

**PRELIMINARY**

|  |  |
| --- | --- |
| MEC2016 | |
| Peripheral Interface | |
| Rev 23.0 | 01/19/2017 |

**MCHP CONFIDENTIAL**

|  |
| --- |
| Copyright © 2015 MICROCHIP TECHNOLOGY INC. or its subsidiaries. All rights reserved.  The information contained herein is confidential and proprietary to MICROCHIP TECHNOLOGY INC., shall be used solely in accordance with the agreement pursuant to which it is provided, and shall not be reproduced or disclosed to others without the prior written consent of MICROCHIP TECHNOLOGY INC.. Although the information is believed to be accurate, no responsibility is assumed for inaccuracies. MICROCHIP TECHNOLOGY INC. reserves the right to make changes to this document and to specifications and product descriptions at any time without notice. Neither the provision of this information nor the sale of the described semiconductor devices conveys any licenses under any patent rights or other intellectual property rights of MICROCHIP TECHNOLOGY INC. or others. The product may contain design defects or errors known as anomalies, including but not necessarily limited to any which may be identified in this document, which may cause the product to deviate from published specifications. MICROCHIP TECHNOLOGY INC. products are not designed, intended, authorized or warranted for use in any life support or other application where product failure could cause or contribute to personal injury or severe property damage. Any and all such uses without prior written approval of an officer of MICROCHIP TECHNOLOGY INC. will be fully at the risk of the customer..  MICROCHIP TECHNOLOGY INC. DISCLAIMS AND EXCLUDES ANY AND ALL WARRANTIES, INCLUDING WITHOUT LIMITATION ANY AND ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, AND AGAINST INFRINGEMENT AND THE LIKE, AND ANY AND ALL WARRANTIES ARISING FROM ANY COURSE OF DEALING OR USAGE OF TRADE. IN NO EVENT SHALL MICROCHIP TECHNOLOGY INC. BE LIABLE FOR ANY DIRECT, INCIDENTAL, INDIRECT, SPECIAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES; OR FOR LOST DATA, PROFITS, SAVINGS OR REVENUES OF ANY KIND; REGARDLESS OF THE FORM OF ACTION, WHETHER BASED ON CONTRACT; TORT; NEGLIGENCE OF MICROCHIP TECHNOLOGY INC. OR OTHERS; STRICT LIABILITY; BREACH OF WARRANTY; OR OTHERWISE; WHETHER OR NOT ANY REMEDY OF BUYER IS HELD TO HAVE FAILED OF ITS ESSENTIAL PURPOSE, AND WHETHER OR NOT MICROCHIP TECHNOLOGY INC. HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. |

**Table of Contents**

[1 INTRODUCTION 5](#_Toc472591926)

[2 Basic Timer 6](#_Toc472591927)

[2.1 Basic Timer APIs 7](#_Toc472591928)

[2.2 Basic Timer Peripheral Functions 17](#_Toc472591929)

[3 PWM 27](#_Toc472591930)

[3.1 PWM API’s 27](#_Toc472591931)

[3.2 PWM Peripheral Functions 29](#_Toc472591932)

[4 GPIO 34](#_Toc472591933)

[4.1 GPIO APIs 34](#_Toc472591934)

[4.2 GPIO Peripheral Functions 40](#_Toc472591935)

[5 SMBus Driver 47](#_Toc472591936)

[5.1 SMBus Driver APIs & Callbacks 47](#_Toc472591937)

[5.2 SMBus driver configuration 48](#_Toc472591938)

[5.3 Driver Set Up & Initialization 48](#_Toc472591939)

[5.4 Configuring SMBus Controller 51](#_Toc472591940)

[5.5 MASTER APIs 53](#_Toc472591941)

[5.6 SLAVE APIs 64](#_Toc472591942)

[5.7 Handling PEC 65](#_Toc472591943)

[5.8 buffer\_info details for smbus protocols 66](#_Toc472591944)

[6 ADC 69](#_Toc472591945)

[6.1 ADC API’s 69](#_Toc472591946)

[6.2 ADC Peripheral Functions 70](#_Toc472591947)

[7 PCR – Power, Clocks, Reset 77](#_Toc472591948)

[7.1 PCR APIs 77](#_Toc472591949)

[7.2 PCR Peripheral Functions 84](#_Toc472591950)

[8 TACH 98](#_Toc472591951)

[8.1 Tach API’s 98](#_Toc472591952)

[8.2 Tach Peripheral Functions 101](#_Toc472591953)

[9 LED 109](#_Toc472591954)

[9.1 LED APIs 109](#_Toc472591955)

[9.2 LED peripheral functions 112](#_Toc472591956)

[10 SPI 120](#_Toc472591957)

[10.1 SPI API’s 120](#_Toc472591958)

[10.2 Applications 125](#_Toc472591959)

[11 WDT 128](#_Toc472591960)

[11.1 WDT API’s 128](#_Toc472591961)

[11.2 WDT Peripheral Functions 131](#_Toc472591962)

[12 Interrupt 134](#_Toc472591963)

[12.1 Interrupt APIs 134](#_Toc472591964)

[12.2 Interrupt ECIA Peripheral Functions 143](#_Toc472591965)

[12.3 Interrupt NVIC Peripheral Functions 152](#_Toc472591966)

[13 Hibernation Timer 155](#_Toc472591967)

[13.1 Hibernation Timer APIs 155](#_Toc472591968)

[13.2 Hibernation Timer Peripheral functions 158](#_Toc472591969)

[14 RTC 161](#_Toc472591970)

[14.1 RTC Peripheral Functions 161](#_Toc472591971)

[14.2 RTC API’s 175](#_Toc472591972)

[15 UART 180](#_Toc472591973)

[15.1 UART APIs 180](#_Toc472591974)

[15.2 UART peripheral functions 183](#_Toc472591975)

[16 QMSPI Functions 192](#_Toc472591976)

[17 Input Capture and Compare Timer 198](#_Toc472591977)

[17.1 ICCT APIs 198](#_Toc472591978)

[17.2 ICCT Peripheral Functions 200](#_Toc472591979)

[18 PS/2 210](#_Toc472591980)

[18.1 PS/2 APIs 210](#_Toc472591981)

[18.2 PS/2 peripheral functions 212](#_Toc472591982)

[19 Keyboard Scan Matrix 215](#_Toc472591983)

[19.1 Keyboard Scan Matrix APIs 215](#_Toc472591984)

[19.2 Keyboard Scan Matrix peripheral functions 216](#_Toc472591985)

[20 Week Timer 219](#_Toc472591986)

[20.1 Week Timer APIs 219](#_Toc472591987)

[20.2 Week Timer Peripheral Functions 222](#_Toc472591988)

[21 EMI 232](#_Toc472591989)

[21.1 EMI block APIs 234](#_Toc472591990)

