSPI and parameter configuration
    ECSPI_Init(BOARD_ECSPI_MASTER_BASEADDR, &ecspiMasterInitConfig);
```

In addition, for some parameters not included in ecspi_init_t structure, the driver provides specific APIs to configure them, such as these functions.

```
static inline void ECSPI_InsertWaitState(ECSPI_Type* base, uint32_t number);
void ECSPI_SetSampClockSource(ECSPI_Type* base, uint32_t source);
static inline void ECSPI_SetDelay(ECSPI_Type* base, uint32_t delay);
static inline void ECSPI_SetSCLKInactiveState(ECSPI_Type* base, uint32_t channel,
       uint32_t state);
static inline void ECSPI_SetDataInactiveState(ECSPI_Type* base, uint32_t channel,
       uint32_t state);
static inline void ECSPI_SetBurstLength(ECSPI_Type* base, uint32_t length);
static inline void ECSPI_SetSSMultipleBurst(ECSPI_Type* base, uint32_t channel,
      bool ssMultiBurst);
static inline void ECSPI_SetSSPolarity(ECSPI_Type* base, uint32_t channel, uint32_t
      polarity);
static inline void ECSPI_SetSPIDataReady(ECSPI_Type* base, uint32_t spidataready);
uint32_t ECSPI_SetBaudRate(ECSPI_Type* base, uint32_t sourceClockInHz, uint32_t bitsPerSec
      );
```

Those APIs provide settings for the wait state number, sample clock source, delay, ECSPI clock inactive state, data line inactive state, burst length, SS wave form, SS polarity, data ready signal, and baudRate.

5.2.3 ECSPI Transfers

The driver supports APIs to implement the write data to register and receive data from the register. The real transfer data functions with blocking mode are provided to the user in demos and examples.

Send data and receive data function APIs:

```
static inline void ECSPI_SendData(ECSPI_Type* base, uint32_t data);
static inline uint32_t ECSPI_ReceiveData(ECSPI_Type* base);
```

To get the number of words in FIFO, use the following APIs:

```
static inline uint32_t ECSPI_GetRxfifoCounter(ECSPI_Type* base);
static inline uint32_t ECSPI_GetTxfifoCounter(ECSPI_Type* base);
```

5.2.4 DMA Management

For the DMA operations, use the following APIs:

```
void ECSPPI_SetDMACmd(ECSPI_Type* base, uint32_t source, bool enable);
static inline void ECSPI_SetDMABurstLength(ECSPI_Type* base, uint32_t length);
```

Those APIs complete the setting of the MDA mode.

5.2.5 ECSPI Interrupt

Enable a specific ECSPI interrupt according to the ECSPI_SetIntCmd function. The following API functions are used to manage the interrupt and status flags:

```
void ECSPI_SetIntCmd(ECSPI_Type* base, uint32_t flags, bool enable);
static inline uint32_t ECSPI_GetStatusFlag(ECSPI_Type* base, uint32_t flags);
static inline void ECSPI_ClearStatusFlag(ECSPI_Type* base, uint32_t flags);
```

ECSPI_SetIntCmd() function can enable or disable specific ECSPI interrupts. ECSPI_GetStatusFlag() can check whether the specific ECSPI flag is set or not. ECSPI_ClearStatusFlag() can clear one or more ECSPI status flags.

Data Structures

- struct ecspi_init_t
  Init structure. More...

Enumerations

- enum _ecspi_channel_select {
  ecspiSelectChannel0 = 0U,
  ecspiSelectChannel1 = 1U,
  ecspiSelectChannel2 = 2U,
  ecspiSelectChannel3 = 3U }
  Channel select.
- enum _ecspi_master_slave_mode {
  ecspiSlaveMode = 0U,
  ecspiMasterMode = 1U }
  Channel mode.
- enum _ecspi_clock_phase {
  ecspiClockPhaseFirstEdge = 0U,
  ecspiClockPhaseSecondEdge = 1U }
  Clock phase.
- enum _ecspi_clock_polarity {
  ecspiClockPolarityActiveHigh = 0U,
  ecspiClockPolarityActiveLow = 1U }
  Clock polarity.
- enum _ecspi_ss_polarity {
  ecspiSSPolarityActiveLow = 0U,
  ecspiSSPolarityActiveHigh = 1U }
  SS signal polarity.
- enum _ecspi_dataline_inactivestate {
  ecspiDataLineStayHigh = 0U,
  ecspiDataLineStayLow = 1U }
  Inactive state of data line.
- enum _ecspi_sclk_inactivestate {
  ecspiSclkStayLow = 0U,
  ecspiSclkStayHigh = 1U }
  Inactive state of SCLK.
- enum _ecspi_sampleperiod_clocksource {
  ecspiSclk = 0U,
  ecspiLowFreq32K = 1U }
  sample period counter clock source.

- enum _ecspi_dma_source {
  ecspiDmaTxfifoEmpty = 7U,
  ecspiDmaRxfifoRequest = 23U,
  ecspiDmaRxfifoTail = 31U }
  DMA Source definition.
- enum _ecspi_fifothreshold {
  ecspiTxfifoThreshold = 0U,
  ecspiRxfifoThreshold = 16U }
  RXFIFO and TXFIFO threshold.
- enum _ecspi_status_flag {
  ecspiFlagTxfifoEmpty = 1U << 0,
  ecspiFlagTxfifoDataRequest = 1U << 1,
  ecspiFlagTxfifoFull = 1U << 2,
  ecspiFlagRxfifoReady = 1U << 3,
  ecspiFlagRxfifoDataRequest = 1U << 4,
  ecspiFlagRxfifoFull = 1U << 5,
  ecspiFlagRxfifoOverflow = 1U << 6,
  ecspiFlagTxfifoTc = 1U << 7 }
  Status flag.
- enum _ecspi_data_ready {
  ecspiRdyNoCare = 0U,
  ecspiRdyFallEdgeTrig = 1U,
  ecspiRdyLowLevelTrig = 2U,
  ecspiRdyReserved = 3U }
  Data Ready Control.

ECSPI Initialization and Configuration functions

- void ECSPI_Init (ECSPI_Type ∗base, ecspi_init_t ∗initStruct)
  Initializes the ECSPI module.
- static void ECSPI_Enable (ECSPI_Type ∗base)
  Enables the specified ECSPI module.
- static void ECSPI_Disable (ECSPI_Type ∗base)
  Disable the specified ECSPI module.
- static void ECSPI_InsertWaitState (ECSPI_Type ∗base, uint32_t number)
  Insert the number of wait states to be inserted in data transfers.
- void ECSPI_SetSampClockSource (ECSPI_Type ∗base, uint32_t source)
  Set the clock source for the sample period counter.
- static void ECSPI_SetDelay (ECSPI_Type ∗base, uint32_t delay)
  Set the ECSPI clocks insert between the chip select active edge and the first ECSPI clock edge.
- static void ECSPI_SetSCLKInactiveState (ECSPI_Type ∗base, uint32_t channel, uint32_t state)
  Set the inactive state of SCLK.
- static void ECSPI_SetDataInactiveState (ECSPI_Type ∗base, uint32_t channel, uint32_t state)
  Set the inactive state of data line.
- static void ECSPI_StartBurst (ECSPI_Type ∗base)
  Trigger a burst.
- static void ECSPI_SetBurstLength (ECSPI_Type ∗base, uint32_t length)

Set the burst length.
- static void ECSPI_SetSSMultipleBurst (ECSPI_Type ∗base, uint32_t channel, bool ssMultiBurst)
  Set ECSPI SS Wave Form.
- static void ECSPI_SetSSPolarity (ECSPI_Type ∗base, uint32_t channel, uint32_t polarity)
  Set ECSPI SS Polarity.
- static void ECSPI_SetSPIDataReady (ECSPI_Type ∗base, uint32_t spidataready)
  Set the Data Ready Control.
- uint32_t ECSPI_SetBaudRate (ECSPI_Type ∗base, uint32_t sourceClockInHz, uint32_t bitsPerSec)
  Calculated the ECSPI baud rate in bits per second.

Data transfers functions

- static void ECSPI_SendData (ECSPI_Type ∗base, uint32_t data)
  Transmits a data to TXFIFO.
- static uint32_t ECSPI_ReceiveData (ECSPI_Type ∗base)
  Receives a data from RXFIFO.
- static uint32_t ECSPI_GetRxfifoCounter (ECSPI_Type ∗base)
  Read the number of words in the RXFIFO.
- static uint32_t ECSPI_GetTxfifoCounter (ECSPI_Type ∗base)
  Read the number of words in the TXFIFO.

DMA management functions

- void ECSPPI_SetDMACmd (ECSPI_Type ∗base, uint32_t source, bool enable)
  Enable or disable the specified DMA Source.
- static void ECSPI_SetDMABurstLength (ECSPI_Type ∗base, uint32_t length)
  Set the burst length of a DMA operation.
- void ECSPI_SetFIFOThreshold (ECSPI_Type ∗base, uint32_t fifo, uint32_t threshold)
  Set the RXFIFO or TXFIFO threshold.

Interrupts and flags management functions

- void ECSPI_SetIntCmd (ECSPI_Type ∗base, uint32_t flags, bool enable)
  Enable or disable the specified ECSPI interrupts.
- static uint32_t ECSPI_GetStatusFlag (ECSPI_Type ∗base, uint32_t flags)
  Checks whether the specified ECSPI flag is set or not.
- static void ECSPI_ClearStatusFlag (ECSPI_Type ∗base, uint32_t flags)
  Clear one or more ECSPI status flag.

5.2.6 Data Structure Documentation

5.2.6.1 struct ecspi_init_t

Data Fields

- uint32_t clockRate
  Specifies ECSPII module clock freq.
- uint32_t baudRate
  Specifies desired ECSPI baud rate.
- uint32_t channelSelect
  Specifies the channel select.
- uint32_t mode
  Specifies the mode.
- bool ecspiAutoStart
  Specifies the start mode.
- uint32_t burstLength
  Specifies the length of a burst to be transferred.
- uint32_t clockPhase
  Specifies the clock phase.
- uint32_t clockPolarity
  Specifies the clock polarity.

5.2.6.1.0.3 Field Documentation

5.2.6.1.0.3.1 uint32_t ecspi_init_t::clockRate

5.2.6.1.0.3.2 uint32_t ecspi_init_t::baudRate

5.2.7 Enumeration Type Documentation

5.2.7.1 enum _ecspi_channel_select

Enumerator

ecspiSelectChannel0	Select Channel 0. Chip Select 0 (SS0) is asserted.
ecspiSelectChannel1	Select Channel 1. Chip Select 1 (SS1) is asserted.
ecspiSelectChannel2	Select Channel 2. Chip Select 2 (SS2) is asserted.
ecspiSelectChannel3	Select Channel 3. Chip Select 3 (SS3) is asserted.

5.2.7.2 enum _ecspi_master_slave_mode

Enumerator

ecspiSlaveMode	Set Slave Mode.
ecspiMasterMode	Set Master Mode.

5.2.7.3 enum _ecspi_clock_phase

Enumerator

ecspiClockPhaseFirstEdge Data is captured on the leading edge of the SCK and changed on the following edge.
ecspiClockPhaseSecondEdge Data is changed on the leading edge of the SCK and captured on the following edge.

5.2.7.4 enum _ecspi_clock_polarity

Enumerator

ecspiClockPolarityActiveHigh Active-high ECSPI clock (idles low)
ecspiClockPolarityActiveLow Active-low ECSPI clock (idles high)

5.2.7.5 enum _ecspi_ss_polarity

Enumerator

ecspiSSPolarityActiveLow Active-low, ECSPI SS signal.
ecspiSSPolarityActiveHigh Active-high, ECSPI SS signal.

5.2.7.6 enum _ecspi_dataline_inactivestate

Enumerator

ecspiDataLineStayHigh Data line inactive state stay high.
ecspiDataLineStayLow Data line inactive state stay low.

5.2.7.7 enum _ecspi_sclk_inactivestate

Enumerator

ecspiSclkStayLow SCLK inactive state stay low.
ecspiSclkStayHigh SCLK line inactive state stay high.

5.2.7.8 enum _ecspi_sampleperiod_clocksource

Enumerator

ecspiSclk SCLK.
ecspiLowFreq32K Low-Frequency Reference Clock (32.768 KHz)

5.2.7.9 enum _ecspi_dma_source

Enumerator

ecspiDmaTxfifoEmpty	TXFIFO Empty DMA Request.
ecspiDmaRxfifoRequest	RXFIFO DMA Request.
ecspiDmaRxfifoTail	RXFIFO TAIL DMA Request.

5.2.7.10 enum _ecspi_fifothreshold

Enumerator

ecspiTxfifoThreshold	Defines the FIFO threshold that triggers a TX DMA/INT request.
ecspiRxfifoThreshold	defines the FIFO threshold that triggers a RX DMA/INT request.

5.2.7.11 enum _ecspi_status_flag

Enumerator

ecspiFlagTxfifoEmpty	TXFIFO Empty Flag.
ecspiFlagTxfifoDataRequest	TXFIFO Data Request Flag.
ecspiFlagTxfifoFull	TXFIFO Full Flag.
ecspiFlagRxfifoReady	RXFIFO Ready Flag.
ecspiFlagRxfifoDataRequest	RXFIFO Data Request Flag.
ecspiFlagRxfifoFull	RXFIFO Full Flag.
ecspiFlagRxfifoOverflow	RXFIFO Overflow Flag.
ecspiFlagTxfifoTc	TXFIFO Transform Completed Flag.

5.2.7.12 enum _ecspi_data_ready

Enumerator

ecspiRdyNoCare	The SPI_RDY signal is a don't care.
ecspiRdyFallEdgeTrig	Burst is triggered by the falling edge of the SPI_RDY signal (edge-triggered)
ecspiRdyLowLevelTrig	Burst is triggered by a low level of the SPI_RDY signal (level-triggered)
ecspiRdyReserved	Reserved.

5.2.8 Function Documentation

5.2.8.1 void ECSPI_Init (ECSPI_Type ∗ *base,* ecspi_init_t ∗ *initStruct*)

Parameters

base,:	ECSPI base pointer.
initStruct,:	pointer to a ecspi_init_t structure.

5.2.8.2 static void ECSPI_Enable (ECSPI_Type ∗ *base*) [inline], [static]

Parameters

base	ECSPI base pointer.

5.2.8.3 static void ECSPI_Disable (ECSPI_Type ∗ *base*) [inline], [static]

Parameters

base	ECSPI base pointer.

5.2.8.4 static void ECSPI_InsertWaitState (ECSPI_Type ∗ *base,* uint32_t *number*) [inline], [static]

Parameters

base	ECSPI base pointer.
number	the number of wait states.

5.2.8.5 void ECSPI_SetSampClockSource (ECSPI_Type ∗ *base,* uint32_t *source*)

Parameters

base	ECSPI base pointer.
source	the clock source (see _ecspi_sampleperiod_clocksource).

5.2.8.6 static void ECSPI_SetDelay (ECSPI_Type ∗ *base,* uint32_t *delay*) [inline], [static]

Parameters

base	ECSPI base pointer.
delay	the number of wait states.

5.2.8.7 static void ECSPI_SetSCLKInactiveState (ECSPI_Type ∗ *base,* uint32_t *channel,* uint32_t *state*) [inline], [static]

Parameters

base	ECSPI base pointer.
channel	ECSPI channel select (see _ecspi_channel_select).
state	SCLK inactive state (see _ecspi_sclk_inactivestate).

5.2.8.8 static void ECSPI_SetDataInactiveState (ECSPI_Type ∗ *base,* uint32_t *channel,* uint32_t *state*) [inline], [static]

Parameters

base	ECSPI base pointer.
channel	ECSPI channel select (see _ecspi_channel_select).
state	Data line inactive state (see _ecspi_dataline_inactivestate).

5.2.8.9 static void ECSPI_StartBurst (ECSPI_Type ∗ *base*) [inline], [static]

Parameters

base	ECSPI base pointer.

5.2.8.10 static void ECSPI_SetBurstLength (ECSPI_Type ∗ *base,* uint32_t *length*) [inline], [static]

Parameters

base	ECSPI base pointer.
length	the value of burst length.

5.2.8.11 static void ECSPI_SetSSMultipleBurst (ECSPI_Type ∗ *base,* uint32_t *channel,* bool *ssMultiBurst*) `[inline]`, `[static]`

Parameters

base	ECSPI base pointer.
channel	ECSPI channel selected (see _ecspi_channel_select).
ssMultiBurst	For master mode, set true for multiple burst and false for one burst. For slave mode, set true to complete burst by SS signal edges and false to complete burst by number of bits received.

5.2.8.12 static void ECSPI_SetSSPolarity (ECSPI_Type ∗ *base,* uint32_t *channel,* uint32_t *polarity*) `[inline]`, `[static]`

Parameters

base	ECSPI base pointer.
channel	ECSPI channel selected (see _ecspi_channel_select).
polarity	set SS signal active logic (see _ecspi_ss_polarity).

5.2.8.13 static void ECSPI_SetSPIDataReady (ECSPI_Type ∗ *base,* uint32_t *spidataready*) `[inline]`, `[static]`

Parameters

base	ECSPI base pointer.
spidataready	ECSPI data ready control (see _ecspi_data_ready).

5.2.8.14 uint32_t ECSPI_SetBaudRate (ECSPI_Type ∗ *base,* uint32_t *sourceClockInHz,* uint32_t *bitsPerSec*)

```
The calculated baud rate must not exceed the desired baud rate.
```

Parameters

base	ECSPI base pointer.
sourceClockIn-Hz	ECSPI Clock(SCLK) (in Hz).
bitsPerSec	the value of Baud Rate.

Returns

The calculated baud rate in bits-per-second, the nearest possible baud rate without exceeding the desired baud rate.

5.2.8.15 static void ECSPI_SendData (ECSPI_Type ∗ *base,* uint32_t *data*) `[inline]`, `[static]`

Parameters

base	ECSPI base pointer.
data	Data to be transmitted.

5.2.8.16 static uint32_t ECSPI_ReceiveData (ECSPI_Type ∗ *base*) `[inline]`, `[static]`

Parameters

base	ECSPI base pointer.

Returns

The value of received data.

5.2.8.17 static uint32_t ECSPI_GetRxfifoCounter (ECSPI_Type ∗ *base*) `[inline]`, `[static]`

Parameters

base	ECSPI base pointer.

Returns

The number of words in the RXFIFO.

5.2.8.18 static uint32_t ECSPI_GetTxfifoCounter (ECSPI_Type ∗ *base*) `[inline]`, `[static]`

Parameters

base	ECSPI base pointer.

Returns

The number of words in the TXFIFO.

5.2.8.19 void ECSPPI_SetDMACmd (ECSPI_Type ∗ *base,* uint32_t *source,* bool *enable*)

Parameters

base	ECSPI base pointer.
source	specifies DMA source (see _ecspi_dma_source).
enable	True or False.

5.2.8.20 static void ECSPI_SetDMABurstLength (ECSPI_Type ∗ *base,* uint32_t *length*) `[inline]`, `[static]`

Parameters

base	ECSPI base pointer.
length	specifies the burst length of a DMA operation.

5.2.8.21 void ECSPI_SetFIFOThreshold (ECSPI_Type ∗ *base,* uint32_t *fifo,* uint32_t *threshold*)

Parameters

base	ECSPI base pointer.
fifo	Data transfer FIFO (see _ecspi_fifothreshold)
threshold	Threshold value.

5.2.8.22 void ECSPI_SetIntCmd (ECSPI_Type ∗ *base,* uint32_t *flags,* bool *enable*)

Parameters

base	ECSPI base pointer.
flags	ECSPI status flag mask (see _ecspi_status_flag for bit definition).
enable	Interrupt enable (true: enable, false: disable).

5.2.8.23 static uint32_t ECSPI_GetStatusFlag (ECSPI_Type ∗ *base,* uint32_t *flags*) `[inline]`, `[static]`

Parameters

base	ECSPI base pointer.
flags	ECSPI status flag mask (see _ecspi_status_flag for bit definition).

Returns

ECSPI status, each bit represents one status flag.

5.2.8.24 static void ECSPI_ClearStatusFlag (ECSPI_Type ∗ *base,* uint32_t *flags*) `[inline]`, `[static]`

Parameters

base	ECSPI base pointer.
flags	ECSPI status flag mask (see _ecspi_status_flag for bit definition).

Chapter 6
Flex Controller Area Network (FlexCAN)

6.1 Overview

The FreeRTOS BSP provides a driver for the Flex Controller Area Network (FlexCAN) block of i.MX devices.

Modules

- FlexCAN driver

6.2 FlexCAN driver

6.2.1 Overview

The section describes the programming interface of the FlexCAN driver(platform/drivers/inc/flexcan.h).

6.2.2 FlexCAN Initialization

To initialize the FlexCAN module, define a flexcan_init_config_t type variable and pass it to the FLEXCAN_Init() function. These are the members of the structure definition:

1. timing: The timing characteristic of CAN Bus communication defined in CAN 2.0B spec;
2. operatingMode: The operating mode of FlexCAN module with 3 modes defined in enum _flexcan_operatining_modes;
3. maxMsgBufNum: The maximum number of message buffers used for CAN communication; the unused message buffer area can be used as a normal SRAM.

The user should also set the Rx Mask Mode using the FLEXCAN_SetRxMaskMode() and the global/individual mask using FLEXCAN_SetRxGlobalMask() / FLEXCAN_SetRxIndividualMask() functions. After that, the user can send/receive messages though the FlexCAN message buffers.

6.2.3 FlexCAN Data Transactions

The FlexCAN driver provides API to acquire the Message Buffer(MB) for data transfers. All data transfers are controlled by setting the MB internal fields.

FlexCAN Data Send

To send data through the UART port, follow these steps:

1. Acquire the message buffer for data sending by calling the FLEXCAN_GetMsgBufPtr() function.
2. Fill the local priority, identifier, data length, remote frame type, identifier format, and substitute the remote request according to the application requirements.
3. Load data to the message buffer data field and write flexcanTxDataOrRemte to the message buffer code field to start the transition.
4. Call the FLEXCAN_GetMsgBufStatusFlag() function to see if the transition is finished.
5. Repeat the above process to send more data to the CAN bus.

FlexCAN Data Receive

To receive data through the UART bus, follow these steps:

1. Acquire the message buffer for data receiving by calling the FLEXCAN_GetMsgBufPtr() function.
2. Fill the message buffer with the identifier of the message you want to receive.

3. Write the flexcanRxEmpty to the message buffer code field to start the transition.
4. Call the FLEXCAN_GetMsgBufStatusFlag() function to see whether the transition is finished.
5. Call the FLEXCAN_LockRxMsgBuf() before copying the received data from Rx MB and call the FLEXCAN_UnlockAllRxMsgBuf() function to unlock all MB to guarantee the data consistency.
6. Repeat step 4 and 5 to read more data from the CAN bus.

FlexCAN Status and Interrupt

This driver provides APIs to handle the FlexCAN module error status and interrupt:

```
FLEXCAN_SetErrIntCmd()
FLEXCAN_GetErrStatusFlag()
FLEXCAN_ClearErrStatusFlag()
```

This driver provides APIs to handle the FlexCAN Message Buffer status and interrupt:

```
FLEXCAN_SetMsgBufIntCmd()
FLEXCAN_GetMsgBufStatusFlag()
FLEXCAN_ClearMsgBufStatusFlag()
```

Specific FlexCAN functions

In addition to the functions mentioned above, the FlexCAN driver also provides a set of functions for a specialized purpose, such as the Rx FIFO and FIFO mask control and functions for optimizing the communication system reliability. See the Chip Reference Manual and function descriptions below for more information about these functions.

Example

For more information about how to use this driver, see the FlexCAN demo/example under examples/<board_name>/.

Data Structures

- struct flexcan_id_table_t
  FlexCAN RX FIFO ID filter table structure. More...
- struct flexcan_msgbuf_t
  FlexCAN message buffer structure. More...
- struct flexcan_timing_t
  FlexCAN timing related structures. More...
- struct flexcan_init_config_t
  FlexCAN module initialize structure. More...

Enumerations

- enum _flexcan_msgbuf_code_rx {
  flexcanRxInactive = 0x0,
  flexcanRxFull = 0x2,
  flexcanRxEmpty = 0x4,
  flexcanRxOverrun = 0x6,
  flexcanRxBusy = 0x8,
  flexcanRxRanswer = 0xA,
  flexcanRxNotUsed = 0xF }
  FlexCAN message buffer CODE for Rx buffers.
- enum _flexcan_msgbuf_code_tx {
  flexcanTxInactive = 0x8,
  flexcanTxAbort = 0x9,
  flexcanTxDataOrRemte = 0xC,
  flexcanTxTanswer = 0xE,
  flexcanTxNotUsed = 0xF }
  FlexCAN message buffer CODE FOR Tx buffers.
- enum _flexcan_operatining_modes {
  flexCanNormalMode = 0x1,
  flexcanListenOnlyMode = 0x2,
  flexcanLoopBackMode = 0x4 }
  FlexCAN operation modes.
- enum _flexcan_rx_mask_mode {
  flexcanRxMaskGlobal = 0x0,
  flexcanRxMaskIndividual = 0x1 }
  FlexCAN RX mask mode.
- enum _flexcan_rx_mask_id_type {
  flexcanRxMaskIdStd = 0x0,
  flexcanRxMaskIdExt = 0x1 }
  The ID type used in rx matching process.
- enum _flexcan_interrutpt
  FlexCAN error interrupt source enumeration.
- enum _flexcan_status_flag
  FlexCAN error interrupt flags.
- enum _flexcan_rx_fifo_id_element_format {
  flexcanFxFifoIdElementFormatA = 0x0,
  flexcanFxFifoIdElementFormatB = 0x1,
  flexcanFxFifoIdElementFormatC = 0x2,
  flexcanFxFifoIdElementFormatD = 0x3 }
  The id filter element type selection.
- enum _flexcan_rx_fifo_filter_id_number {

flexcanRxFifoIdFilterNum8 = 0x0,
flexcanRxFifoIdFilterNum16 = 0x1,
flexcanRxFifoIdFilterNum24 = 0x2,
flexcanRxFifoIdFilterNum32 = 0x3,
flexcanRxFifoIdFilterNum40 = 0x4,
flexcanRxFifoIdFilterNum48 = 0x5,
flexcanRxFifoIdFilterNum56 = 0x6,
flexcanRxFifoIdFilterNum64 = 0x7,
flexcanRxFifoIdFilterNum72 = 0x8,
flexcanRxFifoIdFilterNum80 = 0x9,
flexcanRxFifoIdFilterNum88 = 0xA,
flexcanRxFifoIdFilterNum96 = 0xB,
flexcanRxFifoIdFilterNum104 = 0xC,
flexcanRxFifoIdFilterNum112 = 0xD,
flexcanRxFifoIdFilterNum120 = 0xE,
flexcanRxFifoIdFilterNum128 = 0xF }

FlexCAN Rx FIFO filters number.

FlexCAN Initialization and Configuration functions

- void FLEXCAN_Init (CAN_Type ∗base, flexcan_init_config_t ∗initConfig)
  Initialize FlexCAN module with given initialize structure.
- void FLEXCAN_Deinit (CAN_Type ∗base)
  This function reset FlexCAN module register content to its default value.
- void FLEXCAN_Enable (CAN_Type ∗base)
  This function is used to Enable the FlexCAN Module.
- void FLEXCAN_Disable (CAN_Type ∗base)
  This function is used to Disable the CAN Module.
- void FLEXCAN_SetTiming (CAN_Type ∗base, flexcan_timing_t ∗timing)
  Sets the FlexCAN time segments for setting up bit rate.
- void FLEXCAN_SetOperatingMode (CAN_Type ∗base, uint8_t mode)
  Set operation mode.
- void FLEXCAN_SetMaxMsgBufNum (CAN_Type ∗base, uint32_t bufNum)
  Set the maximum number of Message Buffers.
- static bool FLEXCAN_IsModuleReady (CAN_Type ∗base)
  Get the working status of FlexCAN module.
- void FLEXCAN_SetAbortCmd (CAN_Type ∗base, bool enable)
  Set the Transmit abort feature enablement.
- void FLEXCAN_SetLocalPrioCmd (CAN_Type ∗base, bool enable)
  Set the local transmit priority enablement.
- void FLEXCAN_SetMatchPrioCmd (CAN_Type ∗base, bool priority)
  Set the Rx matching process priority.

FlexCAN Message buffer control functions

- flexcan_msgbuf_t ∗ FLEXCAN_GetMsgBufPtr (CAN_Type ∗base, uint8_t msgBufIdx)

Get message buffer pointer for transition.
- bool FLEXCAN_LockRxMsgBuf (CAN_Type ∗base, uint8_t msgBufIdx)
  Locks the FlexCAN Rx message buffer.
- uint16_t FLEXCAN_UnlockAllRxMsgBuf (CAN_Type ∗base)
  Unlocks the FlexCAN Rx message buffer.

FlexCAN Interrupts and flags management functions

- void FLEXCAN_SetMsgBufIntCmd (CAN_Type ∗base, uint8_t msgBufIdx, bool enable)
  Enables/Disables the FlexCAN Message Buffer interrupt.
- bool FLEXCAN_GetMsgBufStatusFlag (CAN_Type ∗base, uint8_t msgBufIdx)
  Gets the individual FlexCAN MB interrupt flag.
- void FLEXCAN_ClearMsgBufStatusFlag (CAN_Type ∗base, uint32_t msgBufIdx)
  Clears the interrupt flag of the message buffers.
- void FLEXCAN_SetErrIntCmd (CAN_Type ∗base, uint32_t errorSrc, bool enable)
  Enables error interrupt of the FlexCAN module.
- uint32_t FLEXCAN_GetErrStatusFlag (CAN_Type ∗base, uint32_t errFlags)
  Gets the FlexCAN module interrupt flag.
- void FLEXCAN_ClearErrStatusFlag (CAN_Type ∗base, uint32_t errFlags)
  Clears the interrupt flag of the FlexCAN module.
- void FLEXCAN_GetErrCounter (CAN_Type ∗base, uint8_t ∗txError, uint8_t ∗rxError)
  Get the error counter of FlexCAN module.

Rx FIFO management functions

- void FLEXCAN_EnableRxFifo (CAN_Type ∗base, uint8_t numOfFilters)
  Enables the Rx FIFO.
- void FLEXCAN_DisableRxFifo (CAN_Type ∗base)
  Disables the Rx FIFO.
- void FLEXCAN_SetRxFifoFilterNum (CAN_Type ∗base, uint32_t numOfFilters)
  Set the number of the Rx FIFO filters.
- void FLEXCAN_SetRxFifoFilter (CAN_Type ∗base, uint32_t idFormat, flexcan_id_table_t ∗idFilterTable)
  Set the FlexCAN Rx FIFO fields.
- flexcan_msgbuf_t ∗ FLEXCAN_GetRxFifoPtr (CAN_Type ∗base)
  Gets the FlexCAN Rx FIFO data pointer.
- uint16_t FLEXCAN_GetRxFifoInfo (CAN_Type ∗base)
  Gets the FlexCAN Rx FIFO information.

Rx Mask Setting functions

- void FLEXCAN_SetRxMaskMode (CAN_Type ∗base, uint32_t mode)
  Set the Rx masking mode.
- void FLEXCAN_SetRxMaskRtrCmd (CAN_Type ∗base, uint32_t enable)
  Set the remote trasmit request mask enablement.
- void FLEXCAN_SetRxGlobalMask (CAN_Type ∗base, uint32_t mask)
  Set the FlexCAN RX global mask.

- void FLEXCAN_SetRxIndividualMask (CAN_Type ∗base, uint32_t msgBufIdx, uint32_t mask)
  Set the FlexCAN Rx individual mask for ID filtering in the Rx MBs and the Rx FIFO.
- void FLEXCAN_SetRxMsgBuff14Mask (CAN_Type ∗base, uint32_t mask)
  Set the FlexCAN RX Message Buffer BUF14 mask.
- void FLEXCAN_SetRxMsgBuff15Mask (CAN_Type ∗base, uint32_t mask)
  Set the FlexCAN RX Message Buffer BUF15 mask.
- void FLEXCAN_SetRxFifoGlobalMask (CAN_Type ∗base, uint32_t mask)
  Set the FlexCAN RX Fifo global mask.

Misc. Functions

- void FLEXCAN_SetSelfWakeUpCmd (CAN_Type ∗base, bool lpfEnable, bool enable)
  Enable/disable the FlexCAN self wakeup feature.
- void FLEXCAN_SetSelfReceptionCmd (CAN_Type ∗base, bool enable)
  Enable/disable the FlexCAN self reception feature.
- void FLEXCAN_SetRxVoteCmd (CAN_Type ∗base, bool enable)
  Enable/disable the enhance FlexCAN Rx vote.
- void FLEXCAN_SetAutoBusOffRecoverCmd (CAN_Type ∗base, bool enable)
  Enable/disable the Auto Busoff recover feature.
- void FLEXCAN_SetTimeSyncCmd (CAN_Type ∗base, bool enable)
  Enable/disable the Time Sync feature.
- void FLEXCAN_SetAutoRemoteResponseCmd (CAN_Type ∗base, bool enable)
  Enable/disable the Auto Remote Response feature.
- static void FLEXCAN_SetGlitchFilterWidth (CAN_Type ∗base, uint8_t filterWidth)
  Enable/disable the Glitch Filter Width when FLEXCAN enters the STOP mode.
- static uint32_t FLEXCAN_GetLowestInactiveMsgBuf (CAN_Type ∗base)
  Get the lowest inactive message buffer number.
- static void FLEXCAN_SetTxArbitrationStartDelay (CAN_Type ∗base, uint8_t tasd)
  Set the Tx Arbitration Start Delay number.

6.2.4 Data Structure Documentation

6.2.4.1 struct flexcan_id_table_t

Data Fields

- bool isRemoteFrame
  Remote frame.
- bool isExtendedFrame
  Extended frame.
- uint32_t ∗ idFilter
  Rx FIFO ID filter elements.

6.2.4.2 struct flexcan_msgbuf_t

6.2.4.3 struct flexcan_timing_t

Data Fields

- uint32_t preDiv
  Clock pre divider.
- uint32_t rJumpwidth
  Resync jump width.
- uint32_t phaseSeg1
  Phase segment 1.
- uint32_t phaseSeg2
  Phase segment 1.
- uint32_t propSeg
  Propagation segment.

6.2.4.4 struct flexcan_init_config_t

Data Fields

- flexcan_timing_t timing
  Desired FlexCAN module timing configuration.
- uint32_t operatingMode
  Desired FlexCAN module operating mode.
- uint8_t maxMsgBufNum
  The maximal number of available message buffer.

6.2.4.4.0.4 Field Documentation

6.2.4.4.0.4.1 flexcan_timing_t flexcan_init_config_t::timing

6.2.4.4.0.4.2 uint32_t flexcan_init_config_t::operatingMode

6.2.4.4.0.4.3 uint8_t flexcan_init_config_t::maxMsgBufNum

6.2.5 Enumeration Type Documentation

6.2.5.1 enum _flexcan_msgbuf_code_rx

Enumerator

flexcanRxInactive MB is not active.
flexcanRxFull MB is full.
flexcanRxEmpty MB is active and empty.
flexcanRxOverrun MB is overwritten into a full buffer.
flexcanRxBusy FlexCAN is updating the contents of the MB.
flexcanRxRanswer The CPU must not access the MB. A frame was configured to recognize a Re-

mote Request Frame
flexcanRxNotUsed and transmit a Response Frame in return. Not used

6.2.5.2 enum _flexcan_msgbuf_code_tx

Enumerator

flexcanTxInactive MB is not active.
flexcanTxAbort MB is aborted.
flexcanTxDataOrRemte MB is a TX Data Frame(when MB RTR = 0) or. MB is a TX Remote Request Frame (when MB RTR = 1).
flexcanTxTanswer MB is a TX Response Request Frame from.
flexcanTxNotUsed an incoming Remote Request Frame. Not used

6.2.5.3 enum _flexcan_operatining_modes

Enumerator

flexCanNormalMode Normal mode or user mode.
flexcanListenOnlyMode Listen-only mode.
flexcanLoopBackMode Loop-back mode.

6.2.5.4 enum _flexcan_rx_mask_mode

Enumerator

flexcanRxMaskGlobal Rx global mask.
flexcanRxMaskIndividual Rx individual mask.

6.2.5.5 enum _flexcan_rx_mask_id_type

Enumerator

flexcanRxMaskIdStd Standard ID.
flexcanRxMaskIdExt Extended ID.

6.2.5.6 enum _flexcan_interrutpt

6.2.5.7 enum _flexcan_status_flag

6.2.5.8 enum _flexcan_rx_fifo_id_element_format

Enumerator

flexcanFxFifoIdElementFormatA One full ID (standard and extended) per ID Filter Table.
flexcanFxFifoIdElementFormatB element. Two full standard IDs or two partial 14-bit (standard and
flexcanFxFifoIdElementFormatC extended) IDs per ID Filter Table element. Four partial 8-bit Standard IDs per ID Filter Table
flexcanFxFifoIdElementFormatD element. All frames rejected.

6.2.5.9 enum _flexcan_rx_fifo_filter_id_number

Enumerator

flexcanRxFifoIdFilterNum8 8 Rx FIFO Filters.
flexcanRxFifoIdFilterNum16 16 Rx FIFO Filters.
flexcanRxFifoIdFilterNum24 24 Rx FIFO Filters.
flexcanRxFifoIdFilterNum32 32 Rx FIFO Filters.
flexcanRxFifoIdFilterNum40 40 Rx FIFO Filters.
flexcanRxFifoIdFilterNum48 48 Rx FIFO Filters.
flexcanRxFifoIdFilterNum56 56 Rx FIFO Filters.
flexcanRxFifoIdFilterNum64 64 Rx FIFO Filters.
flexcanRxFifoIdFilterNum72 72 Rx FIFO Filters.
flexcanRxFifoIdFilterNum80 80 Rx FIFO Filters.
flexcanRxFifoIdFilterNum88 88 Rx FIFO Filters.
flexcanRxFifoIdFilterNum96 96 Rx FIFO Filters.
flexcanRxFifoIdFilterNum104 104 Rx FIFO Filters.
flexcanRxFifoIdFilterNum112 112 Rx FIFO Filters.
flexcanRxFifoIdFilterNum120 120 Rx FIFO Filters.
flexcanRxFifoIdFilterNum128 128 Rx FIFO Filters.

6.2.6 Function Documentation

6.2.6.1 void FLEXCAN_Init (CAN_Type ∗ *base,* flexcan_init_config_t ∗ *initConfig*)

Parameters

base	CAN base pointer.
initConfig	CAN initialize structure(see flexcan_init_config_t above).

6.2.6.2 void FLEXCAN_Deinit (CAN_Type ∗ *base*)

Parameters

base	FlexCAN base pointer.

6.2.6.3 void FLEXCAN_Enable (CAN_Type ∗ *base*)

Parameters

base	FlexCAN base pointer.

6.2.6.4 void FLEXCAN_Disable (CAN_Type ∗ *base*)

Parameters

base	FlexCAN base pointer.

6.2.6.5 void FLEXCAN_SetTiming (CAN_Type ∗ *base,* flexcan_timing_t ∗ *timing*)

Parameters

base	FlexCAN base pointer.
timing	FlexCAN time segments, which need to be set for the bit rate.

6.2.6.6 void FLEXCAN_SetOperatingMode (CAN_Type ∗ *base,* uint8_t *mode*)

Parameters

base	FlexCAN base pointer.
mode	Set an operation mode.

6.2.6.7 void FLEXCAN_SetMaxMsgBufNum (CAN_Type ∗ *base,* uint32_t *bufNum*)

Parameters

base	FlexCAN base pointer.
bufNum	Maximum number of message buffers

6.2.6.8 static bool FLEXCAN_IsModuleReady (CAN_Type ∗ *base*) `[inline]`, `[static]`

Parameters

base	FlexCAN base pointer.

Returns

true : FLEXCAN module is either in Normal Mode, Listen-Only Mode or Loop-Back Mode false : FLEXCAN module is either in Disable Mode, Stop Mode or Freeze Mode

6.2.6.9 void FLEXCAN_SetAbortCmd (CAN_Type ∗ *base,* bool *enable*)

Parameters

base	FlexCAN base pointer.
enable	- true : Enable Transmit Abort feature. • false : Disable Transmit Abort feature.

6.2.6.10 void FLEXCAN_SetLocalPrioCmd (CAN_Type ∗ *base,* bool *enable*)

Parameters

base	FlexCAN base pointer.
enable	- true : transmit MB with highest local priority. • false : transmit MB with lowest MB number.

6.2.6.11 void FLEXCAN_SetMatchPrioCmd (CAN_Type ∗ *base*, bool *priority*)

Parameters

base	FlexCAN base pointer.
priority	- true : Matching starts from Mailboxes and continues on Rx FIFO. • false : Matching starts from Rx FIFO and continues on Mailboxes.

6.2.6.12 flexcan_msgbuf_t∗ FLEXCAN_GetMsgBufPtr (CAN_Type ∗ *base*, uint8_t *msgBufIdx*)

Parameters

base	FlexCAN base pointer.
msgBufIdx	message buffer index.

Returns

message buffer pointer.

6.2.6.13 bool FLEXCAN_LockRxMsgBuf (CAN_Type ∗ *base*, uint8_t *msgBufIdx*)

Parameters

base	FlexCAN base pointer.

msgBuffIdx	Index of the message buffer

Returns

true : if successful; false : failed.

6.2.6.14 uint16_t FLEXCAN_UnlockAllRxMsgBuf (CAN_Type ∗ *base*)

Parameters

base	FlexCAN base pointer.

Returns

current free run timer counter value.

6.2.6.15 void FLEXCAN_SetMsgBufIntCmd (CAN_Type ∗ *base,* uint8_t *msgBufIdx,* bool *enable*)

Parameters

base	FlexCAN base pointer.
msgBuffIdx	Index of the message buffer.
enable	Choose enable or disable.

6.2.6.16 bool FLEXCAN_GetMsgBufStatusFlag (CAN_Type ∗ *base,* uint8_t *msgBufIdx*)

Parameters

base	FlexCAN base pointer.
msgBuffIdx	Index of the message buffer.

Returns

the individual Message Buffer interrupt flag (true and false are the flag value).

6.2.6.17 void FLEXCAN_ClearMsgBufStatusFlag (CAN_Type ∗ *base,* uint32_t *msgBufIdx*)

Parameters

base	FlexCAN base pointer.
msgBuffIdx	Index of the message buffer.

6.2.6.18 void FLEXCAN_SetErrIntCmd (CAN_Type ∗ *base*, uint32_t *errorSrc*, bool *enable*)

Parameters

base	FlexCAN base pointer.
errorSrc	The interrupt source.
enable	Choose enable or disable.

6.2.6.19 uint32_t FLEXCAN_GetErrStatusFlag (CAN_Type ∗ *base*, uint32_t *errFlags*)

Parameters

base	FlexCAN base pointer.
errFlags	FlexCAN error flags.

Returns

the individual Message Buffer interrupt flag (0 and 1 are the flag value)

6.2.6.20 void FLEXCAN_ClearErrStatusFlag (CAN_Type ∗ *base*, uint32_t *errFlags*)

Parameters

base	FlexCAN base pointer.
errFlags	The value to be written to the interrupt flag1 register.

6.2.6.21 void FLEXCAN_GetErrCounter (CAN_Type ∗ *base*, uint8_t ∗ *txError*, uint8_t ∗ *rxError*)

Parameters

base	FlexCAN base pointer.
txError	Tx_Err_Counter pointer.
rxError	Rx_Err_Counter pointer.

6.2.6.22 void FLEXCAN_EnableRxFifo (CAN_Type ∗ *base,* uint8_t *numOfFilters*)

Parameters

base	FlexCAN base pointer.
numOfFilters	The number of Rx FIFO filters

6.2.6.23 void FLEXCAN_DisableRxFifo (CAN_Type ∗ *base*)

Parameters

base	FlexCAN base pointer.

6.2.6.24 void FLEXCAN_SetRxFifoFilterNum (CAN_Type ∗ *base,* uint32_t *numOfFilters*)

Parameters

base	FlexCAN base pointer.
number	The number of Rx FIFO filters.

6.2.6.25 void FLEXCAN_SetRxFifoFilter (CAN_Type ∗ *base,* uint32_t *idFormat,* flexcan_id_table_t ∗ *idFilterTable*)

Parameters

base	FlexCAN base pointer.
idFormat	The format of the Rx FIFO ID Filter Table Elements
idFilterTable	The ID filter table elements which contain RTR bit, IDE bit and RX message ID.

6.2.6.26 flexcan_msgbuf_t* FLEXCAN_GetRxFifoPtr (CAN_Type * *base*)

Parameters

base	FlexCAN base pointer.

Returns

Rx FIFO data pointer.

6.2.6.27 uint16_t FLEXCAN_GetRxFifoInfo (CAN_Type * *base*)