[21.2 EMI block peripheral functions 236](#_Toc472591991)

[22 MailBox 242](#_Toc472591992)

[22.1 Mailbox APIs 242](#_Toc472591993)

[22.2 Mailbox block peripheral functions 245](#_Toc472591994)

[23 BClink 248](#_Toc472591995)

[23.1 BClink APIs 248](#_Toc472591996)

[23.2 BClink Peripheral functions 252](#_Toc472591997)

[24 VBAT Register Bank 257](#_Toc472591998)

[24.1 VBAT Register Bank API Function 257](#_Toc472591999)

[24.2 VBAT Register Bank Peripheral Function 258](#_Toc472592000)

[25 RPM - PWM 262](#_Toc472592001)

[25.1 RPM - PWM API Functions 262](#_Toc472592002)

[25.2 RPM - PWM Peripheral Functions 263](#_Toc472592003)

[26 ACPI – ECI 270](#_Toc472592004)

[26.1 ACPI ECI Peripheral Functions 270](#_Toc472592005)

[27 8042 Keyboard Emulator 273](#_Toc472592006)

[273](#_Toc472592007)

[27.1 8042 Emulator Peripheral Functions 273](#_Toc472592008)

[28 Port 80 278](#_Toc472592009)

[28.1 Port 80 Peripheral Functions 278](#_Toc472592010)

[29 RTOS Timer 281](#_Toc472592011)

[29.1 RTOS Timer API Functions 281](#_Toc472592012)

[29.2 RTOS Timer Peripheral Functions 282](#_Toc472592013)

[30 EEPROM 286](#_Toc472592014)

[286](#_Toc472592015)

[30.1 EEPROM API Functions 286](#_Toc472592016)

[30.2 EEPROM Peripheral Functions 287](#_Toc472592017)

[31 eSPI 292](#_Toc472592018)

[31.1 eSPI Global API Functions 292](#_Toc472592019)

[31.2 eSPI Global Peripheral Functions 294](#_Toc472592020)

[31.3 eSPI PC API Functions 299](#_Toc472592021)

[31.4 eSPI PC Peripheral Functions 301](#_Toc472592022)

[31.5 eSPI OOB Channel API Functions 312](#_Toc472592023)

[31.6 eSPI OOB Channel Peripheral Functions 314](#_Toc472592024)

[31.7 eSPI Flash Channel API Functions 319](#_Toc472592025)

[31.8 eSPI Flash Channel Peripheral Functions 320](#_Toc472592026)

[324](#_Toc472592027)

[31.9 eSPI VWire Channel Peripheral Functions 324](#_Toc472592028)

[32 Revision History 329](#_Toc472592029)

# INTRODUCTION

The peripheral software interface is provided to communicate with MEC2016 peripherals. It is composed of two layers:

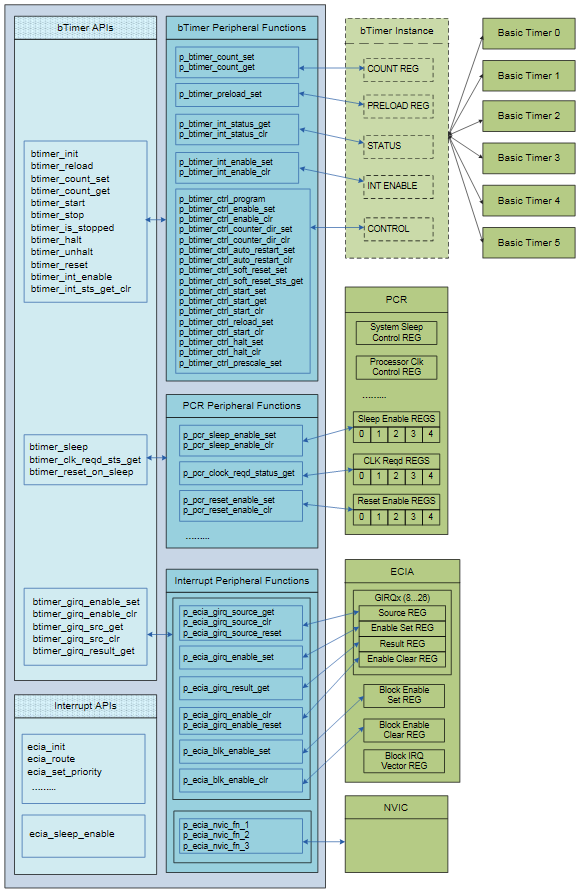
* APIs
* Peripheral functions

Peripheral functions provide a low level interface to the hardware block.

The APIs are built over the peripheral function. Applications are recommended to interface using the APIs. The APIs provides an interface to execute simple operations. The application or driver can use the APIs to perform a sequence of operations (through API call) to perform a task.

For some complex hardware peripheral, driver would be provided which the application can integrate into their kernel/RTOS.

# Basic Timer



## Basic Timer APIs

The list of Basic Timer APIs

1. Basic Timer Intitialization function

* btimer\_init

2. Functions to program and read the Basic Timer Counter

* btimer\_count\_set
* btimer\_count\_get

3. Function to reload counter from Preload Register

* btimer\_reload

4. Functions for stopping and starting the basic Timer

* btimer\_start
* btimer\_stop
* btimer\_is\_started

5. Function to perform basic timer soft reset

* btimer\_reset

6. Functions to halt/unhalt the timer counting

* btimer\_halt
* btimer\_unhalt

7. Functions for Basic Timer interrupt

* btimer\_interrupt\_enable
* btimer\_interrupt\_status\_get\_clr

8. Functions for Basic Timer GIRQ

* btimer\_girq\_enable\_set
* btimer\_girq\_enable\_clr
* btimer\_girq\_src\_get
* btimer\_girq\_src\_clr
* btimer\_girq\_result\_get

9. Functions for Basic Timer Sleep

* btimer\_sleep
* btimer\_clk\_reqd\_sts\_get
* btimer\_reset\_on\_sleep

### btimer\_init

Function Header

**void btimer\_init (uint8\_t btimer\_id,**

**uint16\_t tmr\_cntl,**

**uint16\_t prescaler,**

**uint32\_t initial\_count,**

**uint32\_t preload\_count)**

Description

Initialize specified timer

Note: This function performs a soft reset of the timer before configuration

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | timer ID – PID\_BTIMER\_x (x => 0-5) |
| tmr\_cntl | see tmr\_cntl parameters below |
| prescaler | Timer prescaler |
| **initial\_count** | initial count |
| **preload\_count** | preload count |

//

// Logical flags for tmr\_cntl parameter of btimer\_init

//

BTMR\_AUTO\_RESTART

BTMR\_ONE\_SHOT

BTMR\_COUNT\_UP

BTMR\_COUNT\_DOWN

BTMR\_INT\_EN

BTMR\_NO\_INT

Outputs

None

Example Usage

btimer\_init ( PID\_BTIMER\_0,

BTMR\_AUTO\_RESTART + BTMR\_COUNT\_DOWN + BTMR\_NO\_INT,

11,

0,

0);

### btimer\_count\_set

Function Header

**void btimer\_count\_set(uint8\_t btimer\_id, uint32\_t count)**

Description

Program timer’s counter register

Note: Timer hardware may implement a 16-bit or 32-bit hardware counter. If the timer is 16-bit only the lower 16-bits of the count parameter are used.

Timers 0-3 use 16-bit count value and Timer 4-5 use 32-bit count value.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | timer ID – PID\_BTIMER\_x (x => 0-5) |
| count | New counter value |

Outputs

None

### btimer\_count\_get

Function Header

**uint32\_t btimer\_count\_get(uint8\_t btimer\_id)**

Description

Return current value of timer’s count register

Note: Timers 0-3 have 16-bit count value and Timer 4-5 will have 32-bit count value.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

32-bit or 16-bit timer count value

### btimer\_reload

Function Header

**void btimer\_reload(uint8\_t btimer\_id)**

Description

Force timer to reload counter from preload register

Note: Hardware will only reload counter if timer is running

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

None

### btimer\_start

Function Header

**void btimer\_start(uint8\_t btimer\_id)**

Description

Start timer counting

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

None

### btimer\_stop

Function Header

**void btimer\_stop(uint8\_t btimer\_id)**

Description

Stop Timer

Note: When a stopped timer is started again it will reload the count register from preload value

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

None

### btimer\_is\_started

Function Header

**uint8\_t btimer\_is\_started(uint8\_t btimer\_id)**

Description

Return state of timer’s START bit

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

0 (timer not started), 1 (timer started)

### btimer\_reset

Function Header

**void btimer\_reset(uint8\_t btimer\_id)**

Description

Perform soft reset of specified timer

Note: Soft reset set all registers to POR values. Spins 256 times waiting on hardware to clear reset bit (~1.6us with 48MHz clock).

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

None

### btimer\_halt

Function Header

**void btimer\_halt(uint8\_t btimer\_id)**

Description

Halt timer counting with no reload on unhalt

Note: A halted timer will not reload the count register when unhalted, it will continue counting from the current count value

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

None

### btimer\_unhalt

Function Header

**void btimer\_unhalt(uint8\_t btimer\_id)**

Description

Unhalt timer counting

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

None

### btimer\_interrupt\_enable

Function Header

**void btimer\_interrupt\_enable(uint8\_t btimer\_id, uint8\_t ien)**

Description

Enable/Disable specified timer’s interrupt from the block

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | timer ID – PID\_BTIMER\_x (x => 0-5) |
| ien | 1 – Enable timer block interrupt  0 – disable timer block interrupt |

Outputs

None

### btimer\_interrupt\_status\_get\_clr

Function Header

**uint8\_t btimer\_interrupt\_status\_get\_clr(uint8\_t btimer\_id)**

Description

Read Timer interrupt status and clear if set.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

1 (Timer interrupt status set) else 0

### btimer\_girq\_enable\_set

Function Header

**void btimer\_girq\_enable\_set(uint8\_t btimer\_id)**

Description

Enables GIRQ enable bit for the timer

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

None

### btimer\_girq\_enable\_clr

Function Header

**void btimer\_girq\_enable\_clr(uint8\_t btimer\_id)**

Description

Clears GIRQ enable bit for the timer

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

None

### btimer\_girq\_src\_get

Function Header

**void btimer\_girq\_src\_get(uint8\_t btimer\_id)**

Description

Returns GIRQ source bit for the timer

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

0(source bit not set), Non-zero (source bit set)

### btimer\_girq\_src\_clr

Function Header

**void btimer\_girq\_src\_clr(uint8\_t btimer\_id)**

Description

Clears GIRQ source bit for the timer

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

None

### btimer\_girq\_result\_get

Function Header

**uint8\_t btimer\_girq\_result\_get(uint8\_t btimer\_id)**

Description

Returns GIRQ result bit for the timer

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

0(result bit not set), Non-zero (result bit set)

### btimer\_sleep

Function Header

**void btimer\_sleep(uint8\_t btimer\_id, uint8\_t sleep\_en)**

Description

Enable/Disable clock gating on idle of a timer

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | timer ID – PID\_BTIMER\_x (x => 0-5) |
| sleep\_en | 1 = Sleep Enable  0 = Sleep Disable |

Outputs

None

### btimer\_clk\_reqd\_sts\_get

Function Header

**uint32\_t btimer\_clk\_reqd\_sts\_get (uint8\_t btimer\_id)**

Description

Returns clk required status

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

0(CLK not required), Non-zero (CLK required)

### btimer\_reset\_on\_sleep

Function Header

**void btimer\_reset\_on\_sleep (uint8\_t btimer\_id, uint8\_t reset\_en)**

Description

Enable/Disable timer block reset on sleep

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | timer ID – PID\_BTIMER\_x (x => 0-5) |
| reset\_en | 1 = Enable Reset on Sleep  0 = Disable Reset on Sleep |

Outputs

None

## Basic Timer Peripheral Functions

The list of Basic Timer Peripheral Functions

* 1. Functions to set and read Timer Counter Register
* btimer\_count\_get
* btimer\_count\_set

2. Function to program the Preload

* btimer\_preload\_set

3. Functions for basic timer interrupts

* btimer\_int\_status\_get
* btimer\_int\_status\_clear
* btimer\_int\_enable\_set
* btimer\_int\_enable\_clr

4. Functions for Control Register

* btimer\_ctrl\_write
* btimer\_ctrl\_read
* btimer\_ctrl\_enable\_set
* btimer\_ctrl\_enable\_clr
* btimer\_ctrl\_counter\_dir\_set
* btimer\_ctrl\_counter\_dir\_clr
* btimer\_ctrl\_auto\_restart\_set
* btimer\_ctrl\_auto\_restart\_clr
* btimer\_ctrl\_soft\_reset\_set
* btimer\_ctrl\_soft\_reset\_sts\_get
* btimer\_ctrl\_start\_set
* btimer\_ctrl\_start\_get
* btimer\_ctrl\_reload\_set
* btimer\_ctrl\_reload\_clr
* btimer\_ctrl\_halt\_set
* btimer\_ctrl\_halt\_clr

### p\_btimer\_count\_set

Function Header

**void p\_btimer\_count\_set (uint8\_t btimer\_id, uint32\_t count)**

Description

Sets timer counter

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | Timer ID – PID\_BTIMER\_x (x => 0-5) |
| count | 32-bit counter |