```
The return value indicates which Identifier Acceptance Filter
(see Rx FIFO Structure) was hit by the received message.
```

Parameters

base	FlexCAN base pointer.

Returns

Rx FIFO filter number.

6.2.6.28 void FLEXCAN_SetRxMaskMode (CAN_Type * *base,* uint32_t *mode*)

Parameters

base	FlexCAN base pointer.
mode	The FlexCAN Rx mask mode: can be set to global mode and individual mode.

6.2.6.29 void FLEXCAN_SetRxMaskRtrCmd (CAN_Type * *base,* uint32_t *enable*)

Parameters

base	FlexCAN base pointer.
enable	- true : Enable RTR matching judgement. false : Disable RTR matching judgement.

6.2.6.30 void FLEXCAN_SetRxGlobalMask (CAN_Type * *base,* uint32_t *mask*)

Parameters

base	FlexCAN base pointer.
mask	Rx Global mask.

6.2.6.31 void FLEXCAN_SetRxIndividualMask (CAN_Type ∗ *base*, uint32_t *msgBufIdx*, uint32_t *mask*)

Parameters

base	FlexCAN base pointer.
msgBufIdx	Index of the message buffer.
mask	Individual mask

6.2.6.32 void FLEXCAN_SetRxMsgBuff14Mask (CAN_Type ∗ *base*, uint32_t *mask*)

Parameters

base	FlexCAN base pointer.
mask	Message Buffer BUF14 mask.

6.2.6.33 void FLEXCAN_SetRxMsgBuff15Mask (CAN_Type ∗ *base*, uint32_t *mask*)

Parameters

base	FlexCAN base pointer.
mask	Message Buffer BUF15 mask.

6.2.6.34 void FLEXCAN_SetRxFifoGlobalMask (CAN_Type ∗ *base*, uint32_t *mask*)

Parameters

base	FlexCAN base pointer.
mask	Rx Fifo Global mask.

6.2.6.35 void FLEXCAN_SetSelfWakeUpCmd (CAN_Type ∗ *base,* bool *lpfEnable,* bool *enable*)

Parameters

base	FlexCAN base pointer.
lpfEnable	The low pass filter for Rx self wakeup feature enablement.
enable	The self wakeup feature enablement.

6.2.6.36 void FLEXCAN_SetSelfReceptionCmd (CAN_Type ∗ *base,* bool *enable*)

Parameters

base	FlexCAN base pointer.
enable	- true : enable self reception feature. false : disable self reception feature.

6.2.6.37 void FLEXCAN_SetRxVoteCmd (CAN_Type ∗ *base,* bool *enable*)

Parameters

base	FlexCAN base pointer.
enable	- true : Three samples are used to determine the value of the received bit. false : Just one sample is used to determine the bit value.

6.2.6.38 void FLEXCAN_SetAutoBusOffRecoverCmd (CAN_Type ∗ *base,* bool *enable*)

Parameters

base	FlexCAN base pointer.
enable	- true : Enable Auto Bus Off recover feature. false : Disable Auto Bus Off recover feature.

6.2.6.39 void FLEXCAN_SetTimeSyncCmd (CAN_Type ∗ *base,* bool *enable*)

Parameters

base	FlexCAN base pointer.
enable	- true : Enable Time Sync feature. false : Disable Time Sync feature.

6.2.6.40 void FLEXCAN_SetAutoRemoteResponseCmd (CAN_Type * *base,* bool *enable*)

Parameters

base	FlexCAN base pointer.
enable	- true : Enable Auto Remote Response feature. false : Disable Auto Remote Response feature.

6.2.6.41 static void FLEXCAN_SetGlitchFilterWidth (CAN_Type ∗ *base,* uint8_t *filterWidth*) `[inline]`, `[static]`

Parameters

base	FlexCAN base pointer.
filterWidth	The Glitch Filter Width.

6.2.6.42 static uint32_t FLEXCAN_GetLowestInactiveMsgBuf (CAN_Type ∗ *base*) `[inline]`, `[static]`

Parameters

base	FlexCAN base pointer.

Returns

bit 22-16 : the lowest number inactive Mailbox. bit 14 : indicates whether the number content is valid or not. bit 13 : this bit indicates whether there is any inactive Mailbox.

6.2.6.43 static void FLEXCAN_SetTxArbitrationStartDelay (CAN_Type ∗ *base,* uint8_t *tasd*) `[inline]`, `[static]`

```
This function is used to optimize the transmit performance.
For more information about to set this value, please refer to Reference Manual.
```

Parameters

base	FlexCAN base pointer.

Returns

tasd The lowest number inactive Mailbox.

Chapter 7
General Purpose Input/Output (GPIO)

7.1 Overview

The FreeRTOS BSP provides a driver for the General Purpose Input/Output (GPIO) block of i.MX devices.

Modules

- GPIO driver

7.2 GPIO driver

7.2.1 Overview

This chapter describes the programming interface of the GPIO driver (platform/drivers/inc/gpio_imx.h). The GPIO driver configures pins to digital input/output or interrupt mode and provides a set of APIs to access these registers, including these services:

- GPIO pin configuration;
- GPIO pin input/output operation;
- GPIO pin interrupt management;

7.2.2 GPIO Pin Configuration

Configure GPIO pins according to the target board and ensure that the configurations are correct. Define gpio pins configuration file based on specific board to store the GPIO pin configurations.

GPIO pin configuration file example:

```
// Feel free to change the pin name, base, pin number, muxReg and padReg as what you want.
gpio_config_t gpioLed = {
    "USER LED",                          // name
    &IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO09,   // muxReg
    0,                                   // muxConfig
    &IOMUXC_SW_PAD_CTL_PAD_GPIO1_IO09,   // padReg
    0,                                   // padConfig
    GPIO1,                               // base
    9                                    // pin
};

// Configure specific GPIO pin.
configure_gpio_pin(&gpioLed);
```

7.2.3 GPIO Initialization

To initialize the GPIO module, define a structure gpio_init_t. Firstly, user need to configure the structure. Then, call the GPIO_Init() function and pass the initialize structure.

This is an example of the GPIO module Initialization:

```
#include "gpio_imx.h"

#define BOARD_GPIO_LED_CONFIG   &gpioLed

    // Configure "USER LED" as a digital output and no interrupt mode.
    gpio_init_t ledInitConfig = {
        .pin = BOARD_GPIO_LED_CONFIG->pin,
        .direction = gpioDigitalOutput,
        .interruptMode = gpioNoIntmode
    };

    //Initializes GPIO module.
    GPIO_Init(BOARD_GPIO_LED_CONFIG->base, &ledInitConfig);
```

Note: interruptMode can also be configured as a value of gpio_interrupt_mode_t.

7.2.4 Output Operations

To use the output operation, configure the target GPIO pin as a digital output in gpio_init_t structure. The output operation is provided to configure the output logic level according to passed parameters:

```
void GPIO_WritePinOutput(GPIO_Type* base, uint32_t pin,
      gpio_pin_action_t pinVal);
static inline void GPIO_WritePortOutput(GPIO_Type* base, uint32_t portVal);
```

GPIO_WritePinOutput() function is used for single pin. And GPIO_WritePortOutput() function is used for all 32 pins of a GPIO instance.

7.2.5 Input Operations

To use the input operation, configure the target GPIO pin as a digital input in the gpio_init_t structure. For the input operation, this is the most commonly used API function:

```
static inline uint8_t GPIO_ReadPinInput(GPIO_Type* base, uint32_t pin);
```

7.2.6 Read Pad Status

To use the read pad status operation, no care configuring GPIO pin as a input or output. This operation can read specific GPIO pin logic level according to passed parameters:

```
static inline uint8_t GPIO_ReadPadStatus(GPIO_Type* base, uint32_t pin);
```

7.2.7 ECSPI Interrupt

Enable a specific pin interrupt in GPIO initialization structures according to configure the interrupt mode. The following API functions are used to manage the interrupt and status flags:

```
void GPIO_SetPinIntMode(GPIO_Type* base, uint32_t pin, bool enable);
static inline bool GPIO_IsIntPending(GPIO_Type* base, uint32_t pin);
static inline void GPIO_ClearStatusFlag(GPIO_Type* base, uint32_t pin);
```

GPIO_SetPinIntMode() function can enable or disable specific GPIO pin. GPIO_IsIntPending() can check individual pin interrupt status. GPIO_ClearStatusFlag() can clear pin interrupt flag by writing a 1 to the corresponding bit position.

Data Structures

- struct gpio_init_t
  GPIO Init structure definition. More...

Enumerations

- enum gpio_pin_direction_t {
  gpioDigitalInput = 0U,
  gpioDigitalOutput = 1U }
  GPIO direction definition.
- enum gpio_interrupt_mode_t {
  gpioIntLowLevel = 0U,
  gpioIntHighLevel = 1U,
  gpioIntRisingEdge = 2U,
  gpioIntFallingEdge = 3U,
  gpioNoIntmode = 4U }
  GPIO interrupt mode definition.
- enum gpio_pin_action_t
  GPIO pin(bit) value definition.

GPIO Initialization and Configuration functions

- void GPIO_Init (GPIO_Type ∗base, gpio_init_t ∗initStruct)
  Initializes the GPIO peripheral according to the specified parameters in the initStruct.

GPIO Read and Write Functions

- static uint8_t GPIO_ReadPinInput (GPIO_Type ∗base, uint32_t pin)
  Reads the current input value of the pin when pin's direction is configured as input.
- static uint32_t GPIO_ReadPortInput (GPIO_Type ∗base)
  Reads the current input value of a specific GPIO port when port's direction are all configured as input.
- static uint8_t GPIO_ReadPinOutput (GPIO_Type ∗base, uint32_t pin)
  Reads the current pin output.
- static uint32_t GPIO_ReadPortOutput (GPIO_Type ∗base)
  Reads out all pin output status of the current port.
- void GPIO_WritePinOutput (GPIO_Type ∗base, uint32_t pin, gpio_pin_action_t pinVal)
  Sets the output level of the individual GPIO pin to logic 1 or 0.
- static void GPIO_WritePortOutput (GPIO_Type ∗base, uint32_t portVal)
  Sets the output of the GPIO port pins to a specific logic value.

GPIO Read Pad Status Functions

- static uint8_t GPIO_ReadPadStatus (GPIO_Type ∗base, uint32_t pin)
  Reads the current GPIO pin pad status.

Interrupts and flags management functions

- void GPIO_SetPinIntMode (GPIO_Type ∗base, uint32_t pin, bool enable)

Disable or enable the specific pin interrupt.
- static bool GPIO_IsIntPending (GPIO_Type ∗base, uint32_t pin)
  Check individual pin interrupt status.
- static void GPIO_ClearStatusFlag (GPIO_Type ∗base, uint32_t pin)
  Clear pin interrupt flag.
- void GPIO_SetIntEdgeSelect (GPIO_Type ∗base, uint32_t pin, bool enable)
  Disable or enable the edge select bit to override the ICR register's configuration.

7.2.8 Data Structure Documentation

7.2.8.1 struct gpio_init_t

Data Fields

- uint32_t pin
  Specifies the pin number.
- gpio_pin_direction_t direction
  Specifies the pin direction.
- gpio_interrupt_mode_t interruptMode
  Specifies the pin interrupt mode, a value of gpio_interrupt_mode_t.

7.2.8.1.0.5 Field Documentation

7.2.8.1.0.5.1 uint32_t gpio_init_t::pin

7.2.8.1.0.5.2 gpio_pin_direction_t gpio_init_t::direction

7.2.8.1.0.5.3 gpio_interrupt_mode_t gpio_init_t::interruptMode

7.2.9 Enumeration Type Documentation

7.2.9.1 enum gpio_pin_direction_t

Enumerator

gpioDigitalInput	Set current pin as digital input.
gpioDigitalOutput	Set current pin as digital output.

7.2.9.2 enum gpio_interrupt_mode_t

Enumerator

gpioIntLowLevel	Set current pin interrupt is low-level sensitive.
gpioIntHighLevel	Set current pin interrupt is high-level sensitive.
gpioIntRisingEdge	Set current pin interrupt is rising-edge sensitive.
gpioIntFallingEdge	Set current pin interrupt is falling-edge sensitive.
gpioNoIntmode	Set current pin general IO functionality.

7.2.10 Function Documentation

7.2.10.1 void GPIO_Init (GPIO_Type ∗ *base*, gpio_init_t ∗ *initStruct*)

Parameters

base	GPIO base pointer (GPIO1, GPIO2, GPIO3, and so on).
initStruct	pointer to a gpio_init_t structure that contains the configuration information.

7.2.10.2 static uint8_t GPIO_ReadPinInput (GPIO_Type ∗ *base,* uint32_t *pin*) `[inline]`, `[static]`

Parameters

base	GPIO base pointer (GPIO1, GPIO2, GPIO3, and so on).
pin	GPIO port pin number.

Returns

GPIO pin input value.

- 0: Pin logic level is 0, or is not configured for use by digital function.
- 1: Pin logic level is 1.

7.2.10.3 static uint32_t GPIO_ReadPortInput (GPIO_Type ∗ *base*) `[inline]`, `[static]`

```
This function  gets all 32-pin input as a 32-bit integer.
```

Parameters

base	GPIO base pointer(GPIO1, GPIO2, GPIO3, and so on)

Returns

GPIO port input data. Each bit represents one pin. For each bit:

- 0: Pin logic level is 0, or is not configured for use by digital function.
- 1: Pin logic level is 1.
- LSB: pin 0
- MSB: pin 31

7.2.10.4 static uint8_t GPIO_ReadPinOutput (GPIO_Type ∗ *base,* uint32_t *pin*) `[inline]`, `[static]`

Parameters

base	GPIO base pointer(GPIO1, GPIO2, GPIO3, and so on)
pin	GPIO port pin number.

Returns

current pin output value, 0 - Low logic, 1 - High logic.

7.2.10.5 static uint32_t GPIO_ReadPortOutput (GPIO_Type ∗ *base*) [inline], [static]

```
This function  operates all 32 port pins.
```

Parameters

base	GPIO base pointer(GPIO1, GPIO2, GPIO3, and so on)

Returns

current port output status. Each bit represents one pin. For each bit:
- 0: corresponding pin is outputting logic level 0
- 1: corresponding pin is outputting logic level 1
- LSB: pin 0
- MSB: pin 31

7.2.10.6 void GPIO_WritePinOutput (GPIO_Type ∗ *base,* uint32_t *pin,* gpio_pin_action_t *pinVal*)

Parameters

base	GPIO base pointer(GPIO1, GPIO2, GPIO3, and so on)
pin	GPIO port pin number.
pinVal	pin output value, one of the follow. -gpioPinClear: logic 0; -gpioPinSet: logic 1.

7.2.10.7 static void GPIO_WritePortOutput (GPIO_Type ∗ *base,* uint32_t *portVal*) [inline], [static]