Outputs

None

### p\_btimer\_count\_get

Function Header

**uint32\_t p\_btimer\_count\_get (uint8\_t btimer\_id)**

Description

Read the timer counter

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | Timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

Counter value

### p\_btimer\_preload\_set

Function Header

**void p\_btimer\_preload\_set (uint8\_t btimer\_id, uint32\_t preload\_count)**

Description

Sets preload for the counter

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | Sets preload for the counter |
| preload\_count | 32-bit pre-load value |

Outputs

None

### p\_btimer\_int\_status\_get

Function Header

**uint8\_t p\_btimer\_int\_status\_get (uint8\_t btimer\_id)**

Description

Read the interrupt status bit in the timer block

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | Timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

1 if interrupt status set, else 0

### p\_btimer\_int\_status\_clear

Function Header

**void p\_btimer\_int\_status\_clear (uint8\_t btimer\_id)**

Description

Clears the interrupt status bit in the timer block

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | Timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

None

### p\_btimer\_int\_enable\_set

Function Header

**void p\_btimer\_int\_enable\_set (uint8\_t btimer\_id)**

Description

Sets interrupt enable bit in the timer block

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | Timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

None

### p\_btimer\_int\_enable\_clr

Function Header

**void p\_btimer\_int\_enable\_clr (uint8\_t btimer\_id)**

Description

Clears interrupt enable bit for the timer block

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | Timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

None

### p\_btimer\_ctrl\_write

Function Header

**void p\_btimer\_ctrl\_write (uint8\_t btimer\_id, uint32\_t value)**

Description

Writes the control register 32-bits

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | Timer ID – PID\_BTIMER\_x (x => 0-5) |
| Value | 32-bit value to program |

Outputs

None

### p\_btimer\_ctrl\_read

Function Header

**uint32\_t p\_btimer\_ctrl\_read (uint8\_t btimer\_id)**

Description

Reads the control register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | Timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

32-bit value read

### p\_btimer\_ctrl\_enable\_set

Function Header

**void p\_btimer\_ctrl\_enable\_set (uint8\_t btimer\_id)**

Description

Sets the enable bit in the control register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | Timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

None

### p\_btimer\_ctrl\_enable\_clr

Function Header

**void p\_btimer\_ctrl\_enable\_clr (uint8\_t btimer\_id)**

Description

Clears the enable bit in the control register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | Timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

None

### p\_btimer\_ctrl\_counter\_dir\_set

Function Header

**void p\_btimer\_ctrl\_counter\_dir\_set (uint8\_t btimer\_id)**

Description

Sets counter direction bit in the control register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | Timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

None

### p\_btimer\_ctrl\_counter\_dir\_clr

Function Header

**void p\_btimer\_ctrl\_counter\_dir\_clr (uint8\_t btimer\_id)**

Description

Clears counter direction bit in the control register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | Timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

None

### p\_btimer\_ctrl\_auto\_restart\_set

Function Header

**void p\_btimer\_ctrl\_auto\_restart\_set (uint8\_t btimer\_id)**

Description

Sets auto restart bit in the control register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | Timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

None

### p\_btimer\_ctrl\_auto\_restart\_clr

Function Header

**void p\_btimer\_ctrl\_auto\_restart\_clr (uint8\_t btimer\_id)**

Description

Clears auto restart bit in the control register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | Timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

None

### p\_btimer\_ctrl\_soft\_reset\_set

Function Header

**void p\_btimer\_ctrl\_soft\_reset\_set (uint8\_t btimer\_id)**

Description

Sets soft reset bit in the control register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | Timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

None

### p\_btimer\_ctrl\_soft\_reset\_sts\_get

Function Header

**uint8\_t p\_btimer\_ctrl\_soft\_reset\_sts\_get (uint8\_t btimer\_id)**

Description

Read soft reset bit in the control register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | Timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

0 if soft reset status bit cleared; else non-zero value

### p\_btimer\_ctrl\_start\_set

Function Header

**void p\_btimer\_ctrl\_start\_set (uint8\_t btimer\_id)**

Description

Sets start bit in the control register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | Timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

None

### p\_btimer\_ctrl\_start\_get

Function Header

**uint8\_t p\_btimer\_ctrl\_start\_get (uint8\_t btimer\_id)**

Description

Read start bit in the control register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | Timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

0 if start bit cleared; else non-zero value

### p\_btimer\_ctrl\_start\_clr

Function Header

**void p\_btimer\_ctrl\_start\_clr (uint8\_t btimer\_id)**

Description

Clears start bit in the control register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | Timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

None

### p\_btimer\_ctrl\_reload\_set

Function Header

**void p\_btimer\_ctrl\_reload\_set (uint8\_t btimer\_id)**

Description

Sets reload bit in the control register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | Timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

None

### p\_btimer\_ctrl\_reload\_clr

Function Header

**void p\_btimer\_ctrl\_reload\_clr (uint8\_t btimer\_id)**

Description

Clears reload bit in the control register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | Timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

None

### p\_btimer\_ctrl\_halt\_set

Function Header

**void p\_btimer\_ctrl\_halt\_set (uint8\_t btimer\_id)**

Description

Sets halt bit in the control register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | Timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

None

### p\_btimer\_ctrl\_halt\_clr

Function Header

**void p\_btimer\_ctrl\_halt\_clr (uint8\_t btimer\_id)**

Description

Clears halt bit in the control register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| btimer\_id | Timer ID – PID\_BTIMER\_x (x => 0-5) |

Outputs

None

# PWM

MEC2016 has 12 PWM channels, supports PWM frequency from 0.1 Hz to 500KHz.

User can use the below API’s. User can initiate the PWM channel & PWM frequency using PWM\_init routine.

User can vary the duty cycle with a minimum interval of 1%.

PWM\_enable should follow the PWM\_init routine to start the PWM generation.

User shall use the below API’s with valid PWM Channel Number.

**Number of PWM’s**

PWM0

PWM1

PWM2

PWM3

PWM4

PWM5

PWM6

PWM7

PWM8

PWM9

PWM10

PWM11

**PWM Peripheral Functions**

p\_PWM\_set\_ON\_time

p\_PWM\_counter\_ON\_Time\_read

p\_PWM\_set\_OFF\_time

p\_PWM\_counter\_OFF\_Time\_read

p\_PWM\_set\_predivider

p\_PWM\_set\_invert

p\_PWM\_select\_clock

p\_PWM\_enable

p\_PWM\_disable

p\_PWM\_configuration\_read

p\_PWM\_configuration\_write

**PWM API’s**

PWM\_init

PWM\_set\_dutycycle

PWM\_sleep\_enable

PWM\_sleep\_disable

pwm\_gpio\_configure

**PWM instance**

PWMx Counter ON Time Register

PWMx Counter OFF Time Register

PWMx Configuration Register

GPIO Peripheral Functions

PCR Peripheral Functions

## PWM API’s

The list of PWM APIs

* PWM\_init
* PWM\_set\_dutycycle
* PWM\_sleep\_enable
* PWM\_sleep\_disable
* pwm\_gpio\_configure

### PWM\_init

Function Header

**void PWM\_init(uint8\_t pwm\_ch,**

**uint32\_t pwm\_frequency,**

**uint8\_t invert,**

**uint8\_t dutycycle,**

**)**

Description

Configure the PWM channel with required configuration.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| pwm\_ch | Pwm ch – BPWMx\_ch (x => 0-3) |
| Pwm\_frequency | Period is calculated based on the pwm\_frequency  PWM\_frequency in Hz, use non-zero value only. |
| Invert | 0 – Invert not required (ON state is active High)  1 - Invert output (ON state is active Low) |
| Duty Cycle | Duty Cycle in percentage, minimum resolution is of 1%. |

Outputs

None

### PWM\_set\_dutycycle

Function Header

**void PWM\_set\_dutycycle(uint8\_t pwm\_ch,**

**uint32\_t pwm\_frequency,**

**uint8\_t dutycycle**

**)**

Description

Reinitialize or modify the duty cycle of any existing initialized PWM channel.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| pwm\_ch | Pwm Ch – BPWMx\_ch (x => 0-3) |
| Pwm\_frequency | Period is calculated based on the pwm\_frequency  PWM\_frequency in Hz, use non zero values. |
| Duty Cycle | Duty Cycle in percentage, minimum resolution is of 1%. |

Outputs

None

### PWM\_sleep\_enable

Function Header

**void PWM\_sleep\_enable(uint8\_t pwm\_ch)**

Description

Disables the PWM channel and put the specific PWM channel into sleep. This is to reduce the power consumption and keep the device in very low power state.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| pwm\_ch | Pwm ch – BPWMx\_ch (x => 0-3) |

Outputs

None

### PWM\_sleep\_disable

Function Header

**void PWM\_sleep\_disable(uint8\_t pwm\_ch)**

Description

Disable the sleep of the specific pwm channel. Need to call bPWM\_enable routine to start the PWM channel.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| pwm\_ch | Pwm ch – BPWMx\_ch (x => 0-3) |

Outputs

None

### PWM\_gpio\_configure

Function Header

**void pwm\_gpio\_configure(uint8\_t pwm\_ch)**

Description

Initializes the PWM channel GPIO pin based on the PWM channel ID.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| pwm\_ch | Pwm ch – BPWMx\_ch (x => 0-3) |

Outputs

None

## PWM Peripheral Functions

The list of PWM peripheral functions are listed below:

* p\_PWM\_set\_ON\_time
* p\_PWM\_counter\_ON\_Time\_read
* p\_PWM\_set\_OFF\_time
* p\_PWM\_counter\_OFF\_Time\_read
* p\_PWM\_set\_predivider
* p\_PWM\_set\_invert
* p\_PWM\_select\_clock
* p\_PWM\_enable
* p\_PWM\_disable
* p\_PWM\_configuration\_read
* p\_PWM\_configuration\_write

### p\_PWM\_set\_ON\_time

Function Header

**void p\_PWM\_set\_ON\_time (uint8\_t pwm\_ch,**

**uint16\_t ON\_time**

**)**

Description

Load the PWMx Counter ON time.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| Pwm\_ch | PWM channel number |
| ON\_time | Time of Pulse ON |

Outputs

None

### p\_PWM\_counter\_ON\_Time\_read

Function Header

**Uint32\_t** p\_PWM\_counter\_ON\_Time\_read  **(uint8\_t pwm\_ch)**

Description

Read the Counter ON Time register and returns its value.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| Pwm\_ch | PWM channel number |

Outputs

Returns counter value.

### p\_PWM\_set\_OFF\_time

Function Header

**void p\_PWM\_set\_OFF\_time (uint8\_t pwm\_ch,**

**uint16\_t OFF\_time**

**)**

Description

Load the PWMx Counter OFF time.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| Pwm\_ch | PWM channel number |
| OFF\_time | Time of Pulse OFF |

Outputs

None

### p\_PWM\_counter\_OFF\_Time\_read

Function Header

**Uint32\_t** p\_PWM\_counter\_OFF\_Time\_read  **(uint8\_t pwm\_ch)**

Description

Read the Counter OFF Time register and returns its value.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| Pwm\_ch | PWM channel number |

Outputs

Returns counter value.

### p\_PWM\_set\_predivider

Function Header

**void p\_PWM\_set\_predivider (uint8\_t pwm\_ch,**

**uint8\_t pre\_divider**

**)**

Description

Based on PWM frequency, set the required pre\_divider value. This is to choose the clock pulse for PWM generator.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| Pwm\_ch | PWM channel number |
| Pre-divider | Divider value to choose the clock input |

Outputs

None

### p\_PWM\_set\_invert

Function Header

**void p\_PWM\_set\_invert (uint8\_t pwm\_ch,**

**uint8\_t invert**

**)**

Description

If no invert is required load value 0, if invert output is required load value 1.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| Pwm\_ch | PWM channel number |
| Invert bit | 0 – Invert Off, PWM output High for Active High  1 - Invert ON, PWM output High for Active Low. |

Outputs

None

### p\_PWM\_select\_clock

Function Header

**void p\_PWM\_select\_clock (uint8\_t pwm\_ch,**

**uint8\_t Clock\_source**

**)**

Description

Choose the clock source required for PWM generator.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| Pwm\_ch | PWM channel number |
| Clock\_Source | 0 – High Clock – 48MHz Clock  1 - Low Clock - 100KHz Clock |

Outputs

None

### p\_PWM\_enable

Function Header

**void p\_PWM\_enable (uint8\_t pwm\_ch)**

Description

Set’s the enable bit in the PWM Configuration register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| Pwm\_ch | PWM channel number |

Outputs

None

### p\_PWM\_disable

Function Header

**void p\_PWM\_disable (uint8\_t pwm\_ch)**

Description

Set the disable bit in the PWM configuration register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| Pwm\_ch | PWM channel number |

Outputs

None

### p\_PWM\_configuration\_read

Function Header

**uint32\_t PWM\_configuration\_read(uint8\_t pwm\_ch)**

Description

Reads the Configuration register and returns its value.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| pwm\_ch | Pwm ch – BPWMx\_ch (x => 0-3) |

Outputs

Returns the Configuration register Value.

### p\_PWM\_configuration\_write

Function Header

**void PWM\_configuration\_read(uint8\_t pwm\_ch,**

**uint32\_t configuration**

**)**

Description

Update the configuration register value.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| pwm\_ch | Pwm ch – BPWMx\_ch (x => 0-3) |
| Configuration | Configuration value as per register settings. |

Outputs

Returns the Configuration register Value.