```
This function  operates all 32 port pins.
```

Parameters

base	GPIO base pointer(GPIO1, GPIO2, GPIO3, and so on)
portVal	data to configure the GPIO output. Each bit represents one pin. For each bit: • 0: set logic level 0 to pin • 1: set logic level 1 to pin • LSB: pin 0 • MSB: pin 31

7.2.10.8 static uint8_t GPIO_ReadPadStatus (GPIO_Type ∗ *base*, uint32_t *pin*) `[inline]`, `[static]`

Parameters

base	GPIO base pointer (GPIO1, GPIO2, GPIO3, and so on).
pin	GPIO port pin number.

Returns

GPIO pin pad status value.
- 0: Pin pad status logic level is 0.
- 1: Pin pad status logic level is 1.

7.2.10.9 void GPIO_SetPinIntMode (GPIO_Type ∗ *base*, uint32_t *pin*, bool *enable*)

Parameters

base	GPIO base pointer(GPIO1, GPIO2, GPIO3, and so on).
pin	GPIO pin number.
enable	enable or disable interrupt.

7.2.10.10 static bool GPIO_IsIntPending (GPIO_Type ∗ *base*, uint32_t *pin*) `[inline]`, `[static]`

Parameters

base	GPIO base pointer(GPIO1, GPIO2, GPIO3, and so on)
pin	GPIO port pin number.

Returns

current pin interrupt status flag.
- 0: interrupt is not detected.
- 1: interrupt is detected.

7.2.10.11 static void GPIO_ClearStatusFlag (GPIO_Type ∗ *base,* uint32_t *pin*) [inline], [static]

Status flags are cleared by writing a 1 to the corresponding bit position.

Parameters

base	GPIO base pointer(GPIO1, GPIO2, GPIO3, and so on)
pin	GPIO port pin number.

7.2.10.12 void GPIO_SetIntEdgeSelect (GPIO_Type ∗ *base,* uint32_t *pin,* bool *enable*)

Parameters

base	GPIO base pointer(GPIO1, GPIO2, GPIO3, and so on).
pin	GPIO port pin number.
enable	enable or disable.

Chapter 8
General Purpose Timer (GPT)

8.1 Overview

The FreeRTOS BSP provides a driver for the General Purpose Timer (GPT) block of i.MX devices.

Modules

- GPT driver

8.2 GPT driver

8.2.1 Overview

The chapter describes the programming interface of the GPT driver (platform/drivers/inc/gpt.h). The GPT has a 32-bit up-counter and the counter can be captured into a register using an event on an external pin. GPT also generates an event on the output pin and an interrupt when the timer reaches a programmed value. The GPT driver provides a set of APIs to provide these services:

- GPT general setting;
- GPT input/output signal control;
- GPT interrupt control;

8.2.2 GPT general setting

Before any other function is called, GPT_Init() must be invoked. GPT_Init() initializes the module to reset state and configure the GPT behavior in different CPU modes.

To keep the GPT clock source, mode setting and reset all other configurations, GPT_SoftReset() is used. And after the function return, the reset operation is finished.

GPT counter has several source to select, including OSC(24M), low reference clock(32K), peripheral clock(GPT module clock), or external clock. Use GPT_SetClockSource() to set clock source for the counter. All the counter sources other than peripheral clock are asynchronous clock to GPT module, there's some ratio limitation if asynchronous clock source is selected. Please refer to your Device's Reference Manual for this limitation. GPT_GetClockSource() can help getting current counter clock source setting.

GPT also provide divider to make the counter clock source fit into appropriate frequency range. The user can use GPT_SetPrescaler() to set the divider, or GPT_GetPrescaler() to get current divider setting. OSC counter clock source is somehow special: it provide additional OSC divider before synchronising to GPT module. This is useful when both GPT module's clock source and counter clock source are OSC, as OSC divider helps to guarantee the ratio that meets synchronization requirement. OSC divider is controlled by GPT_SetOscPrescaler() and GPT_GetOscPrescaler(), and if both OSC divider and counter divider are set with OSC counter clock source, the final frequency is divided by product of OSC divider and counter divider.

When above GPT setting is done, GPT_Enable() can be used to start the counter, and then GPT_Disable() to stop the counter. To get current counter value, GPT_ReadCounter() can be used.

8.2.3 GPT input/output signal control

Each GPT instance has 2 capture channels and can be triggered from an external signal to capture the counter value. GPT_SetInputOperationMode() is to set the identified input channel mode to one of following 4 modes:

1. Disable capture
2. Capture on rise edge

3. Capture on fall edge
4. Capture on both edge
   Once the capture mode is set, the user can use GPT_GetInputOperationMode() to get current mode, and get captured counter value with GPT_GetInputCaptureValue() when the capture event occurs. The capture event can be got by interrupt.
   Each GPT instance has 3 output channels and GPT_SetOutputCompareValue() can be used to set the compare value for identified channel. Once the counter reaches the compare value, an event is triggered and some kind of operation on external pin occurs. The user can use GPT_SetOutputOperationMode() to set the operation when the event happen:

1. Nothing
2. Toggle the value
3. Set to low (clear)
4. Set to high (set)
5. Active low pulse
   Similarly, GPT_GetOutputOperationMode() and GPT_GetOutputCompareValue() can be used to get current setting for certain channel.
   There's a special operation which can be used to trigger the output event without comparing the counter and the compare value. GPT_ForceOutput() is for this purpose.

8.2.4 GPT interrupt control

GPT module provide 3 kinds of interrupt events: input capture, output compare and rollover. The user can use GPT_SetIntCmd() to enable or disable specific interrupt, and use GPT_GetStatusFlag() to get current event status. Once some event occurs, GPT_ClearStatusFlag() can be used to clear the event.

Data Structures

- struct gpt_mode_config_t
  Structure to configure the running mode. More...

Enumerations

- enum _gpt_clock_source {
  gptClockSourceNone = 0U,
  gptClockSourcePeriph = 1U,
  gptClockSourceLowFreq = 4U,
  gptClockSourceOsc = 5U }
  Clock source.
- enum _gpt_input_capture_channel
  Input capture channel number.
- enum _gpt_input_operation_mode {

gptInputOperationDisabled = 0U,
gptInputOperationRiseEdge = 1U,
gptInputOperationFallEdge = 2U,
gptInputOperationBothEdge = 3U }
Input capture operation mode.
- enum _gpt_output_compare_channel
  Output compare channel number.
- enum _gpt_output_operation_mode {
  gptOutputOperationDisconnected = 0U,
  gptOutputOperationToggle = 1U,
  gptOutputOperationClear = 2U,
  gptOutputOperationSet = 3U,
  gptOutputOperationActivelow = 4U }
  Output compare operation mode.
- enum _gpt_status_flag {
  gptStatusFlagOutputCompare1 = 1U << 0,
  gptStatusFlagOutputCompare2 = 1U << 1,
  gptStatusFlagOutputCompare3 = 1U << 2,
  gptStatusFlagInputCapture1 = 1U << 3,
  gptStatusFlagInputCapture2 = 1U << 4,
  gptStatusFlagRollOver = 1U << 5 }
  Status flag.

GPT State Control

- void GPT_Init (GPT_Type ∗base, gpt_mode_config_t ∗config)
  Initialize GPT to reset state and initialize running mode.
- static void GPT_SoftReset (GPT_Type ∗base)
  Software reset of GPT module.
- void GPT_SetClockSource (GPT_Type ∗base, uint32_t source)
  Set clock source of GPT.
- static uint32_t GPT_GetClockSource (GPT_Type ∗base)
  Get clock source of GPT.
- static void GPT_SetPrescaler (GPT_Type ∗base, uint32_t prescaler)
  Set pre scaler of GPT.
- static uint32_t GPT_GetPrescaler (GPT_Type ∗base)
  Get pre scaler of GPT.
- static void GPT_SetOscPrescaler (GPT_Type ∗base, uint32_t prescaler)
  OSC 24M pre scaler before selected by clock source.
- static uint32_t GPT_GetOscPrescaler (GPT_Type ∗base)
  Get pre scaler of GPT.
- static void GPT_Enable (GPT_Type ∗base)
  Enable GPT module.
- static void GPT_Disable (GPT_Type ∗base)
  Disable GPT module.
- static uint32_t GPT_ReadCounter (GPT_Type ∗base)
  Get GPT counter value.

GPT Input/Output Signal Control

- static void GPT_SetInputOperationMode (GPT_Type ∗base, uint32_t channel, uint32_t mode)
  Set GPT operation mode of input capture channel.
- static uint32_t GPT_GetInputOperationMode (GPT_Type ∗base, uint32_t channel)
  Get GPT operation mode of input capture channel.
- static uint32_t GPT_GetInputCaptureValue (GPT_Type ∗base, uint32_t channel)
  Get GPT input capture value of certain channel.
- static void GPT_SetOutputOperationMode (GPT_Type ∗base, uint32_t channel, uint32_t mode)
  Set GPT operation mode of output compare channel.
- static uint32_t GPT_GetOutputOperationMode (GPT_Type ∗base, uint32_t channel)
  Get GPT operation mode of output compare channel.
- static void GPT_SetOutputCompareValue (GPT_Type ∗base, uint32_t channel, uint32_t value)
  Set GPT output compare value of output compare channel.
- static uint32_t GPT_GetOutputCompareValue (GPT_Type ∗base, uint32_t channel)
  Get GPT output compare value of output compare channel.
- static void GPT_ForceOutput (GPT_Type ∗base, uint32_t channel)
  Force GPT output action on output compare channel, ignoring comparator.

GPT Interrupt and Status Control

- static uint32_t GPT_GetStatusFlag (GPT_Type ∗base, uint32_t flags)
  Get GPT status flag.
- static void GPT_ClearStatusFlag (GPT_Type ∗base, uint32_t flags)
  Clear one or more GPT status flag.
- void GPT_SetIntCmd (GPT_Type ∗base, uint32_t flags, bool enable)
  Enable or disable GPT interrupts.

8.2.5 Data Structure Documentation

8.2.5.1 struct gpt_mode_config_t

Data Fields

- bool freeRun
  true: FreeRun mode, false: Restart mode
- bool waitEnable
  GPT enabled in wait mode.
- bool stopEnable
  GPT enabled in stop mode.
- bool dozeEnable
  GPT enabled in doze mode.
- bool dbgEnable
  GPT enabled in debug mode.
- bool enableMode
  true: counter reset to 0 when enabled, false: counter retain its value when enabled

8.2.6 Enumeration Type Documentation

8.2.6.1 enum _gpt_clock_source

Enumerator

gptClockSourceNone	No source selected.
gptClockSourcePeriph	Use peripheral module clock.
gptClockSourceLowFreq	Use 32 K clock.
gptClockSourceOsc	Use 24 M OSC clock.

8.2.6.2 enum _gpt_input_operation_mode

Enumerator

gptInputOperationDisabled	Don't capture.
gptInputOperationRiseEdge	Capture on rising edge of input pin.
gptInputOperationFallEdge	Capture on falling edge of input pin.
gptInputOperationBothEdge	Capture on both edges of input pin.

8.2.6.3 enum _gpt_output_operation_mode

Enumerator

gptOutputOperationDisconnected	Don't change output pin.
gptOutputOperationToggle	Toggle output pin.
gptOutputOperationClear	Set output pin low.
gptOutputOperationSet	Set output pin high.
gptOutputOperationActivelow	Generate a active low pulse on output pin.

8.2.6.4 enum _gpt_status_flag

Enumerator

gptStatusFlagOutputCompare1	Output compare channel 1 event.
gptStatusFlagOutputCompare2	Output compare channel 2 event.
gptStatusFlagOutputCompare3	Output compare channel 3 event.
gptStatusFlagInputCapture1	Capture channel 1 event.
gptStatusFlagInputCapture2	Capture channel 2 event.
gptStatusFlagRollOver	Counter reaches maximum value and rolled over to 0 event.

8.2.7 Function Documentation

8.2.7.1 void GPT_Init (GPT_Type ∗ *base,* gpt_mode_config_t ∗ *config*)

Parameters

base	GPT base pointer.
config	GPT mode setting configuration.

8.2.7.2 static void GPT_SoftReset (GPT_Type ∗ *base*) `[inline]`, `[static]`

Parameters

base	GPT base pointer.

8.2.7.3 void GPT_SetClockSource (GPT_Type ∗ *base,* uint32_t *source*)

Parameters

base	GPT base pointer.
source	clock source (see _gpt_clock_source)

8.2.7.4 static uint32_t GPT_GetClockSource (GPT_Type ∗ *base*) `[inline]`, `[static]`

Parameters

base	GPT base pointer.

Returns

clock source (see _gpt_clock_source)

8.2.7.5 static void GPT_SetPrescaler (GPT_Type ∗ *base,* uint32_t *prescaler*) `[inline]`, `[static]`

Parameters

base	GPT base pointer.
prescaler	pre scaler of GPT (0-4095, divider = prescaler + 1)

8.2.7.6 static uint32_t GPT_GetPrescaler (GPT_Type * *base*) [inline], [static]

Parameters

base	GPT base pointer.

Returns

pre scaler of GPT (0-4095)

8.2.7.7 static void GPT_SetOscPrescaler (GPT_Type * *base,* uint32_t *prescaler*) [inline], [static]

Parameters

base	GPT base pointer.
prescaler	OSC pre scaler(0-15, divider = prescaler + 1)

8.2.7.8 static uint32_t GPT_GetOscPrescaler (GPT_Type * *base*) [inline], [static]

Parameters

base	GPT base pointer.

Returns

OSC pre scaler of GPT (0-15)

8.2.7.9 static void GPT_Enable (GPT_Type * *base*) [inline], [static]

Parameters

base	GPT base pointer.

8.2.7.10 static void GPT_Disable (GPT_Type * *base*) [inline], [static]

Parameters

base	GPT base pointer.

8.2.7.11 static uint32_t GPT_ReadCounter (GPT_Type * *base*) [inline], [static]

Parameters

base	GPT base pointer.

Returns

GPT counter value

8.2.7.12 static void GPT_SetInputOperationMode (GPT_Type * *base,* uint32_t *channel,* uint32_t *mode*) [inline], [static]

Parameters

base	GPT base pointer.
channel	GPT capture channel (see _gpt_input_capture_channel).
mode	GPT input capture operation mode (see _gpt_input_operation_mode).

8.2.7.13 static uint32_t GPT_GetInputOperationMode (GPT_Type * *base,* uint32_t *channel*) [inline], [static]

Parameters

base	GPT base pointer.
channel	GPT capture channel (see _gpt_input_capture_channel).

Returns

GPT input capture operation mode (see _gpt_input_operation_mode).

8.2.7.14 static uint32_t GPT_GetInputCaptureValue (GPT_Type ∗ *base,* uint32_t *channel*) `[inline]`, `[static]`

Parameters

base	GPT base pointer.
channel	GPT capture channel (see _gpt_input_capture_channel).

Returns

GPT input capture value

8.2.7.15 static void GPT_SetOutputOperationMode (GPT_Type ∗ *base,* uint32_t *channel,* uint32_t *mode*) `[inline]`, `[static]`

Parameters

base	GPT base pointer.
channel	GPT output compare channel (see _gpt_output_compare_channel).
mode	GPT output operation mode (see _gpt_output_operation_mode).

8.2.7.16 static uint32_t GPT_GetOutputOperationMode (GPT_Type ∗ *base,* uint32_t *channel*) `[inline]`, `[static]`

Parameters

base	GPT base pointer.
channel	GPT output compare channel (see _gpt_output_compare_channel).

Returns

GPT output operation mode (see _gpt_output_operation_mode).

8.2.7.17 static void GPT_SetOutputCompareValue (GPT_Type ∗ *base,* uint32_t *channel,* uint32_t *value*) `[inline]`, `[static]`

Parameters

base	GPT base pointer.
channel	GPT output compare channel (see _gpt_output_compare_channel).
value	GPT output compare value

8.2.7.18 static uint32_t GPT_GetOutputCompareValue (GPT_Type ∗ *base,* uint32_t *channel*) `[inline]`, `[static]`

Parameters

base	GPT base pointer.
channel	GPT output compare channel (see _gpt_output_compare_channel).

Returns

GPT output compare value

8.2.7.19 static void GPT_ForceOutput (GPT_Type ∗ *base,* uint32_t *channel*) `[inline]`, `[static]`

Parameters

base	GPT base pointer.
channel	GPT output compare channel (see _gpt_output_compare_channel).

8.2.7.20 static uint32_t GPT_GetStatusFlag (GPT_Type ∗ *base*, uint32_t *flags*) `[inline]`, `[static]`

Parameters

base	GPT base pointer.
flags	GPT status flag mask (see _gpt_status_flag for bit definition).

Returns

GPT status, each bit represents one status flag

8.2.7.21 static void GPT_ClearStatusFlag (GPT_Type ∗ *base*, uint32_t *flags*) `[inline]`, `[static]`

Parameters

base	GPT base pointer.
flags	GPT status flag mask (see _gpt_status_flag for bit definition).

8.2.7.22 void GPT_SetIntCmd (GPT_Type ∗ *base*, uint32_t *flags*, bool *enable*)

Parameters

base	GPT base pointer.
flags	GPT status flag mask (see _gpt_status_flag for bit definition).
enable	Interrupt enable (true: enable, false: disable).

Chapter 9
InterIntegrated Circuit (I2C)

9.1 Overview

The FreeRTOS BSP provides a driver for the InterIntegrated Circuit (I2C) block of i.MX devices.

Modules

- I2C driver

9.2 I2C driver

9.2.1 Overview

The section describes the programming interface of the I2C driver (platform/drivers/inc/i2c_imx.h).

9.2.2 I2C Initialization

To initialize the I2C module, define an i2c_init_config_t type variable and pass it to the I2C_Init() function. Here is the Members of the structure definition:

1. clockRate: Current I2C module clock freq. This variable can be obtained by calling get_i2c_clock-_freq() function;
2. baudRate: Desired I2C baud rate. The legal baud rate should not exceed 400kHz which is the highest baud rate supported by this module;
3. slaveAddress: I2C module's own address when addressed as slave device. Please watch out that this value is the I2C module's own address, not the i2c slave's address that you want to communicate with.

After I2C module Initialization, user should call I2C_Enable() to enable I2C module before any data transaction.

9.2.3 I2C Data Transactions

I2C driver provides these APIs for data transactions:

```
I2C_WriteByte
I2C_ReadByte
I2C_SendRepeatStart
I2C_SetWorkMode
I2C_SetDirMode
I2C_SetAckBit
```

I2C Data Send

To send data through I2C bus, user should follow these steps:

1. Set the I2C module work under Tx mode by calling I2C_SetDirMode();
2. Switch to Master Mode and Send Start Signal by calling I2C_SetWorkMode();
3. Send the data to I2C bus by calling I2C_WriteByte();
4. Waiting for I2C interrupt or polling I2C status bit to see if the data send successfully.

I2C Data Receive

To receive data through I2C bus, user should follow these steps:

1. Set the I2C module work under Rx mode by calling I2C_SetDirMode();
2. Switch to Master Mode and Send Start Signal by calling I2C_SetWorkMode();
3. calling I2C_ReadByte() to trigger a I2C bus Read;
4. Waiting for I2C interrupt or polling I2C status bit to see if the data received successfully;
5. Read data by calling I2C_ReadByte() function.

I2C Status and Interrupt

This driver also provide APIs to handle I2C module Status and Interrupt:

1. Calling I2C_SetIntCmd() to enable/disable I2C module interrupt;
2. Calling I2C_GetStatusFlag() to get the I2C status flags(described in enum _i2c_status_flag) condition;
3. Calling I2C_ClearStatusFlag() to clear specified status flags.

Example

For more information about how to use this driver, please refer to I2C demo/example under examples/<board_name>/.

Data Structures

- struct i2c_init_config_t
  I2C module initialize structure. More...

Enumerations

- enum _i2c_status_flag
  Flag for I2C interrupt status check or polling status.
- enum _i2c_work_mode
  I2C Bus role of this module.
- enum _i2c_direction_mode
  Data transfer direction.

Variables

- uint32_t i2c_init_config_t::clockRate
  Current I2C module clock freq.
- uint32_t i2c_init_config_t::baudRate
  Desired I2C baud rate.
- uint8_t i2c_init_config_t::slaveAddress
  I2C module's own address when addressed as slave device.

I2C Initialization and Configuration functions

- void I2C_Init (I2C_Type ∗base, i2c_init_config_t ∗initConfig)
  Initialize I2C module with given initialize structure.
- void I2C_Deinit (I2C_Type ∗base)
  This function reset I2C module register content to its default value.
- static void I2C_Enable (I2C_Type ∗base)
  This function is used to Enable the I2C Module.
- static void I2C_Disable (I2C_Type ∗base)
  This function is used to Disable the I2C Module.
- void I2C_SetBaudRate (I2C_Type ∗base, uint32_t clockRate, uint32_t baudRate)
  This function is used to set the baud rate of I2C Module.
- static void I2C_SetSlaveAddress (I2C_Type ∗base, uint8_t slaveAddress)
  This function is used to set the own I2C bus address when addressed as a slave.

I2C Bus Control functions

- static void I2C_SendRepeatStart (I2C_Type ∗base)
  This function is used to Generate a Repeat Start Signal on I2C Bus.
- static void I2C_SetWorkMode (I2C_Type ∗base, uint32_t mode)
  This function is used to select the I2C bus role of this module, both I2C Bus Master and Slave can be select.
- static void I2C_SetDirMode (I2C_Type ∗base, uint32_t direction)
  This function is used to select the data transfer direction of this module, both Transmit and Receive can be select.
- void I2C_SetAckBit (I2C_Type ∗base, bool ack)
  This function is used to set the Transmit Acknowledge action when receive data from other device.

Data transfers functions

- static void I2C_WriteByte (I2C_Type ∗base, uint8_t byte)
  Writes one byte of data to the I2C bus.
- static uint8_t I2C_ReadByte (I2C_Type ∗base)
  Returns the last byte of data read from the bus and initiate another read.

Interrupts and flags management functions

- void I2C_SetIntCmd (I2C_Type ∗base, bool enable)
  Enables or disables I2C interrupt requests.
- static uint32_t I2C_GetStatusFlag (I2C_Type ∗base, uint32_t flags)
  Gets the I2C status flag state.
- static void I2C_ClearStatusFlag (I2C_Type ∗base, uint32_t flags)
  Clear one or more I2C status flag state.

9.2.4 Data Structure Documentation

9.2.4.1 struct i2c_init_config_t

Data Fields

- uint32_t clockRate
  Current I2C module clock freq.
- uint32_t baudRate
  Desired I2C baud rate.
- uint8_t slaveAddress
  I2C module's own address when addressed as slave device.

9.2.5 Function Documentation

9.2.5.1 void I2C_Init (I2C_Type ∗ *base,* i2c_init_config_t ∗ *initConfig*)

Parameters

base	I2C base pointer.
initConfig	I2C initialize structure(see i2c_init_config_t above).

9.2.5.2 void I2C_Deinit (I2C_Type ∗ *base*)

Parameters

base	I2C base pointer.

9.2.5.3 static void I2C_Enable (I2C_Type ∗ *base*) [inline], [static]

Parameters

base	I2C base pointer.

9.2.5.4 static void I2C_Disable (I2C_Type ∗ *base*) [inline], [static]

Parameters

base	I2C base pointer.

9.2.5.5 void I2C_SetBaudRate (I2C_Type ∗ *base,* uint32_t *clockRate,* uint32_t *baudRate*)

Parameters

base	I2C base pointer.
clockRate	I2C module clock frequency.
baudRate	Desired I2C module baud rate.

9.2.5.6 static void I2C_SetSlaveAddress (I2C_Type ∗ *base,* uint8_t *slaveAddress*) `[inline]`, `[static]`

Parameters

base	I2C base pointer.
slaveAddress	Own I2C Bus address.

9.2.5.7 static void I2C_SendRepeatStart (I2C_Type ∗ *base*) `[inline]`, `[static]`

Parameters

base	I2C base pointer.

9.2.5.8 static void I2C_SetWorkMode (I2C_Type ∗ *base,* uint32_t *mode*) `[inline]`, `[static]`

Parameters

base	I2C base pointer.

mode	I2C Bus role to set (see _i2c_work_mode enumeration).

9.2.5.9 static void I2C_SetDirMode (I2C_Type * *base,* uint32_t *direction*) `[inline]`, `[static]`

Parameters

base	I2C base pointer.
direction	I2C Bus data transfer direction (see _i2c_direction_mode enumeration).

9.2.5.10 void I2C_SetAckBit (I2C_Type * *base,* bool *ack*)

Parameters

base	I2C base pointer.
ack	true: An acknowledge signal is sent to the bus at the ninth clock bit false: No acknowledge signal response is sent

9.2.5.11 static void I2C_WriteByte (I2C_Type * *base,* uint8_t *byte*) `[inline]`, `[static]`

Parameters

base	The I2C peripheral base pointer.
byte	The byte of data to transmit.

9.2.5.12 static uint8_t I2C_ReadByte (I2C_Type * *base*) `[inline]`, `[static]`

In a master receive mode, calling this function initiates receiving the next byte of data.

Parameters

base	The I2C peripheral base pointer

Returns

This function returns the last byte received while the I2C module is configured in master receive or slave receive mode.

9.2.5.13 void I2C_SetIntCmd (I2C_Type * *base,* bool *enable*)

Parameters

base	The I2C peripheral base pointer
enable	Pass true to enable interrupt, false to disable.

9.2.5.14 static uint32_t I2C_GetStatusFlag (I2C_Type * *base,* uint32_t *flags*) `[inline]`, `[static]`

Parameters

base	I2C base pointer.
flags	I2C status flag mask defined in _i2c_status_flag enumeration.

Returns

I2C status, each bit represents one status flag

9.2.5.15 static void I2C_ClearStatusFlag (I2C_Type * *base,* uint32_t *flags*) `[inline]`, `[static]`

Parameters

base	I2C base pointer.
flags	I2C status flag mask defined in _i2c_status_flag enumeration.

9.2.6 Variable Documentation

9.2.6.1 uint32_t i2c_init_config_t::clockRate

9.2.6.2 uint32_t i2c_init_config_t::baudRate

9.2.6.3 uint8_t i2c_init_config_t::slaveAddress

Chapter 10
Messaging Unit (MU)

10.1 Overview

The FreeRTOS BSP provides a driver for the Messaging Unit (MU) block of i.MX devices.

Modules

- MU driver

10.2 MU driver

10.2.1 Overview

This chapter describes the programming interface of the MU driver (platform/drivers/inc/mu_imx.h).

The MU driver provides these kinds of functions:

- Functions for send/receive message.
- Functions for general purpose interrupt.
- Functions for the flags between processor A and B.

10.2.2 Message send and receive functions

MU driver provides similar functions for message send and message receive. They are:

- Function to check whether the send/receive register is ready.
- Non-blocking function. Send/receive if the register is ready, otherwise return error status immediately.
- Blocking function. Wait until the send/receive register is ready, and send/receive the message.
- Function to enable/disable the RX/TX interrupt.

There are the functions:

```
// Functions for send message.
mu_status_t MU_TrySendMsg(MU_Type * base, uint32_t regIndex, uint32_t msg);
void MU_SendMsg(MU_Type * base, uint32_t regIndex, uint32_t msg);
bool MU_IsTxEmpty(MU_Type * base, uint32_t index);
void MU_EnableTxEmptyInt(MU_Type * base, uint32_t index);
void MU_DisableTxEmptyInt(MU_Type * base, uint32_t index);

// Functions for receive message.
mu_status_t MU_TryReceiveMsg(MU_Type * base, uint32_t regIndex, uint32_t *msg);
void MU_ReceiveMsg(MU_Type * base, uint32_t regIndex, uint32_t *msg);
bool MU_IsRxFull(MU_Type * base, uint32_t index);
void MU_EnableRxFullInt(MU_Type * base, uint32_t index);
void MU_DisableRxFullInt(MU_Type * base, uint32_t index);
```

10.2.3 General purpose interrupt functions

MU driver provides such functions for general purpose interrupt:

- Function to enable/disable the general purpose interrupt.
- Function to check and clear the general purpose interrupt pending status.
- Function to trigger general purpose interrupt to the other core.
- Function to check whether the general purpose interrupt has been processed by the other core.

The functions are:

```
void MU_EnableGeneralInt(MU_Type * base, uint32_t index);
void MU_DisableGeneralInt(MU_Type * base, uint32_t index);
bool MU_IsGeneralIntPending(MU_Type * base, uint32_t index);
```

```
void MU_ClearGeneralIntPending(MU_Type * base, uint32_t index);
mu_status_t MU_TriggerGeneralInt(MU_Type * base, uint32_t index);
bool MU_IsGeneralIntAccepted(MU_Type * base, uint32_t index);
```

Note that the enable/disable functions only control interrupt is issued or not. It means, if core B disables general purpose interrupt, and core A triggers the general purpose interrupt, then core B general purpose interrupt state is still pending, but it does not issue an interrupt.

10.2.4 Flag functions

By setting the flags on one side, the flags reflect on the other side, during the internal synchronization, it is not allowed to set flags again. Therefore, MU driver provides such functions for the MU flag:

```
mu_status_t MU_TrySetFlags(MU_Type * base, uint32_t flags);
void MU_SetFlags(MU_Type * base, uint32_t flags);
bool MU_IsFlagPending(MU_Type * base);
static inline uint32_t MU_GetFlags(MU_Type * base);
```

They are used for the non-blocking set, blocking set, pending status checking and flag check.

10.2.5 Other MU functions

MU driver provides functions for dual core boot up, clock and power setting, please check the functions for details.

Macros

- #define MU_SR_GIP0_MASK (1U<<31U)
  Bit mask for general purpose interrupt 0 pending.
- #define MU_SR_RF0_MASK (1U<<27U)
  Bit mask for RX full interrupt 0 pending.
- #define MU_SR_TE0_MASK (1U<<23U)
  Bit mask for TX empty interrupt 0 pending.
- #define MU_CR_GIE0_MASK (1U<<31U)
  Bit mask for general purpose interrupt 0 enable.
- #define MU_CR_RIE0_MASK (1U<<27U)
  Bit mask for RX full interrupt 0 enable.
- #define MU_CR_TIE0_MASK (1U<<23U)
  Bit mask for TX empty interrupt 0 enable.
- #define MU_CR_GIR0_MASK (1U<<19U)
  Bit mask to trigger general purpose interrupt 0.
- #define MU_GPn_COUNT (4U)
  Number of general purpose interrupt.

Enumerations

- enum mu_status_t {
  kStatus_MU_Success = 0U,
  kStatus_MU_TxNotEmpty = 1U,
  kStatus_MU_RxNotFull = 2U,
  kStatus_MU_FlagPending = 3U,
  kStatus_MU_EventPending = 4U,
  kStatus_MU_Initialized = 5U,
  kStatus_MU_IntPending = 6U,
  kStatus_MU_Failed = 7U }

  MU status return codes.
- enum mu_msg_status_t {
  kMuTxEmpty0 = MU_SR_TE0_MASK,
  kMuTxEmpty1 = MU_SR_TE0_MASK >> 1U,
  kMuTxEmpty2 = MU_SR_TE0_MASK >> 2U,
  kMuTxEmpty3 = MU_SR_TE0_MASK >> 3U,
  kMuTxEmpty,
  kMuRxFull0 = MU_SR_RF0_MASK,
  kMuRxFull1 = MU_SR_RF0_MASK >> 1U,
  kMuRxFull2 = MU_SR_RF0_MASK >> 2U,
  kMuRxFull3 = MU_SR_RF0_MASK >> 3U,
  kMuRxFull,
  kMuGenInt0 = MU_SR_GIP0_MASK,
  kMuGenInt1 = MU_SR_GIP0_MASK >> 1U,
  kMuGenInt2 = MU_SR_GIP0_MASK >> 2U,
  kMuGenInt3 = MU_SR_GIP0_MASK >> 3U,
  kMuGenInt,
  kMuStatusAll }

  MU message status.
- enum mu_power_mode_t {
  kMuPowerModeRun = 0x00U,
  kMuPowerModeWait = 0x01U,
  kMuPowerModeStop = 0x02U,
  kMuPowerModeDsm = 0x03U }

  Power mode definition.

Initialization.

- static void MU_Init (MU_Type ∗base)

  Initializes the MU module to reset state.

Send Messages.

- mu_status_t MU_TrySendMsg (MU_Type ∗base, uint32_t regIndex, uint32_t msg)
  Try to send a message.
- void MU_SendMsg (MU_Type ∗base, uint32_t regIndex, uint32_t msg)
  Block to send a message.
- static bool MU_IsTxEmpty (MU_Type ∗base, uint32_t index)
  Check TX empty status.
- static void MU_EnableTxEmptyInt (MU_Type ∗base, uint32_t index)
  Enable TX empty interrupt.
- static void MU_DisableTxEmptyInt (MU_Type ∗base, uint32_t index)
  Disable TX empty interrupt.

Receive Messages.

- mu_status_t MU_TryReceiveMsg (MU_Type ∗base, uint32_t regIndex, uint32_t ∗msg)
  Try to receive a message.
- void MU_ReceiveMsg (MU_Type ∗base, uint32_t regIndex, uint32_t ∗msg)
  Block to receive a message.
- static bool MU_IsRxFull (MU_Type ∗base, uint32_t index)
  Check RX full status.
- static void MU_EnableRxFullInt (MU_Type ∗base, uint32_t index)
  Enable RX full interrupt.
- static void MU_DisableRxFullInt (MU_Type ∗base, uint32_t index)
  Disable RX full interrupt.

General Purpose Interrupt.

- static void MU_EnableGeneralInt (MU_Type ∗base, uint32_t index)
  Enable general purpose interrupt.
- static void MU_DisableGeneralInt (MU_Type ∗base, uint32_t index)
  Disable general purpose interrupt.
- static bool MU_IsGeneralIntPending (MU_Type ∗base, uint32_t index)
  Check specific general purpose interrupt pending flag.
- static void MU_ClearGeneralIntPending (MU_Type ∗base, uint32_t index)
  Clear specific general purpose interrupt pending flag.
- mu_status_t MU_TriggerGeneralInt (MU_Type ∗base, uint32_t index)
  Trigger specific general purpose interrupt.
- static bool MU_IsGeneralIntAccepted (MU_Type ∗base, uint32_t index)
  Check specific general purpose interrupt is accepted or not.

Flags

- mu_status_t MU_TrySetFlags (MU_Type ∗base, uint32_t flags)
  Try to set some bits of the 3-bit flag reflect on the other MU side.
- void MU_SetFlags (MU_Type ∗base, uint32_t flags)
  Set some bits of the 3-bit flag reflect on the other MU side.

- static bool MU_IsFlagPending (MU_Type ∗base)
  Checks whether the previous flag update is pending.
- static uint32_t MU_GetFlags (MU_Type ∗base)
  Get the current value of the 3-bit flag set by other side.

Misc.

- static mu_power_mode_t MU_GetOtherCorePowerMode (MU_Type ∗base)
  Get the power mode of the other core.
- static bool MU_IsEventPending (MU_Type ∗base)
  Get the event pending status.
- static uint32_t MU_GetMsgStatus (MU_Type ∗base, uint32_t statusToCheck)
  Get the the MU message status.

10.2.6 Macro Definition Documentation

10.2.6.1 #define MU_SR_GIP0_MASK (1U<<31U)

10.2.6.2 #define MU_SR_RF0_MASK (1U<<27U)

10.2.6.3 #define MU_SR_TE0_MASK (1U<<23U)

10.2.6.4 #define MU_CR_GIE0_MASK (1U<<31U)

10.2.6.5 #define MU_CR_RIE0_MASK (1U<<27U)

10.2.6.6 #define MU_CR_TIE0_MASK (1U<<23U)

10.2.6.7 #define MU_CR_GIR0_MASK (1U<<19U)

10.2.6.8 #define MU_GPn_COUNT (4U)

10.2.7 Enumeration Type Documentation

10.2.7.1 enum mu_status_t

Enumerator

kStatus_MU_Success	Success.
kStatus_MU_TxNotEmpty	TX register is not empty.
kStatus_MU_RxNotFull	RX register is not full.
kStatus_MU_FlagPending	Previous flags update pending.
kStatus_MU_EventPending	MU event is pending.
kStatus_MU_Initialized	MU driver has initialized previously.

kStatus_MU_IntPending	Previous general interrupt still pending.
kStatus_MU_Failed	Execution failed.

10.2.7.2 enum mu_msg_status_t

Enumerator

kMuTxEmpty0	TX0 empty status.
kMuTxEmpty1	TX1 empty status.
kMuTxEmpty2	TX2 empty status.
kMuTxEmpty3	TX3 empty status.
kMuTxEmpty	TX empty status.
kMuRxFull0	RX0 full status.
kMuRxFull1	RX1 full status.
kMuRxFull2	RX2 full status.
kMuRxFull3	RX3 full status.
kMuRxFull	RX empty status.
kMuGenInt0	General purpose interrupt 0 pending status.
kMuGenInt1	General purpose interrupt 2 pending status.
kMuGenInt2	General purpose interrupt 2 pending status.
kMuGenInt3	General purpose interrupt 3 pending status.
kMuGenInt	General purpose interrupt pending status.
kMuStatusAll	All MU status.

10.2.7.3 enum mu_power_mode_t

Enumerator

kMuPowerModeRun	Run mode.
kMuPowerModeWait	WAIT mode.
kMuPowerModeStop	STOP mode.
kMuPowerModeDsm	DSM mode.

10.2.8 Function Documentation

10.2.8.1 static void MU_Init (MU_Type ∗ *base*) `[inline]`, `[static]`

This function sets the MU module control register to its default reset value.

Parameters

base	Register base address for the module.

10.2.8.2 mu_status_t MU_TrySendMsg (MU_Type ∗ *base,* uint32_t *regIndex,* uint32_t *msg*)

This function tries to send a message, if the TX register is not empty, this function returns kStatus_MU_-TxNotEmpty.

Parameters

base	Register base address for the module.
regIdex	Tx register index.
msg	Message to send.

Return values

kStatus_MU_Success	Message send successfully.
kStatus_MU_TxNotEmpty	Message not send because TX is not empty.

10.2.8.3 void MU_SendMsg (MU_Type ∗ *base,* uint32_t *regIndex,* uint32_t *msg*)

This function waits until TX register is empty and send the message.

Parameters

base	Register base address for the module.
regIdex	Tx register index.
msg	Message to send.

10.2.8.4 static bool MU_IsTxEmpty (MU_Type ∗ *base,* uint32_t *index*) `[inline]`, `[static]`

This function checks the specific transmit register empty status.

Parameters

base	Register base address for the module.
index	TX register index to check.

Return values

true	TX register is empty.
false	TX register is not empty.

10.2.8.5 static void MU_EnableTxEmptyInt (MU_Type ∗ *base,* uint32_t *index*) [inline], [static]

This function enables specific TX empty interrupt.

Parameters

base	Register base address for the module.
index	TX interrupt index to enable.

Example:

```
// To enable TX0 empty interrupts.
MU_EnableTxEmptyInt(MU0_BASE, 0U);
```

10.2.8.6 static void MU_DisableTxEmptyInt (MU_Type ∗ *base,* uint32_t *index*) [inline], [static]

This function disables specific TX empty interrupt.

Parameters

base	Register base address for the module.
disableMask	Bitmap of the interrupts to disable.

Example:

```
// To disable TX0 empty interrupts.
MU_DisableTxEmptyInt(MU0_BASE, 0U);
```

10.2.8.7 mu_status_t MU_TryReceiveMsg (MU_Type ∗ *base,* uint32_t *regIndex,* uint32_t ∗ *msg*)

This function tries to receive a message, if the RX register is not full, this function returns kStatus_MU_-RxNotFull.

Parameters

base	Register base address for the module.
regIdex	Rx register index.
msg	Message to receive.

Return values

kStatus_MU_Success	Message receive successfully.
kStatus_MU_RxNotFull	Message not received because RX is not full.

10.2.8.8 void MU_ReceiveMsg (MU_Type ∗ *base,* uint32_t *regIndex,* uint32_t ∗ *msg*)

This function waits until RX register is full and receive the message.

Parameters

base	Register base address for the module.
regIdex	Rx register index.
msg	Message to receive.

10.2.8.9 static bool MU_IsRxFull (MU_Type ∗ *base,* uint32_t *index*) `[inline]`, `[static]`

This function checks the specific receive register full status.

Parameters

base	Register base address for the module.
index	RX register index to check.

Return values

true	RX register is full.
false	RX register is not full.

10.2.8.10 static void MU_EnableRxFullInt (MU_Type ∗ *base,* uint32_t *index*) `[inline]`, `[static]`

This function enables specific RX full interrupt.

Parameters

base	Register base address for the module.
index	RX interrupt index to enable.

Example:

```
// To enable RX0 full interrupts.
MU_EnableRxFullInt(MU0_BASE, 0U);
```

10.2.8.11 static void MU_DisableRxFullInt (MU_Type ∗ *base*, uint32_t *index*) [inline], [static]

This function disables specific RX full interrupt.

Parameters

base	Register base address for the module.
disableMask	Bitmap of the interrupts to disable.

Example:

```
// To disable RX0 full interrupts.
MU_DisableRxFullInt(MU0_BASE, 0U);
```

10.2.8.12 static void MU_EnableGeneralInt (MU_Type ∗ *base*, uint32_t *index*) [inline], [static]

This function enables specific general purpose interrupt.

Parameters

base	Register base address for the module.
index	General purpose interrupt index to enable.

Example:

```
// To enable general purpose interrupts 0.
MU_EnableGeneralInt(MU0_BASE, 0U);
```

10.2.8.13 static void MU_DisableGeneralInt (MU_Type ∗ *base*, uint32_t *index*) [inline], [static]

This function disables specific general purpose interrupt.

Parameters

base	Register base address for the module.
index	General purpose interrupt index to disable.

Example:

```
// To disable general purpose interrupts 0.
MU_DisableGeneralInt(MU0_BASE, 0U);
```

10.2.8.14 static bool MU_IsGeneralIntPending (MU_Type ∗ *base,* uint32_t *index*) [inline], [static]

This function checks the specific general purpose interrupt pending status.

Parameters

base	Register base address for the module.
index	Index of the general purpose interrupt flag to check.

Return values

true	General purpose interrupt is pending.
false	General purpose interrupt is not pending.

10.2.8.15 static void MU_ClearGeneralIntPending (MU_Type ∗ *base,* uint32_t *index*) [inline], [static]

This function clears the specific general purpose interrupt pending status.

Parameters

base	Register base address for the module.
index	Index of the general purpose interrupt flag to clear.

10.2.8.16 mu_status_t MU_TriggerGeneralInt (MU_Type ∗ *base,* uint32_t *index*)

This function triggers specific general purpose interrupt to other core.

To ensure proper operations, please make sure the correspond general purpose interrupt triggered previously has been accepted by the other core. The function MU_IsGeneralIntAccepted can be used for this check. If the previous general interrupt has not been accepted by the other core, this function does not trigger interrupt actually and returns an error.

Parameters

base	Register base address for the module.
index	Index of general purpose interrupt to trigger.

Return values

kStatus_MU_Success	Interrupt has been triggered successfully.
kStatus_MU_IntPending	Previous interrupt has not been accepted.

10.2.8.17 static bool MU_IsGeneralIntAccepted (MU_Type ∗ *base,* uint32_t *index*) `[inline]`, `[static]`

This function checks whether the specific general purpose interrupt has been accepted by the other core or not.

Parameters

base	Register base address for the module.
index	Index of the general purpose interrupt to check.

Return values

true	General purpose interrupt is accepted.
false	General purpose interrupt is not accepted.

10.2.8.18 mu_status_t MU_TrySetFlags (MU_Type ∗ *base,* uint32_t *flags*)

This functions tries to set some bits of the 3-bit flag. If previous flags update is still pending, this function returns kStatus_MU_FlagPending.

Parameters

base	Register base address for the module.

Return values

kStatus_MU_Success	Flag set successfully.
kStatus_MU_Flag-Pending	Previous flag update is pending.

10.2.8.19 void MU_SetFlags (MU_Type ∗ *base*, uint32_t *flags*)

This functions set some bits of the 3-bit flag. If previous flags update is still pending, this function blocks and polls to set the flag.

Parameters

base	Register base address for the module.

10.2.8.20 static bool MU_IsFlagPending (MU_Type ∗ *base*) [inline], [static]

After setting flags, the flags update request is pending until internally acknowledged. During the pending period, it is not allowed to set flags again. This function is used to check the pending status, it can be used together with function MU_TrySetFlags.

Parameters

base	Register base address for the module.

Returns

True if pending, false if not.

10.2.8.21 static uint32_t MU_GetFlags (MU_Type ∗ *base*) [inline], [static]

This functions gets the current value of the 3-bit flag.

Parameters

base	Register base address for the module.

Returns

flags Current value of the 3-bit flag.

10.2.8.22 static mu_power_mode_t MU_GetOtherCorePowerMode (MU_Type * *base*) `[inline]`, `[static]`

This functions gets the power mode of the other core.

Parameters

base	Register base address for the module.

Returns

powermode Power mode of the other core.

10.2.8.23 static bool MU_IsEventPending (MU_Type ∗ *base*) [inline], [static]

This functions gets the event pending status. To ensure events have been posted to the other side before entering STOP mode, please verify the event pending status using this function.

Parameters

base	Register base address for the module.

Return values

true	Event is pending.
false	Event is not pending.

10.2.8.24 static uint32_t MU_GetMsgStatus (MU_Type ∗ *base,* uint32_t *statusToCheck*) [inline], [static]

This functions gets TX/RX and general purpose interrupt pending status. The parameter is passed in as bitmask of the status to check.

Parameters

base	Register base address for the module.
statusToCheck	The status to check, see mu_msg_status_t.

Returns

Status checked.

Example:

```
// To check TX0 empty status.
MU_GetMsgStatus(MU0_BASE, kMuTxEmpty0);

// To check all RX full status.
MU_GetMsgStatus(MU0_BASE, kMuRxFull);
```

```
// To check general purpose interrupt 0 and 3 pending status.
MU_GetMsgStatus(MU0_BASE, kMuGenInt0 | kMuGenInt3);

// To check all status.
MU_GetMsgStatus(MU0_BASE, kMuStatusAll);
```

Chapter 11
Resource Domain Controller (RDC)

11.1 Overview

The FreeRTOS BSP provides a driver for the Resource Domain Controller (RDC) block of i.MX devices.

Modules

- RDC Semaphore driver
- RDC definitions on i.MX 7Dual
- RDC driver

11.2 RDC driver

11.2.1 Overview

The chapter describes the programming interface of the RDC driver (platform/drivers/inc/rdc.h). The RD-C provides robust support for isolation of peripherals and memory among different bus masters. The RDC driver provides a set of APIs to provide these services:

- RDC domain control;
- RDC status control;

11.2.2 RDC domain control

RDC defines "domain" which is the unit of the access control. One or more bus masters can be put into one domain to get all the domain's permission to access peripherals or memory. Each bus master is allocated a unique RDC master ID called "MDA", and the user can find the MDA enumeration *rdc_mda in rdc_defs*<device>.h, where <device> specifies the i.MX device name. Normally i.MX devices have 4 domains with ID from 0 to 3, and RDC_SetDomainID() can be used to put some MDA into one of those domains. To avoid malfunction, the setting can be locked with "lock" parameter. RDC_GetDomainID() is used to check which domain is some MDA associated.

There are some functions related to peripheral access permission for certain domain. Just like MDA, each peripheral has a RDC peripheral ID called "PDAP", which is also defined in rdc_defs_<device>.h. R-DC_SetPdapAccess() is to set PDAP permission for each domain, and "lock" parameter is also available to avoid further change of this setting. If some peripheral need to be accessed by multiple MDA, access conflict must be resolved. RDC provides RDC SEMAPHORE to allow each MDA to access the peripheral exclusively. And when using RDC_SetPdapAccess(), the user can also force RDC SEMAPHORE being acquired before peripheral access. Parameter "sreq" is for this purpose. RDC_GetPdapAccess() and RD-C_IsPdapSemaphoreRequired() can be used to check current setting of some PDAP.

Besides peripheral access, memory can also be protected by RDC. Here memory can be QSPI, DDR, O-CRAM, PCIE, and so on. Each type of memory can have several regions in RDC, with different access permission for each region. RDC memory region setting must be enabled before it take effects. RDC_-SetMrAccess() is used for memory region access permission setting, and RDC memory region (defined in rdc_defs_<device>.h) type has to be matched with the [startAddr, endAddr) area. RDC_GetMrAccess() and RDC_IsMrEnabled() are used to check current setting of some memory region. In additional, the user can also use RDC_GetViolationStatus() to get the memory region's violation address and which domain causes this violation. Once violation occurs and gets properly handled, RDC_ClearViolationStatus() is used to clear the violation status.

11.2.3 RDC status control

RDC provides a interrupt which indicates the memory region setting restoration has completed. This is useful when some memory region as well as the RDC memory region configuration is powered off in low power mode. The interrupt can guarantee the memory and memory access configuration work before the bus master accesses this region. RDC_IsMemPowered(), RDC_IsIntPending() and RDC_ClearStatus-Flag() are functions for low power recovery. RDC_GetSelfDomainID() is used to return the domain ID of running CPU.

RDC State Control

- static uint32_t RDC_GetSelfDomainID (RDC_Type ∗base)
  Get domain ID of core that is reading this.
- static bool RDC_IsMemPowered (RDC_Type ∗base)
  Check whether memory region controlled by RDC is accessible after low power recovery.
- static bool RDC_IsIntPending (RDC_Type ∗base)
  Check whether there's pending RDC memory region restoration interrupt.
- static void RDC_ClearStatusFlag (RDC_Type ∗base)
  Clear interrupt status.
- static void RDC_SetIntCmd (RDC_Type ∗base, bool enable)
  Set RDC interrupt mode.

RDC Domain Control

- static void RDC_SetDomainID (RDC_Type ∗base, uint32_t mda, uint32_t domainId, bool lock)
  Set RDC domain ID for RDC master.
- static uint32_t RDC_GetDomainID (RDC_Type ∗base, uint32_t mda)
  Get RDC domain ID for RDC master.
- static void RDC_SetPdapAccess (RDC_Type ∗base, uint32_t pdap, uint8_t perm, bool sreq, bool lock)
  Set RDC peripheral access permission for RDC domains.
- static uint8_t RDC_GetPdapAccess (RDC_Type ∗base, uint32_t pdap)
  Get RDC peripheral access permission for RDC domains.
- static bool RDC_IsPdapSemaphoreRequired (RDC_Type ∗base, uint32_t pdap)
  Check whether RDC semaphore is required to access the peripheral.
- void RDC_SetMrAccess (RDC_Type ∗base, uint32_t mr, uint32_t startAddr, uint32_t endAddr, uint8_t perm, bool enable, bool lock)
  Set RDC memory region access permission for RDC domains.
- uint8_t RDC_GetMrAccess (RDC_Type ∗base, uint32_t mr, uint32_t ∗startAddr, uint32_t ∗end-Addr)
  Get RDC memory region access permission for RDC domains.
- static bool RDC_IsMrEnabled (RDC_Type ∗base, uint32_t mr)
  Check whether the memory region is enabled.
- bool RDC_GetViolationStatus (RDC_Type ∗base, uint32_t mr, uint32_t ∗violationAddr, uint32_t ∗violationDomain)
  Get memory violation status.
- static void RDC_ClearViolationStatus (RDC_Type ∗base, uint32_t mr)
  Clear RDC violation status.

11.2.4 Function Documentation

11.2.4.1 static uint32_t RDC_GetSelfDomainID (RDC_Type ∗ *base*) [inline], [static]

Parameters

base	RDC base pointer.

Returns

Domain ID of self core

11.2.4.2 static bool RDC_IsMemPowered (RDC_Type ∗ *base*) [inline], [static]

Parameters

base	RDC base pointer.

Returns

Memory region power status (true: on and accessible, false: off)

11.2.4.3 static bool RDC_IsIntPending (RDC_Type ∗ *base*) [inline], [static]

Parameters

base	RDC base pointer.

Returns

RDC interrupt status (true: interrupt pending, false: no interrupt pending)

11.2.4.4 static void RDC_ClearStatusFlag (RDC_Type ∗ *base*) [inline], [static]

Parameters

base	RDC base pointer.

11.2.4.5 static void RDC_SetIntCmd (RDC_Type ∗ *base*, bool *enable*) [inline], [static]

Parameters

base	RDC base pointer
enable	RDC interrupt control (true: enable, false: disable)

11.2.4.6 static void RDC_SetDomainID (RDC_Type ∗ *base,* uint32_t *mda,* uint32_t *domainId,* bool *lock*) `[inline]`, `[static]`

Parameters

base	RDC base pointer
mda	RDC master assignment (see *rdc_mda in rdc_defs*<device>.h)
domainId	RDC domain ID (0-3)
lock	Whether to lock this setting? Once locked, no one can change the domain assignment until reset

11.2.4.7 static uint32_t RDC_GetDomainID (RDC_Type ∗ *base,* uint32_t *mda*) `[inline]`, `[static]`

Parameters

base	RDC base pointer
mda	RDC master assignment (see *rdc_mda in rdc_defs*<device>.h)

Returns

RDC domain ID (0-3)

11.2.4.8 static void RDC_SetPdapAccess (RDC_Type ∗ *base,* uint32_t *pdap,* uint8_t *perm,* bool *sreq,* bool *lock*) `[inline]`, `[static]`

Parameters

base	RDC base pointer
pdap	RDC peripheral assignment (see *rdc_pdap in rdc_defs*<device>.h)
perm	RDC access permission from RDC domain to peripheral (byte: D3R D3W D2R D2W D1R D1W D0R D0W)
sreq	Force acquiring SEMA42 to access this peripheral or not
lock	Whether to lock this setting or not. Once locked, no one can change the RDC setting until reset

11.2.4.9 static uint8_t RDC_GetPdapAccess (RDC_Type ∗ *base,* uint32_t *pdap*) `[inline]`, `[static]`

Parameters

base	RDC base pointer
pdap	RDC peripheral assignment (see *rdc_pdap in rdc_defs*<device>.h)

Returns

RDC access permission from RDC domain to peripheral (byte: D3R D3W D2R D2W D1R D1W D0R D0W)

11.2.4.10 static bool RDC_IsPdapSemaphoreRequired (RDC_Type ∗ *base,* uint32_t *pdap*) `[inline]`, `[static]`

Parameters

base	RDC base pointer
pdap	RDC peripheral assignment (see *rdc_pdap in rdc_defs*<device>.h)

Returns

RDC semaphore required or not (true: required, false: not required)

11.2.4.11 void RDC_SetMrAccess (RDC_Type ∗ *base,* uint32_t *mr,* uint32_t *startAddr,* uint32_t *endAddr,* uint8_t *perm,* bool *enable,* bool *lock*)

Parameters

base	RDC base pointer
mr	RDC memory region assignment (see *rdc_mr in rdc_defs*<device>.h)
startAddr	memory region start address (inclusive)
endAddr	memory region end address (exclusive)
perm	RDC access permission from RDC domain to peripheral (byte: D3R D3W D2R D2W D1R D1W D0R D0W)
enable	Enable this memory region for RDC control or not
lock	Whether to lock this setting or not. Once locked, no one can change the RDC setting until reset

11.2.4.12 uint8_t RDC_GetMrAccess (RDC_Type * *base,* uint32_t *mr,* uint32_t * *startAddr,* uint32_t * *endAddr*)

Parameters

base	RDC base pointer
mr	RDC memory region assignment (see *rdc_mr in rdc_defs*<device>.h)
startAddr	pointer to get memory region start address (inclusive), NULL is allowed.
endAddr	pointer to get memory region end address (exclusive), NULL is allowed.

Returns

RDC access permission from RDC domain to peripheral (byte: D3R D3W D2R D2W D1R D1W D0R D0W)

11.2.4.13 static bool RDC_IsMrEnabled (RDC_Type * *base,* uint32_t *mr*) `[inline]`, `[static]`

Parameters

base	RDC base pointer

mr	RDC memory region assignment (see *rdc_mr in rdc_defs*<device>.h)

Returns

Memory region enabled or not (true: enabled, false: not enabled)

11.2.4.14 bool RDC_GetViolationStatus (RDC_Type ∗ *base*, uint32_t *mr*, uint32_t ∗ *violationAddr*, uint32_t ∗ *violationDomain*)

Parameters

base	RDC base pointer
mr	RDC memory region assignment (see *rdc_mr in rdc_defs*<device>.h)
violationAddr	Pointer to store violation address, NULL allowed
violation-Domain	Pointer to store domain ID causing violation, NULL allowed

Returns

Memory violation occurred or not (true: violation happened, false: no violation happened)

11.2.4.15 static void RDC_ClearViolationStatus (RDC_Type ∗ *base*, uint32_t *mr*) `[inline]`, `[static]`

Parameters

base	RDC base pointer
mr	RDC memory region assignment (see *rdc_mr in rdc_defs*<device>.h)

11.3 RDC definitions on i.MX 7Dual

11.3.1 Overview

The chapter describes the RDC MDA, PDAP and Memory Region definitions (platform/drivers/inc/rdc_-defs_imx7d.h).

Enumerations

- enum _rdc_mda
  RDC master assignment.
- enum _rdc_pdap
  RDC peripheral assignment.
- enum _rdc_mr
  RDC memory region.

11.4 RDC Semaphore driver

11.4.1 Overview

The chapter describes the programming interface of the RDC Semaphore driver (platform/drivers/inc/rdc_-semaphore.h). The RDC SEMAPHORE provides hardware semaphores for peripheral exclusively access. The RDC SEMAPHORE driver provides a set of APIs to provide these services:

- RDC SEMAPHORE lock/unlock control;
- RDC SEMAPHORE reset control;

11.4.2 RDC SEMAPHORE lock/unlock control

Peripheral can be configured in RDC to access with hardware semaphore. In this mode, accessing it without acquiring the semaphore first will cause violation, even if the peripheral is accessible with this master.

RDC_SEMAPHORE_TryLock(), RDC_SEMAPHORE_Lock(), RDC_SEMAPHORE_Unlock() are the operations for the hardware semaphore.

If the hardware semaphore is locked, the user can use RDC_SEMAPHORE_GetLockDomainID() to get the domain ID who locks the semaphore and RDC_SEMAPHORE_GetLockMaster() to get the master index on the bus who locks the semaphore.

11.4.3 RDC SEMAPHORE reset control

In some use cases, the user might need to recover from error status and the hardware semaphore need to be reset to free status. Hereby RDC_SEMAPHORE_Reset() is introduced to reset single peripheral semaphore and RDC_SEMAPHORE_ResetAll() to reset all peripheral semaphores.

Enumerations

- enum rdc_semaphore_status_t {
  statusRdcSemaphoreSuccess = 0U,
  statusRdcSemaphoreBusy = 1U }

  RDC Semaphore status return codes.

RDC_SEMAPHORE State Control

- rdc_semaphore_status_t RDC_SEMAPHORE_TryLock (uint32_t pdap)

  Lock RDC semaphore for shared peripheral access.
- void RDC_SEMAPHORE_Lock (uint32_t pdap)

  Lock RDC semaphore for shared peripheral access, polling until success.
- void RDC_SEMAPHORE_Unlock (uint32_t pdap)

Unlock RDC semaphore.
- uint32_t RDC_SEMAPHORE_GetLockDomainID (uint32_t pdap)
  Get domain ID which locks the semaphore.
- uint32_t RDC_SEMAPHORE_GetLockMaster (uint32_t pdap)
  Get master index which locks the semaphore.

RDC_SEMAPHORE Reset Control

- void RDC_SEMAPHORE_Reset (uint32_t pdap)
  Reset RDC semaphore to unlocked status.
- void RDC_SEMAPHORE_ResetAll (RDC_SEMAPHORE_Type ∗base)
  Reset all RDC semaphore to unlocked status for certain RDC_SEMAPHORE instance.

11.4.4 Enumeration Type Documentation

11.4.4.1 enum rdc_semaphore_status_t

Enumerator

statusRdcSemaphoreSuccess	Success.
statusRdcSemaphoreBusy	RDC semaphore has been locked by other processor.

11.4.5 Function Documentation

11.4.5.1 rdc_semaphore_status_t RDC_SEMAPHORE_TryLock (uint32_t *pdap*)

Parameters

pdap	RDC peripheral assignment (see *rdc_pdap in rdc_defs*<device>.h)

Return values

statusRdcSemaphore-Success	Lock the semaphore successfully.
statusRdcSemaphoreBusy	Semaphore has been locked by other processor.

11.4.5.2 void RDC_SEMAPHORE_Lock (uint32_t *pdap*)

Parameters

pdap	RDC peripheral assignment (see *rdc_pdap in rdc_defs*<device>.h)

11.4.5.3 void RDC_SEMAPHORE_Unlock (uint32_t *pdap*)

Parameters

pdap	RDC peripheral assignment (see *rdc_pdap in rdc_defs*<device>.h)

11.4.5.4 uint32_t RDC_SEMAPHORE_GetLockDomainID (uint32_t *pdap*)

Parameters

pdap	RDC peripheral assignment (see *rdc_pdap in rdc_defs*<device>.h)

Returns

domain ID which locks the RDC semaphore

11.4.5.5 uint32_t RDC_SEMAPHORE_GetLockMaster (uint32_t *pdap*)

Parameters

pdap	RDC peripheral assignment (see *rdc_pdap in rdc_defs*<device>.h)

Returns

master index which locks the RDC semaphore, or RDC_SEMAPHORE_MASTER_NONE to indicate it is not locked.

11.4.5.6 void RDC_SEMAPHORE_Reset (uint32_t *pdap*)

Parameters

pdap	RDC peripheral assignment (see *rdc_pdap in rdc_defs*<device>.h)

11.4.5.7 void RDC_SEMAPHORE_ResetAll (RDC_SEMAPHORE_Type ∗ *base*)

Parameters

base	RDC semaphore base pointer.

Chapter 12
Hardware Semaphores (SEMA4)

12.1 Overview

The FreeRTOS BSP provides a driver for the hardware semaphore (SEMA4) block of i.MX devices.

Modules

- SEMA4 driver

12.2 SEMA4 driver

12.2.1 Overview

This chapter describes the programming interface of the SEMA4 driver (platform/drivers/inc/sema4.h).

SEMA4 driver provides three kinds of APIs:

- APIs to lock/unlock SEMA4 gate.
- APIs to reset SEMA4.
- APIs to control SEMA4 interrupt.

12.2.2 SEMA4 lock and unlock

To lock some SEMA4 gate, there are two functions to use. SEMA4_TryLock() is a non-block function, it only tries to lock the gate, if not locked, this function returns error. SEMA4_Lock() is a blocking function, it spins to lock the gate until it is locked.

SEMA4_Unlock() can be used to unlock the gate. To get the SEMA4 gate lock status, use the function SEMA4_GetLockProcessor(), which returns the processor number that locks the gate, or SEMA4_PROCESSOR_NONE if the SEMA4 is free to use.

12.2.3 SEMA4 reset

SEMA4 driver provides the functions to reset specific gate or all gates, the functions are SEMA4_ResetGate() and SEMA4_ResetAllGates(). To check the reset status or which bus master reset the SEMA4, use the function SEMA4_GetGateResetState() and SEMA4_GetGateResetBus().

SEMA driver can also reset specific gates or all gates' notifications, the functions are SEMA4_ResetNotification() and SEMA4_ResetAllNotifications(). To check the reset status or which bus master reset the SEMA4 notification, use the function SEMA4_GetNotificationResetState() and SEMA4_GetNotificationResetBus().

12.2.4 SEMA4 interrupt control

SEMA4 driver can also provider interrupt control to help the user to implement event driven hardware semaphore and free CPU from spinning, because when the semaphore is locked by the other processor, the user can get unlock interrupt if the gate's interrupt is enabled. SEMA4_SetIntCmd() is used to enable or disable specific gate's interrupt.

SEMA4_GetStatusFlag() and SEMA4_GetIntEnabled() can be used to get current interrupt status and check whether the specific gate's interrupt is enabled.

Enumerations

- enum _sema4_status_flag
  Status flag.
- enum _sema4_reset_state {
  sema4ResetIdle = 0U,
  sema4ResetMid = 1U,
  sema4ResetFinished = 2U }
  SEMA4 reset finite state machine.
- enum sema4_status_t {
  statusSema4Success = 0U,
  statusSema4Busy = 1U }
  SEMA4 status return codes.

SEMA4 State Control

- sema4_status_t SEMA4_TryLock (SEMA4_Type ∗base, uint32_t gateIndex)
  Lock SEMA4 gate for exclusive access between multicore.
- void SEMA4_Lock (SEMA4_Type ∗base, uint32_t gateIndex)
  Lock SEMA4 gate for exclusive access between multicore, polling until success.
- void SEMA4_Unlock (SEMA4_Type ∗base, uint32_t gateIndex)
  Unlock SEMA4 gate.
- uint32_t SEMA4_GetLockProcessor (SEMA4_Type ∗base, uint32_t gateIndex)
  Get processor number which locks the SEMA4 gate.

SEMA4 Reset Control

- void SEMA4_ResetGate (SEMA4_Type ∗base, uint32_t gateIndex)
  Reset SEMA4 gate to unlocked status.
- void SEMA4_ResetAllGates (SEMA4_Type ∗base)
  Reset all SEMA4 gates to unlocked status.
- static uint8_t SEMA4_GetGateResetBus (SEMA4_Type ∗base)
  Get bus master number which performing the gate reset function.
- static uint8_t SEMA4_GetGateResetState (SEMA4_Type ∗base)
  Get sema4 gate reset state.
- void SEMA4_ResetNotification (SEMA4_Type ∗base, uint32_t gateIndex)
  Reset SEMA4 IRQ notification.
- void SEMA4_ResetAllNotifications (SEMA4_Type ∗base)
  Reset all IRQ notifications.
- static uint8_t SEMA4_GetNotificationResetBus (SEMA4_Type ∗base)
  Get bus master number which performing the notification reset function.
- static uint8_t SEMA4_GetNotificationResetState (SEMA4_Type ∗base)
  Get sema4 notification reset state.

SEMA4 Interrupt and Status Control

- static uint16_t SEMA4_GetStatusFlag (SEMA4_Type ∗base, uint16_t flags)
  Get SEMA4 notification status.
- void SEMA4_SetIntCmd (SEMA4_Type ∗base, uint16_t intMask, bool enable)
  Enable or disable SEMA4 IRQ notification.
- static uint16_t SEMA4_GetIntEnabled (SEMA4_Type ∗base, uint16_t flags)
  check whether SEMA4 IRQ notification enabled.

12.2.5 Enumeration Type Documentation

12.2.5.1 enum _sema4_reset_state

Enumerator

sema4ResetIdle	Idle, waiting for the first data pattern write.
sema4ResetMid	Waiting for the second data pattern write.
sema4ResetFinished	Reset completed. Software can't get this state.

12.2.5.2 enum sema4_status_t

Enumerator

statusSema4Success	Success.
statusSema4Busy	SEMA4 gate has been locked by other processor.

12.2.6 Function Documentation

12.2.6.1 sema4_status_t SEMA4_TryLock (SEMA4_Type ∗ *base,* uint32_t *gateIndex*)

Parameters

base	SEMA4 base address
gateIndex	SEMA4 gate index

Return values

statusSema4Success	Lock the gate successfully.

statusSema4Busy	SEMA4 gate has been locked by other processor.

12.2.6.2 void SEMA4_Lock (SEMA4_Type ∗ *base,* uint32_t *gateIndex*)

Parameters

base	SEMA4 base address
gateIndex	SEMA4 gate index

12.2.6.3 void SEMA4_Unlock (SEMA4_Type ∗ *base,* uint32_t *gateIndex*)

Parameters

base	SEMA4 base pointer.
gateIndex	SEMA4 gate index

12.2.6.4 uint32_t SEMA4_GetLockProcessor (SEMA4_Type ∗ *base,* uint32_t *gateIndex*)

Parameters

base	SEMA4 base pointer.
gateIndex	SEMA4 gate index

Returns

processor number which locks the SEMA4 gate, or SEMA4_PROCESSOR_NONE to indicate the gate is not locked.

12.2.6.5 void SEMA4_ResetGate (SEMA4_Type ∗ *base,* uint32_t *gateIndex*)

Parameters

base	SEMA4 base pointer.
gateIndex	SEMA4 gate index

12.2.6.6 void SEMA4_ResetAllGates (SEMA4_Type ∗ *base*)

Parameters

base	SEMA4 base pointer.

12.2.6.7 static uint8_t SEMA4_GetGateResetBus (SEMA4_Type ∗ *base*) [inline], [static]

This function gets the bus master number which performing the gate reset function.

Parameters

base	SEMA4 base pointer.

Returns

Bus master number.

12.2.6.8 static uint8_t SEMA4_GetGateResetState (SEMA4_Type ∗ *base*) [inline], [static]

This function gets current state of the sema4 reset gate finite state machine.

Parameters

base	SEMA4 base pointer.

Returns

Current state, see _sema4_reset_state.

12.2.6.9 void SEMA4_ResetNotification (SEMA4_Type ∗ *base,* uint32_t *gateIndex*)

Parameters

base	SEMA4 base pointer.
gateIndex	SEMA4 gate index

12.2.6.10 void SEMA4_ResetAllNotifications (SEMA4_Type ∗ *base*)

Parameters

base	SEMA4 base pointer.

12.2.6.11 static uint8_t SEMA4_GetNotificationResetBus (SEMA4_Type ∗ *base*) `[inline]`, `[static]`

This function gets the bus master number which performing the notification reset function.

Parameters

base	SEMA4 base pointer.

Returns

Bus master number.

12.2.6.12 static uint8_t SEMA4_GetNotificationResetState (SEMA4_Type ∗ *base*) `[inline]`, `[static]`

This function gets current state of the sema4 reset notification finite state machine.

Parameters

base	SEMA4 base pointer.

Returns

Current state, see _sema4_reset_state.

12.2.6.13 static uint16_t SEMA4_GetStatusFlag (SEMA4_Type ∗ *base,* uint16_t *flags*) `[inline]`, `[static]`

Parameters

base	SEMA4 base pointer.
flags	SEMA4 gate status mask (see _sema4_status_flag)

Returns

SEMA4 notification status bits. If bit value is set, the corresponding gate's notification is available.

12.2.6.14 void SEMA4_SetIntCmd (SEMA4_Type ∗ *base,* uint16_t *intMask,* bool *enable*)

Parameters

base	SEMA4 base pointer.
intMask	SEMA4 gate status mask (see _sema4_status_flag)
enable	Interrupt enable (true: enable, false: disable), only those gates whose intMask is set are affected.

12.2.6.15 static uint16_t SEMA4_GetIntEnabled (SEMA4_Type ∗ *base,* uint16_t *flags*) `[inline]`, `[static]`

Parameters

base	SEMA4 base pointer.
flags	SEMA4 gate status mask (see _sema4_status_flag)

Returns

SEMA4 notification interrupt enable status bits. If bit value is set, the corresponding gate's notification is enabled

Chapter 13
Universal Asynchronous Receiver/Transmitter (UART)

13.1 Overview

The FreeRTOS BSP provides a driver for the Universal Asynchronous Receiver/Transmitter (UART) block of i.MX devices.

Modules

- UART driver

13.2 UART driver

13.2.1 Overview

The section describes the programming interface of the UART driver (platform/drivers/inc/uart_imx.h).

13.2.2 UART Initialization

To initialize the UART module, define an uart_init_config_t type variable and pass it to the UART_Init() function. Here is the Members of the structure definition:

1. clockRate: Current UART module clock frequency. This variable can be obtained by calling get_-uart_clock_freq() function;
2. baudRate: Desired UART baud rate. If the desired baud rate exceed UART module's limitation, the most nearest legal value is chosen;
3. wordLength: Data bits in one frame;
4. stopBitNum: Number of stop bits in one frame;
5. parity: Parity error check mode of this module;
6. direction: Data transfer direction of this module. this field is used to select the transfer direction. Choose the direction you used only can save system' s power.

User should also call UART_SetTxFifoWatermark() and UART_SetRxFifoWatermark() to set the watermark of TX/RX FIFO. After that, user can call UART_Enable() to enable UART module and transfer data through UART port.

13.2.3 FlexCAN Data Transactions

UART driver provides these APIs for data transactions:

```
UART_Putchar()
UART_Getchar()
```

UART Data Send

To send data through UART port, user should follow these steps:

1. Calling UART_GetStatusFlag() to check if UART Tx FIFO has available space;
2. If Tx FIFO is not full, call UART_Putchar() to add data to Tx FIFO;
3. Calling UART_GetStatusFlag() to check if UART Transmit is finished;
4. Repeat the above process to send more data.

UART Data Receive

To receive data through UART bus, user should follow these steps:

1. Calling UART_GetStatusFlag() to check if UART Rx FIFO has received data;
2. If Rx FIFO is not empty, call UART_Getchar() to read data from Rx FIFO;
3. Repeat the above process to read more data until Rx FIFO empty.

UART Status and Interrupt

This driver also provide APIs to handle UART module Status and Interrupt:

1. Calling UART_SetIntCmd() to enable/disable UART module interrupt;
2. Calling UART_GetStatusFlag() to get the UART status flags(described in enum _uart_status_flag) condition;
3. Calling UART_ClearStatusFlag() to clear specified status flags.

Specific UART functions

Besides the functions mentioned above, the UART driver also provide a set of functions for special purpose, like Auto Baud Detection, RS-485 multidrop communication and IrDA compatible low-speed optical communication. For more information about how to use these driver please refer to Chip Reference Manual and the function description below.

Example

For more information about how to use this driver, please refer to UART demo/example under examples/<board_name>/.

Data Structures

- struct uart_init_config_t
  Uart module initialize structure. More...

Enumerations

- enum _uart_word_length
  UART number of data bits in a character.
- enum _uart_stop_bit_num
  UART number of stop bits.
- enum _uart_partity_mode
  UART parity mode.
- enum _uart_direction_mode
  Data transfer direction.
- enum _uart_interrupt
  This enumeration contains the settings for all of the UART interrupt configurations.
- enum _uart_status_flag
  Flag for UART interrupt/DMA status check or polling status.

- enum _uart_dma
  The events generate the DMA Request.
- enum _uart_rts_int_trigger_edge
  RTS pin interrupt trigger edge.
- enum _uart_modem_mode
  UART module modem role selections.
- enum _uart_dtr_int_trigger_edge
  DTR pin interrupt trigger edge.
- enum _uart_irda_vote_clock
  IrDA vote clock selections.
- enum _uart_rx_idle_condition
  UART module Rx Idle condition selections.

UART Initialization and Configuration functions

- void UART_Init (UART_Type ∗base, uart_init_config_t ∗initConfig)
  Initialize UART module with given initialize structure.
- void UART_Deinit (UART_Type ∗base)
  This function reset UART module register content to its default value.
- static void UART_Enable (UART_Type ∗base)
  This function is used to Enable the UART Module.
- static void UART_Disable (UART_Type ∗base)
  This function is used to Disable the UART Module.
- void UART_SetBaudRate (UART_Type ∗base, uint32_t clockRate, uint32_t baudRate)
  This function is used to set the baud rate of UART Module.
- static void UART_SetDirMode (UART_Type ∗base, uint32_t direction)
  This function is used to set the transform direction of UART Module.
- static void UART_SetRxIdleCondition (UART_Type ∗base, uint32_t idleCondition)
  This function is used to set the number of frames RXD is allowed to be idle before an idle condition is reported.
- void UART_SetInvertCmd (UART_Type ∗base, uint32_t direction, bool invert)
  This function is used to set the polarity of UART signal.

Low Power Mode functions.

- void UART_SetDozeMode (UART_Type ∗base, bool enable)
  This function is used to set UART enable condition in the DOZE state.
- void UART_SetLowPowerMode (UART_Type ∗base, bool enable)
  This function is used to set UART enable condition of the UART low power feature.

Data transfer functions.

- static void UART_Putchar (UART_Type ∗base, uint8_t data)
  This function is used to send data in RS-232 and IrDA Mode.
- static uint8_t UART_Getchar (UART_Type ∗base)
  This function is used to receive data in RS-232 and IrDA Mode.

Interrupt and Flag control functions.

- void UART_SetIntCmd (UART_Type ∗base, uint32_t intSource, bool enable)
  This function is used to set the enable condition of specific UART interrupt source.
- bool UART_GetStatusFlag (UART_Type ∗base, uint32_t flag)
  This function is used to get the current status of specific UART status flag(including interrupt flag).
- void UART_ClearStatusFlag (UART_Type ∗base, uint32_t flag)
  This function is used to get the current status of specific UART status flag.

DMA control functions.

- void UART_SetDmaCmd (UART_Type ∗base, uint32_t dmaSource, bool enable)
  This function is used to set the enable condition of specific UART DMA source.

FIFO control functions.

- static void UART_SetTxFifoWatermark (UART_Type ∗base, uint8_t watermark)
  This function is used to set the watermark of UART Tx FIFO.
- static void UART_SetRxFifoWatermark (UART_Type ∗base, uint8_t watermark)
  This function is used to set the watermark of UART Rx FIFO.

Hardware Flow control and Modem Signal functions.

- void UART_SetRtsFlowCtrlCmd (UART_Type ∗base, bool enable)
  This function is used to set the enable condition of RTS Hardware flow control.
- static void UART_SetRtsIntTriggerEdge (UART_Type ∗base, uint32_t triggerEdge)
  This function is used to set the RTS interrupt trigger edge.
- void UART_SetCtsFlowCtrlCmd (UART_Type ∗base, bool enable)
  This function is used to set the enable condition of CTS auto control.
- void UART_SetCtsPinLevel (UART_Type ∗base, bool active)
  This function is used to control the CTS_B pin state when auto CTS control is disabled.
- static void UART_SetCtsTriggerLevel (UART_Type ∗base, uint8_t triggerLevel)
  This function is used to set the auto CTS_B pin control trigger level.
- void UART_SetModemMode (UART_Type ∗base, uint32_t mode)
  This function is used to set the role(DTE/DCE) of UART module in RS-232 communication.
- static void UART_SetDtrIntTriggerEdge (UART_Type ∗base, uint32_t triggerEdge)
  This function is used to set the edge of DTR_B (DCE) or DSR_B (DTE) on which an interrupt is generated.
- void UART_SetDtrPinLevel (UART_Type ∗base, bool active)
  This function is used to set the pin state of DSR pin(for DCE mode) or DTR pin(for DTE mode) for the modem interface.
- void UART_SetDcdPinLevel (UART_Type ∗base, bool active)
  This function is used to set the pin state of DCD pin.
- void UART_SetRiPinLevel (UART_Type ∗base, bool active)
  This function is used to set the pin state of RI pin.

Multi-processor and RS-485 functions.

- void UAER_Putchar9 (UART_Type ∗base, uint16_t data)
  This function is used to send 9 Bits length data in RS-485 Multidrop mode.
- uint16_t UAER_Getchar9 (UART_Type ∗base)
  This functions is used to receive 9 Bits length data in RS-485 Multidrop mode.
- void UART_SetMultidropMode (UART_Type ∗base, bool enable)
  This function is used to set the enable condition of 9-Bits data or Multidrop mode.
- void UART_SetSlaveAddressDetectCmd (UART_Type ∗base, bool enable)
  This function is used to set the enable condition of Automatic Address Detect Mode.
- static void UART_SetSlaveAddress (UART_Type ∗base, uint8_t slaveAddress)
  This function is used to set the slave address char that the receiver tries to detect.

IrDA control functions.

- void UART_SetIrDACmd (UART_Type ∗base, bool enable)
  This function is used to set the enable condition of IrDA Mode.
- void UART_SetIrDAVoteClock (UART_Type ∗base, uint32_t voteClock)
  This function is used to set the clock for the IR pulsed vote logic.

Misc. functions.

- void UART_SetAutoBaudRateCmd (UART_Type ∗base, bool enable)
  This function is used to set the enable condition of Automatic Baud Rate Detection feature.
- static uint16_t UART_ReadBaudRateCount (UART_Type ∗base)
  This function is used to read the current value of Baud Rate Count Register value.
- void UART_SendBreakChar (UART_Type ∗base, bool active)
  This function is used to send BREAK character.It is important that SNDBRK is asserted high for a sufficient period of time to generate a valid BREAK.
- void UART_SetEscapeDecectCmd (UART_Type ∗base, bool enable)
  This function is used to send BREAK character.It is important that SNDBRK is asserted high for a sufficient period of time to generate a valid BREAK.
- static void UART_SetEscapeChar (UART_Type ∗base, uint8_t escapeChar)
  This function is used to set the enable condition of Escape Sequence Detection feature.
- static void UART_SetEscapeTimerInterval (UART_Type ∗base, uint16_t timerInterval)
  This function is used to set the maximum time interval (in ms) allowed between escape characters.

13.2.4 Data Structure Documentation

13.2.4.1 struct uart_init_config_t

Data Fields

- uint32_t clockRate
  Current UART module clock freq.
- uint32_t baudRate

Desired UART baud rate.
- uint32_t wordLength
  Data bits in one frame.
- uint32_t stopBitNum
  Number of stop bits in one frame.
- uint32_t parity
  Parity error check mode of this module.
- uint32_t direction
  Data transfer direction of this module.

13.2.4.1.0.6 Field Documentation

13.2.4.1.0.6.1 uint32_t uart_init_config_t::clockRate

13.2.4.1.0.6.2 uint32_t uart_init_config_t::baudRate

13.2.4.1.0.6.3 uint32_t uart_init_config_t::wordLength

13.2.4.1.0.6.4 uint32_t uart_init_config_t::stopBitNum

13.2.4.1.0.6.5 uint32_t uart_init_config_t::parity

13.2.4.1.0.6.6 uint32_t uart_init_config_t::direction

13.2.5 Function Documentation

13.2.5.1 void UART_Init (UART_Type ∗ *base*, uart_init_config_t ∗ *initConfig*)

Parameters

base	UART base pointer.
initConfig	UART initialize structure (see uart_init_config_t above).

13.2.5.2 void UART_Deinit (UART_Type ∗ *base*)

Parameters

base	UART base pointer.

13.2.5.3 static void UART_Enable (UART_Type ∗ *base*) `[inline]`, `[static]`

Parameters

base	UART base pointer.

13.2.5.4 static void UART_Disable (UART_Type ∗ *base*) [inline], [static]

Parameters

base	UART base pointer.

13.2.5.5 void UART_SetBaudRate (UART_Type ∗ *base,* uint32_t *clockRate,* uint32_t *baudRate*)

Parameters

base	UART base pointer.
clockRate	UART module clock frequency.
baudRate	Desired UART module baud rate.

13.2.5.6 static void UART_SetDirMode (UART_Type ∗ *base,* uint32_t *direction*) [inline], [static]

Parameters

base	UART base pointer.
direction	UART transfer direction (see _uart_direction_mode enumeration above).

13.2.5.7 static void UART_SetRxIdleCondition (UART_Type ∗ *base,* uint32_t *idleCondition*) [inline], [static]

The available condition can be select from _uart_idle_condition enumeration.

Parameters

base	UART base pointer.
idleCondition	The condition that an idle condition is reported (see _uart_idle_condition enumeration above).

13.2.5.8 void UART_SetInvertCmd (UART_Type ∗ *base,* uint32_t *direction,* bool *invert*)

The polarity of Tx and Rx can be set separately.

Parameters

base	UART base pointer.
direction	UART transfer direction (see _uart_direction_mode enumeration above).
invert	Set true to invert the polarity of UART signal.

13.2.5.9 void UART_SetDozeMode (UART_Type ∗ *base,* bool *enable*)

Parameters

base	UART base pointer.
enable	Set true to enable UART module in doze mode.

13.2.5.10 void UART_SetLowPowerMode (UART_Type ∗ *base,* bool *enable*)

Parameters

base	UART base pointer.
enable	Set true to enable UART module low power feature.

13.2.5.11 static void UART_Putchar (UART_Type ∗ *base,* uint8_t *data*) `[inline], [static]`