# GPIO

GPIO API’s

gpio\_init

gpio\_property\_set

gpio\_property\_get

gpio\_output\_set

gpio\_input\_get

gpio\_slewRate\_set

gpio\_slewRate\_get

gpio\_driveStr\_set

gpio\_driveStr\_get

GPIO Peripheral Functions

p\_gpio\_is\_valid

p\_gpio\_ctrl\_get

p\_gpio\_ctrl\_set

p\_gpio\_ctrl2\_get

p\_gpio\_ctrl2\_set

p\_gpio\_pad\_get

p\_gpio\_alt\_out

p\_gpio\_mux\_set

p\_gpio\_polarity\_set

p\_gpio\_output\_write\_enable

p\_gpio\_dir\_set

p\_gpio\_obuff\_set

p\_gpio\_idet\_set

p\_gpio\_pwrgate\_set

p\_gpio\_pud\_set

p\_gpio\_input\_get

p\_gpio\_output\_set

GPIO registers

GPIO Control 1 register

GPIO control 2 register

GPIO input register

GPIO output register

Number of GPIO’s

GPIO 000

To

GPIO 276

## 

## 

## GPIO APIs

The list of GPIO APIs

* gpio\_init
* gpio\_property\_set
* gpio\_property\_get
* gpio\_output\_set
* gpio\_input\_get
* gpio\_slewRate\_get
* gpio\_slewRate\_set
* gpio\_driveStr\_get
* gpio\_driveStr\_set

### gpio\_init

Function Header

**uint8\_t gpio\_init( enum GPIO\_PIN pin,**

**enum GPIO\_MUX new\_mux,**

**enum GPIO\_POLARITY new\_pol,**

**enum GPIO\_DIR new\_dir,**

**enum GPIO\_OUTDRV new\_obuf,**

**enum GPIO\_INTDET new\_idet,**

**enum GPIO\_PWRGATE new\_pwrg,**

**enum GPIO\_PUD new\_pud**

**)**

Description

Initializes the specified GPIO Pin

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****pin**** | 0-based GPIO ID |
| ****new\_mux**** | New output mux control mode  GPIO\_MUX\_GPIO – GPIO mode  GPIO\_MUX\_ALT\_FUNC1 – Signal 1 mode  GPIO\_MUX\_ALT\_FUNC2 – Signal 2 mode  GPIO\_MUX\_ALT\_FUNC3 – Signal 3 mode |
| ****new\_pol**** | Polarity mode  GPIO\_NON\_INVERTED  GPIO\_INVERTED |
| ****new\_dir**** | GPIO direction  GPIO\_INPUT  GPIO\_OUTPUT |
| ****new\_obuf**** | GPIO pin’s output buffer type  GPIO\_PUSH\_PULL  GPIO\_OPEN\_DRAIN |
| ****new\_idet**** | GPIO pin’s interrupt detection mode  GPIO\_INTDET\_LVL\_LO  GPIO\_INTDET\_LVL\_HI  GPIO\_INTDET\_DISABLED  GPIO\_INTDET\_EDG\_RISE  GPIO\_INTDET\_EDG\_FALL  GPIO\_INTDET\_EDG\_BOTH |
| ****new\_pwrg**** | GPIO pin’s power source  GPIO\_VCC1\_SUS  GPIO\_VCC2\_MAIN  GPIO\_ALWAYS\_UNPWRD  GPIO\_ALWAYS\_PWRD |
| ****new\_pud**** | GPIO pin’s internal resistor mode  GPIO\_PUD\_NONE  GPIO\_PU – Pull up mode  GPIO\_PD – Pull down mode  GPIO\_KEEPER |

Outputs

1 = success, 0 = fail

### gpio\_property\_set

Function Header

**uint8\_t gpio\_property\_set ( enum GPIO\_PIN pin,**

**enum GPIO\_PROPERTY gpio\_prop,**

**uint32\_t new\_prop\_val**

**)**

Description

Enables the user to change any property of the specified gpio pin at run time

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****pin**** | 0-based GPIO ID |
| gpio\_prop | Property type that is to be updated  GPIO\_PROP\_PU\_PD  GPIO\_PROP\_PWR\_GATE  GPIO\_PROP\_INT\_DET  GPIO\_PROP\_OBUFF\_TYPE  GPIO\_PROP\_DIR  GPIO\_PROP\_OUT\_SRC  GPIO\_PROP\_POLARITY  GPIO\_PROP\_MUX\_SEL  GPIO\_PROP\_ALL |
| new\_prop\_val | New value of the property  **GPIO\_PROP\_PU\_PD**  GPIO\_PUD\_NONE  GPIO\_PU – Pull up mode  GPIO\_PD – Pull down mode  GPIO\_KEEPER  **GPIO\_PROP\_PWR\_GATE**  GPIO\_VTR  GPIO\_VCC\_MAIN  GPIO\_ALWAYS\_UNPWRD  **GPIO\_PROP\_INT\_DET**  GPIO\_INTDET\_LVL\_LO  GPIO\_INTDET\_LVL\_HI  GPIO\_INTDET\_DISABLED  GPIO\_INTDET\_EDG\_RISE  GPIO\_INTDET\_EDG\_FALL  GPIO\_INTDET\_EDG\_BOTH  **GPIO\_PROP\_OBUFF\_TYPE**  GPIO\_PUSH\_PULL  GPIO\_OPEN\_DRAIN  **GPIO\_PROP\_DIR**  GPIO\_INPUT  GPIO\_OUTPUT  **GPIO\_PROP\_OUT\_SRC**  GPIO\_ALT\_OUT\_EN  GPIO\_ALT\_OUT\_DIS  **GPIO\_PROP\_POLARITY**  GPIO\_NON\_INVERTED  GPIO\_INVERTED  **GPIO\_PROP\_MUX\_SEL**  GPIO\_MUX\_GPIO  GPIO\_MUX\_ALT\_FUNC1  GPIO\_MUX\_ALT\_FUNC2  GPIO\_MUX\_ALT\_FUNC3 |

Outputs

1 = success, 0 = fail

### gpio\_property\_get

Function Header

**uint8\_t gpio\_property\_set ( enum GPIO\_PIN pin, enum GPIO\_PROPERTY gpio\_prop )**

Description

Returns the current value of the requested property type of the specified gpio pin

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****pin**** | 0-based GPIO ID |
| gpio\_prop | Property type that is to be read  GPIO\_PROP\_PU\_PD  GPIO\_PROP\_PWR\_GATE  GPIO\_PROP\_INT\_DET  GPIO\_PROP\_OBUFF\_TYPE  GPIO\_PROP\_DIR  GPIO\_PROP\_OUT\_SRC  GPIO\_PROP\_POLARITY  GPIO\_PROP\_MUX\_SEL |