```
A independent 9 Bits RS-485 send data function is provided.
```

Parameters

base	UART base pointer.
data	Data to be set through UART module.

13.2.5.12 static uint8_t UART_Getchar (UART_Type ∗ *base*) [inline], [static]

```
A independent 9 Bits RS-485 receive data function is provided.
```

Parameters

base	UART base pointer.

Returns

The data received from UART module.

13.2.5.13 void UART_SetIntCmd (UART_Type ∗ *base,* uint32_t *intSource,* bool *enable*)

The available interrupt source can be select from uart_interrupt enumeration.

Parameters

base	UART base pointer.
intSource	Available interrupt source for this module.
enable	Set true to enable corresponding interrupt.

13.2.5.14 bool UART_GetStatusFlag (UART_Type ∗ *base,* uint32_t *flag*)

The available status flag can be select from _uart_status_flag enumeration.

Parameters

base	UART base pointer.
flag	Status flag to check.

Returns

current state of corresponding status flag.

13.2.5.15 void UART_ClearStatusFlag (UART_Type ∗ *base,* uint32_t *flag*)

The available status flag can be select from _uart_status_flag enumeration.

Parameters

base	UART base pointer.
flag	Status flag to clear.

13.2.5.16 void UART_SetDmaCmd (UART_Type ∗ *base,* uint32_t *dmaSource,* bool *enable*)

The available DMA source can be select from _uart_dma enumeration.

Parameters

base	UART base pointer.
dmaSource	The Event that can generate DMA request.
enable	Set true to enable corresponding DMA source.

13.2.5.17 static void UART_SetTxFifoWatermark (UART_Type ∗ *base,* uint8_t *watermark*) [inline], [static]

```
A maskable interrupt is generated whenever the data level in
the TxFIFO falls below the Tx FIFO watermark.
```

Parameters

base	UART base pointer.
watermark	The Tx FIFO watermark.

13.2.5.18 static void UART_SetRxFifoWatermark (UART_Type ∗ *base,* uint8_t *watermark*) [inline], [static]

```
A maskable interrupt is generated whenever the data level in
the RxFIFO reaches the Rx FIFO watermark.
```

Parameters

base	UART base pointer.
watermark	The Rx FIFO watermark.

13.2.5.19 void UART_SetRtsFlowCtrlCmd (UART_Type ∗ *base,* bool *enable*)

Parameters

base	UART base pointer.
enable	Set true to enable RTS hardware flow control.

13.2.5.20 static void UART_SetRtsIntTriggerEdge (UART_Type ∗ *base,* uint32_t *triggerEdge*) `[inline]`, `[static]`

```
The available trigger edge can be select from
_uart_rts_trigger_edge enumeration.
```

Parameters

base	UART base pointer.
triggerEdge	Available RTS pin interrupt trigger edge.

13.2.5.21 void UART_SetCtsFlowCtrlCmd (UART_Type ∗ *base,* bool *enable*)

if CTS control is enabled, the CTS_B pin is controlled by the receiver, otherwise the CTS_B pin is controlled by UART_CTSPinCtrl function.

Parameters

base	UART base pointer.
enable	Set true to enable CTS auto control.

13.2.5.22 void UART_SetCtsPinLevel (UART_Type ∗ *base,* bool *active*)

```
The CTS_B pin is low(active)
The CTS_B pin is high(inactive)
```

Parameters

base	UART base pointer.
active	Set true: the CTS_B pin active; Set false: the CTS_B pin inactive.

13.2.5.23 static void UART_SetCtsTriggerLevel (UART_Type ∗ *base,* uint8_t *triggerLevel*) [inline], [static]

The CTS_B pin is de-asserted when Rx FIFO reach CTS trigger level.

Parameters

base	UART base pointer.
triggerLevel	Auto CTS_B pin control trigger level.

13.2.5.24 void UART_SetModemMode (UART_Type ∗ *base,* uint32_t *mode*)

Parameters

base	UART base pointer.
mode	The role(DTE/DCE) of UART module (see _uart_modem_mode enumeration above).

13.2.5.25 static void UART_SetDtrIntTriggerEdge (UART_Type ∗ *base,* uint32_t *triggerEdge*) [inline], [static]

Parameters

base	UART base pointer.
triggerEdge	The trigger edge on which an interrupt is generated. (see _uart_dtr_trigger_edge enumeration above)

13.2.5.26 void UART_SetDtrPinLevel (UART_Type ∗ *base,* bool *active*)

Parameters

base	UART base pointer.
active	Set true: DSR/DTR pin is logic one. Set false: DSR/DTR pin is logic zero.

13.2.5.27 void UART_SetDcdPinLevel (UART_Type ∗ *base,* bool *active*)

THIS FUNCTION IS FOR DCE MODE ONLY.

Parameters

base	UART base pointer.
active	Set true: DCD_B pin is logic one (DCE mode) Set false: DCD_B pin is logic zero (DCE mode)

13.2.5.28 void UART_SetRiPinLevel (UART_Type ∗ *base,* bool *active*)

THIS FUNCTION IS FOR DCE MODE ONLY.

Parameters

base	UART base pointer.
active	Set true: RI_B pin is logic one (DCE mode) Set false: RI_B pin is logic zero (DCE mode)

13.2.5.29 void UAER_Putchar9 (UART_Type ∗ *base,* uint16_t *data*)

Parameters

base	UART base pointer.
data	Data(9 bits) to be set through UART module.

13.2.5.30 uint16_t UAER_Getchar9 (UART_Type ∗ *base*)

Parameters

base	UART base pointer.

Returns

The data(9 bits) received from UART module.

13.2.5.31 void UART_SetMultidropMode (UART_Type ∗ *base*, bool *enable*)

Parameters

base	UART base pointer.
enable	Set true to enable Multidrop mode.

13.2.5.32 void UART_SetSlaveAddressDetectCmd (UART_Type ∗ *base*, bool *enable*)

Parameters

base	UART base pointer.
enable	Set true to enable Automatic Address Detect mode.

13.2.5.33 static void UART_SetSlaveAddress (UART_Type ∗ *base*, uint8_t *slaveAddress*) `[inline]`, `[static]`

Parameters

base	UART base pointer.
slaveAddress	The slave to detect.

13.2.5.34 void UART_SetIrDACmd (UART_Type ∗ *base*, bool *enable*)

Parameters

base	UART base pointer.
enable	Set true to enable IrDA mode.

13.2.5.35 void UART_SetIrDAVoteClock (UART_Type ∗ *base*, uint32_t *voteClock*)

The available clock can be select from _uart_irda_vote_clock enumeration.

Parameters

base	UART base pointer.
voteClock	The available IrDA vote clock selection.

13.2.5.36 void UART_SetAutoBaudRateCmd (UART_Type * *base,* bool *enable*)

Parameters

base	UART base pointer.
enable	Set true to enable Automatic Baud Rate Detection feature.

13.2.5.37 static uint16_t UART_ReadBaudRateCount (UART_Type * *base*) [inline], [static]

this counter is used by Auto Baud Rate Detect feature.

Parameters

base	UART base pointer.

Returns

Current Baud Rate Count Register value.

13.2.5.38 void UART_SendBreakChar (UART_Type * *base,* bool *active*)

Parameters

base	UART base pointer.
active	Asserted high to generate BREAK.

13.2.5.39 void UART_SetEscapeDecectCmd (UART_Type * *base,* bool *enable*)

Parameters

base	UART base pointer.
active	Asserted high to generate BREAK.

13.2.5.40 static void UART_SetEscapeChar (UART_Type * *base,* uint8_t *escapeChar*) `[inline]`, `[static]`

Parameters

base	UART base pointer.
escapeChar	The Escape Character to detect.

13.2.5.41 static void UART_SetEscapeTimerInterval (UART_Type * *base,* uint16_t *timerInterval*) `[inline]`, `[static]`

Parameters

base	UART base pointer.
timerInterval	Maximum time interval allowed between escape characters.

Chapter 14
Watchdog Timer (WDOG)

14.1 Overview

The FreeRTOS BSP provides a driver for the Watchdog Timer (WDOG) block of i.MX devices.

Modules

- WDOG driver on i.MX

14.2 WDOG driver on i.MX

14.2.1 Overview

The chapter describes the programming interface of the WDOG driver on i.MX (platform/drivers/inc/wdog-_imx.h). The i.MX watchdog protects against system failures by providing a method by which to escape from unexpected events or programming errors. The WDOG driver on i.MX provides a set of APIs to provide these services:

- Watchdog general control;
- Watchdog interrupt control;

14.2.2 Watchdog general control

After reset, WDOG_DisablePowerdown() must be called to avoid the power down timeout. It's a one-shot function and cannot be called more than once.

Before enabling the watchdog, WDOG_Init() is needed to initialize the watchdog driver and specify the watchdog behavior in different modes. This function is also a one-shot function.

Then WDOG_Enable() can be used to enable the watchdog with specified timeout, and when timeout, CPU is reset and external pin WDOG_B might be asserted based on the behavior setting. Once enabled, the watchdog cannot be disabled so it's also a one-shot function.

To avoid timeout, the program must WDOG_Refresh() the counter periodically.

The user can also use WDOG_Reset() to reset the CPU and assert some reset signal specified by parameters immediately.

14.2.3 Watchdog interrupt control

i.MX watchdog also provides interrupt before timeout reset occurs. The user can use WDOG_EnableInt() with proper time to make sure the interrupt would happen some time before timeout reset.

When the interrupt occurs, WDOG_ClearStatusFlag() is used to clear the status. WDOG_IsIntPending() can be used to check whether there's any interrupt pending.

Data Structures

- struct wdog_mode_config_t
  Structure to configure the running mode. More...

WDOG State Control

- static void WDOG_Init (WDOG_Type ∗base, wdog_mode_config_t ∗config)

Configure WDOG functions, call once only.
- void WDOG_Enable (WDOG_Type ∗base, uint8_t timeout)
    Enable WDOG with timeout, call once only.
- void WDOG_Reset (WDOG_Type ∗base, bool wda, bool srs)
    Assert WDOG software reset signal.
- void WDOG_Refresh (WDOG_Type ∗base)
    Refresh the WDOG to prevent timeout.
- static void WDOG_DisablePowerdown (WDOG_Type ∗base)
    Disable WDOG power down counter.

WDOG Interrupt Control

- static void WDOG_EnableInt (WDOG_Type ∗base, uint8_t time)
    Enable WDOG interrupt.
- static bool WDOG_IsIntPending (WDOG_Type ∗base)
    Check whether WDOG interrupt is pending.
- static void WDOG_ClearStatusFlag (WDOG_Type ∗base)
    Clear WDOG interrupt status.

14.2.4 Data Structure Documentation

14.2.4.1 struct wdog_mode_config_t

Data Fields

- bool wdw
    true: suspend in low power wait, false: not suspend
- bool wdt
    true: assert WDOG_B when timeout, false: not assert WDOG_B
- bool wdbg
    true: suspend in debug mode, false: not suspend
- bool wdzst
    true: suspend in doze and stop mode, false: not suspend

14.2.5 Function Documentation

14.2.5.1 static void WDOG_Init (WDOG_Type ∗ *base*, wdog_mode_config_t ∗ *config*) `[inline]`, `[static]`

Parameters

base	WDOG base pointer.
config	WDOG mode configuration

14.2.5.2 void WDOG_Enable (WDOG_Type ∗ *base,* uint8_t *timeout*)

Parameters

base	WDOG base pointer.
timeout	WDOG timeout ((n+1)/2 second)

14.2.5.3 void WDOG_Reset (WDOG_Type ∗ *base,* bool *wda,* bool *srs*)

Parameters

base	WDOG base pointer.
wda	WDOG reset (true: assert WDOG_B, false: no impact on WDOG_B)
srs	System reset (true: assert system reset WDOG_RESET_B_DEB, false: no impact on system reset)

14.2.5.4 void WDOG_Refresh (WDOG_Type ∗ *base*)

Parameters

base	WDOG base pointer.

14.2.5.5 static void WDOG_DisablePowerdown (WDOG_Type ∗ *base*) `[inline]`, `[static]`

Parameters

base	WDOG base pointer.

14.2.5.6 static void WDOG_EnableInt (WDOG_Type ∗ *base,* uint8_t *time*) `[inline]`, `[static]`

Parameters

base	WDOG base pointer.
time	how long before the timeout must the interrupt occur (n/2 seconds).

14.2.5.7 static bool WDOG_IsIntPending (WDOG_Type ∗ *base*) `[inline]`, `[static]`

Parameters

base	WDOG base pointer.

Returns

WDOG interrupt status (true: pending, false: not pending)

14.2.5.8 static void WDOG_ClearStatusFlag (WDOG_Type ∗ *base*) `[inline]`, `[static]`

Parameters

base	WDOG base pointer.

Chapter 15
Utilities for the FreeRTOS BSP

15.1 Overview

The FreeRTOS BSP provides a set of utilities to help user with their development.

Modules

- Debug Console

15.2 Debug Console

15.2.1 Overview

This section describes the programming interface of the debug console driver.

15.2.2 Debug Console Initialization

To initialize the DbgConsole module, call the DbgConsole_Init() function and pass in the parameters needed by this function. This function automatically enables the module and clock. After the DbgConsole_Init() function is called and returned, stdout and stdin are connected to the selected UART.

The parameters needed by this function are shown here:

```
1. UART_Type* base      : The base address of the UART module used as debug console;
2. uint32_t   clockRate : The clock source frequency of UART module, this value can be obtained by calling
      get_uart_clock_freq() function;
3. uint32_t   baudRate  : The desired baud rate frequency.
```

Debug console state is stored in debug_console_state_t structure:

```
typedef struct DebugConsoleState {
    bool  inited;                    /*<! Identify debug console initialized or not. */
    void* base;                      /*<! Base of the IP register. */
    debug_console_ops_t ops;         /*<! Operation function pointers for debug UART operations. */
} debug_console_state_t;
```

This example shows how to call the DbgConsole_Init() given the user configuration parameters.

```
DbgConsole_Init(BOARD_DEBUG_UART_BASEADDR, get_uart_clock_freq(BOARD_DEBUG_UART_BASEADDR), 1
      15200);
```

Debug Console formatted IO

Debug console has its own printf/scanf/putchar/getchar functions which are defined in the header:

```
int debug_printf(const char  *fmt_s, ...);
int debug_putchar(int ch);
int debug_scanf(const char  *fmt_ptr, ...);
int debug_getchar(void);
```

Choose toolchain's printf/scanf or FreeRTOS BSP version printf/scanf:

```
/*Configuration for toolchain's printf/scanf or FreeRTOS BSP version printf/scanf */
#define PRINTF          debug_printf
//#define PRINTF          printf
#define SCANF           debug_scanf
//#define SCANF           scanf
#define PUTCHAR         debug_putchar
//#define PUTCHAR         putchar
#define GETCHAR         debug_getchar
//#define GETCHAR         getchar
```

Function _doprint outputs its parameters according to a formatted string. I/O is performed by calling given function pointer using (∗func_ptr)(c,farg).

```
int _doprint(void *farg, PUTCHAR_FUNC func_ptr, int max_count, char *fmt, va_list ap)
```

Function scan_prv converts an input line of ASCII characters based upon a provided string format.

```
int scan_prv(const char *line_ptr, char *format, va_list args_ptr)
```

Function mknumstr converts a radix number to a string and return its length.

```
static int32_t mknumstr (char *numstr, void *nump, int32_t neg, int32_t radix, bool use_caps);
```

Function mkfloatnumstr converts a floating radix number to a string and return its length.

```
static int32_t mkfloatnumstr (char *numstr, void *nump, int32_t radix, uint32_t precision_width);
```

Macros

- #define PRINTF debug_printf
  Configuration for toolchain's printf/scanf or Freescale version printf/scanf.

Enumerations

- enum debug_console_status_t
  Error code for the debug console driver.

Initialization

- debug_console_status_t DbgConsole_Init (UART_Type ∗base, uint32_t clockRate, uint32_t baudRate)
  Initialize the UART_IMX used for debug messages.
- debug_console_status_t DbgConsole_DeInit (void)
  Deinitialize the UART/LPUART used for debug messages.
- int debug_printf (const char ∗fmt_s,...)
  Prints formatted output to the standard output stream.
- int debug_putchar (int ch)
  Writes a character to stdout.
- int debug_scanf (const char ∗fmt_ptr,...)
  Reads formatted data from the standard input stream.
- int debug_getchar (void)
  Reads a character from standard input.

15.2.3 Enumeration Type Documentation

15.2.3.1 enum debug_console_status_t

15.2.4 Function Documentation

15.2.4.1 debug_console_status_t DbgConsole_Init (UART_Type ∗ *base,* uint32_t *clockRate,* uint32_t *baudRate*)

Call this function to enable debug log messages to be output via the specified UART_IMX base address and at the specified baud rate. Just initializes the UART_IMX to the given baud rate and 8N1. After this function has returned, stdout and stdin are connected to the selected UART_IMX. The debug_printf() function also uses this UART_IMX.

Parameters

base	Which UART_IMX instance is used to send debug messages.
clockRate	The input clock of UART_IMX module.
baudRate	The desired baud rate in bits per second.

Returns

Whether initialization was successful or not.

15.2.4.2 debug_console_status_t DbgConsole_DeInit (void)

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

Returns

Whether de-initialization was successful or not.

15.2.4.3 int debug_printf (const char ∗ *fmt_s,* ...)

Call this function to print 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.

15.2.4.4 int debug_putchar (int *ch*)

Call this function to write a character to stdout.

Parameters

ch	Character to be written.

Returns

Returns the character written.

15.2.4.5 int debug_scanf (const char ∗ *fmt_ptr,* ...)

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

Parameters

fmt_ptr	Format control string.

Returns

Returns the number of fields successfully converted and assigned.

15.2.4.6 int debug_getchar (void)

Call this function to read a character from standard input.

Returns

Returns the character read.

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: FRTOS7DAPIRM
Rev. 0
08/2015