Outputs

Property values (refer datasheet), 0xFF = fail

### gpio\_output\_set

Function Header

**uint8\_t gpio\_output\_set( enum GPIO\_PIN pin, enum GPIO\_ALT\_OUT out\_src, const uint32\_t gpio\_state )**

Description

Writes the output value to the specified gpio pin depending upon the output source register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****pin**** | 0-based GPIO ID |
| ****out\_src**** | Output source register  GPIO\_ALT\_OUT\_EN – bit[16] of control 1  register  GPIO\_ALT\_OUT\_DIS – output register |
| gpio\_state | Desired pin state |

Outputs

1 = success, 0 = fail

### gpio\_input\_get

Function Header

**uint8\_t gpio\_input\_get( enum GPIO\_PIN pin )**

Description

Reads the GPIO pin’s input register using the gpio input register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****pin**** | 0-based GPIO ID |

Outputs

0 or 1 = pin state, 0xFF = fail

### gpio\_slewRate\_get

Function Header

**uint8\_t gpio\_slewRate\_get( enum GPIO\_PIN pin )**

Description

Returns the current slew rate configuration of the specified GPIO Pin.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****pin**** | 0-based GPIO ID |

Outputs

Pin slew rate: 0 = slow, 1 = fast, 0xFF = fail

### gpio\_slewRate\_set

Function Header

**uint8\_t gpio\_slewRate\_set (enum GPIO\_PIN pin, enum GPIO\_SLEW new\_slew )**

Description

Programs the slew rate configuration for the specified GPIO Pin.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| gpio\_id | gpio\_id 0-based GPIO ID |
| new\_slew | new slew rate setting  GPIO\_SLEW\_SLOW  GPIO\_SLEW\_FAST |

Outputs

1 = success, 0 = fail

### gpio\_driveStr\_get

Function Header

**uint8\_t gpio\_driveStr\_get( enum GPIO\_PIN pin )**

Description

Reads the current drive strength setting of the specified gpio pin.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| pin | 0-based GPIO ID |

Outputs

Pin Drive Strength: 0 = 2mA, 1 = 4mA, 2 = 8mA, 3 = 12mA, 0xFF = fail

### gpio\_driveStr\_set

Function Header

**uint8\_t gpio\_driveStr\_set ( enum GPIO\_PIN pin, enum GPIO\_DRV drv\_str )**

Description

Programs the drive strength configuration for the specified gpio pin.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| pin | 0-based GPIO ID |
| drv\_str | Drive strength value  GPIO\_DRV\_2MA  GPIO\_DRV\_4MA  GPIO\_DRV\_8MA  GPIO\_DRV\_12MA |

Outputs

1 = success, 0 = fail

## GPIO Peripheral Functions

List of GPIO Peripheral functions

* p\_gpio\_is\_valid
* p\_gpio\_ctrl\_get
* p\_gpio\_ctrl\_set
* p\_gpio\_ctrl2\_get
* p\_gpio\_ctrl2\_set
* p\_gpio\_pad\_get
* p\_gpio\_alt\_out
* p\_gpio\_mux\_set
* p\_gpio\_polarity\_set
* p\_gpio\_output\_write\_enable
* p\_gpio\_dir\_set
* p\_gpio\_obuff\_set
* p\_gpio\_idet\_set
* p\_gpio\_pwrgate\_set
* p\_gpio\_pud\_set
* p\_gpio\_input\_get
* p\_gpio\_output\_set

### p\_gpio\_is\_valid

Function Header

**uint8\_t** p\_gpio\_is\_valid **( enum GPIO\_PIN pin )**

Description

Checks if the specified gpio pin has been implemented in the hardware.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****pin**** | 0-based GPIO ID |

Outputs

1 = valid, 0 = invalid

### p\_gpio\_ctrl\_get

Function Header

**uint32\_t** p\_gpio\_ctrl\_get **( enum GPIO\_PIN pin )**

Description

Reads the contents of the control 1 register of the specified gpio pin

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****pin**** | 0-based GPIO ID |

Outputs

32-bit value of GPIO pin’s control 1 register contents

### p\_gpio\_ctrl\_set

Function Header

**void** p\_gpio\_ctrl\_set **( enum GPIO\_PIN pin, uint32\_t new\_ctrl )**

Description

Writes to the control 1 register of the specified gpio pin.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****pin**** | 0-based GPIO ID |
| ****new\_ctrl**** | 32-bit value of the new value |

Outputs

None

### p\_gpio\_ctrl2\_get

Function Header

**uint8\_t** p\_gpio\_ctrl2\_get **( enum GPIO\_PIN pin )**

Description

Reads the contents of the control 2 register of the specified gpio pin

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****pin**** | 0-based GPIO ID |

Outputs

GPIO pin’s control 2 register contents

### p\_gpio\_ctrl2\_set

Function Header

**void** p\_gpio\_ctrl2\_set **( enum GPIO\_PIN pin, uint8\_t new\_ctrl2 )**

Description

Writes to the control 2 register of the specified gpio pin.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****pin**** | 0-based GPIO ID |
| ****new\_ctrl2**** | 32-bit value of the new value |

Outputs

None

### p\_gpio\_pad\_get

Function Header

**uint8\_t** p\_gpio\_pad\_get **( enum GPIO\_PIN pin )**

Description

Read GPIO pin’s input via the GPIO\_INPUT bit of the control register.

Note: Performs a byte read of offset 3 of the GPIO Pin’s 32-bit control 1 register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****pin**** | 0-based GPIO ID |

Outputs

Current state of the gpio pin

### p\_gpio\_alt\_out

Function Header

**void** p\_gpio\_alt\_out **( enum GPIO\_PIN pin, uint8\_t new\_val )**

Description

Writes to the gpio pin via the ALTERNATE\_GPIO\_DATA bit of the control 1 register

Note:

1. To use this feature, ‘Output GPIO Write Enable’ bit should be cleared in the GPIO

Control 1 register.

1. Performs a byte wide write to byte offset 2 of the GPIO Pin’s 32-bit configuration register. No read-modify-write.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****pin**** | 0-based GPIO ID |
| ****new\_val**** | New output value |

Outputs

None

### p\_gpio\_mux\_set

Function Header

**void** p\_gpio\_mux\_set **( enum GPIO\_PIN pin, uint8\_t new\_mux )**

Description

Sets the mode for the gpio pin’s output mux

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****pin**** | 0-based GPIO ID |
| ****new\_mux**** | New mux mode  GPIO\_MUX\_GPIO – GPIO mode  GPIO\_MUX\_ALT\_FUNC1 – Signal 1 mode  GPIO\_MUX\_ALT\_FUNC2 – Signal 2 mode  GPIO\_MUX\_ALT\_FUNC3 – Signal 3 mode |

Outputs

None

### p\_gpio\_polarity\_set

Function Header

**void** p\_gpio\_polarity\_set **( enum GPIO\_PIN pin, enum GPIO\_POLARITY invert )**

Description

Sets the mode for the gpio pin’s polarity

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****pin**** | 0-based GPIO ID |
| ****invert**** | New polarity mode  GPIO\_NON\_INVERTED  GPIO\_INVERTED |

Outputs

None

### p\_gpio\_output\_write\_enable

Function Header

**void** p\_gpio\_outen\_set **( enum GPIO\_PIN pin, enum GPIO\_ALT\_OUT enable\_par\_out )**

Description

Selects the output source register for the specified gpio pin.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****pin**** | 0-based GPIO ID |
| ****enable\_par\_out**** | Output register  GPIO\_ALT\_OUT\_EN – bit[16] of control 1 register  GPIO\_ALT\_OUT\_DIS – gpio output register |

Outputs

None

### p\_gpio\_dir\_set

Function Header

**void** p\_gpio\_dir\_set **( enum GPIO\_PIN pin, enum GPIO\_DIR dir\_output )**

Description

Sets the direction of the specified gpio pin

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****pin**** | 0-based GPIO ID |
| ****dir\_output**** | Direction mode value  GPIO\_INPUT  GPIO\_OUTPUT |

Outputs

None

### p\_gpio\_obuff\_set

Function Header

**void** p\_gpio\_obuff\_set **( enum GPIO\_PIN pin, enum GPIO\_OUTDRV new\_obuf )**

Description

Selects the output buffer for the specified gpio pin

Inputs

|  |  |
| --- | --- |
| *Input Parameter* | Description |
| ****pin**** | 0-based GPIO ID |
| ****new\_obuff**** | Output buffer type  GPIO\_PUSH\_PULL  GPIO\_OPEN\_DRAIN |

Outputs

None

### p\_gpio\_idet\_set

Function Header

**void** p\_gpio\_idet\_set **( enum GPIO\_PIN pin, enum GPIO\_INTDET new\_idet )**

Description

Selects the interrupt detection mode for the specified gpio pin.

Note: This function accounts for all the possible combinations including bit[7] – EDGE\_ENABLE of the control 1 register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****pin**** | 0-based GPIO ID |
| ****new\_idet**** | Interrupt detection mode  GPIO\_INTDET\_LVL\_LO  GPIO\_INTDET\_LVL\_HI  GPIO\_INTDET\_DISABLED  GPIO\_INTDET\_EDG\_RISE  GPIO\_INTDET\_EDG\_FALL  GPIO\_INTDET\_EDG\_BOTH |

Outputs

None

### p\_gpio\_pwrgate\_set

Function Header

**void** p\_gpio\_pwrgate\_set **( enum GPIO\_PIN pin, enum GPIO\_PWRGATE new\_pwrg )**

Description

Selects the power gating source for the specified gpio pin.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****pin**** | 0-based GPIO ID |
| ****new\_pwrg**** | Power gate mode  GPIO\_VTR  GPIO\_VCC\_MAIN  GPIO\_ALWAYS\_UNPWRD |

Outputs

None

### p\_gpio\_pud\_set

Function Header

**void** p\_gpio\_pud\_set **( enum GPIO\_PIN pin, enum GPIO\_PUD new\_pud )**

Description

Selects the internal resistor mode for the specified gpio pin

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****pin**** | 0-based GPIO ID |
| ****new\_pud**** | Internal resistor mode select  GPIO\_PUD\_NONE  GPIO\_PU – Pull up mode  GPIO\_PD – Pull down mode  GPIO\_KEEPER |

Outputs

None

### p\_gpio\_input\_get

Function Header

**uint8\_t** p\_gpio\_input\_get **( enum GPIO\_PIN pin )**

Description

Reads the input value of the specified gpio pin using the gpio input register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****pin**** | 0-based GPIO ID |

Outputs

0 or 1 = gpio input state, 0xFF = invalid pin

### p\_gpio\_output\_set

Function Header

**void** p\_gpio\_output\_set **( enum GPIO\_PIN pin, const uint32\_t new\_val )**

Description

Writes to the gpio output register of the specified gpio pin.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****pin**** | 0-based GPIO ID |
| ****New\_val**** | New output value |

Outputs

None

# SMBus Driver

## SMBus Driver APIs & Callbacks

* Driver Set Up & Initialization
  + smb\_callback
  + smb\_register\_eventFlag\_and\_callback
  + smb\_dma\_isr
  + smb\_isr
  + smbus\_main\_task
  + smbus\_app\_timer
  + smbus\_init\_timer
* Configuring SMBus Controller
  + smbus\_configure\_and\_enable
  + smbus\_disable
  + smb\_enable\_timeouts
* MASTER APIs
  + smb\_busyStatus\_get
  + smb\_portBusyStatus\_get
  + smb\_change\_port
  + smb\_set\_speed
  + smb\_protocol\_execute
  + smb\_protocol\_execute\_blocking
  + Master callback function
* SLAVE APIs
  + smb\_register\_slave
  + smb\_deregister\_slave
  + smbApp\_slave\_callback

## SMBus driver configuration

The SMBus driver can be customized and configured based on the requirements. User needs to update the smb\_config\_user.h file. The file will have the following entries:

/\* Maximum number of smbus controller \*/

#define MAX\_SMB 4

/\* Maximum number of ports per SMBus controller \*/

#define SMB\_MAX\_PORT\_PER\_CHANNEL 11

/\* Maximum number of buffers per slave controller \*/

#define SMB\_MAX\_NUM\_SLAVE\_BUFFER 8

/\* Size of buffer on each controller (must be greater than 4) \*/

#define SLAVE1\_BUFFER\_SIZE 64

#define SLAVE2\_BUFFER\_SIZE 64

#define SLAVE3\_BUFFER\_SIZE 64

#define SLAVE4\_BUFFER\_SIZE 64

/\* Maximum number of application per slave controller \*/

#define SMB\_MAX\_NUM\_SLAVE\_APP 5

## Driver Set Up & Initialization

**FreeRTOS**

For hooking-in the smbus/i2c driver to FreeRTOS, it needs certain services and calls.

In summary, following is the sequence to be followed:

1. Create a smbus callback function whose prototype is

*void smb\_callback(UINT8 channel, UINT8 eventType, UINT8 eventValue)*

1. In the irq12 interrupt service routine call *smb\_isr* function
2. In the irq13 interrupt service routine call *smb\_dma\_isr* function
3. As a part of global initialization (before entering freeRTOS) call *smbus\_init\_task* function
4. In FreeRTOS :-
   1. Create a freeRTOS event flag for smbus driver purpose
   2. Register the event flag and the smbus callback using the *smb\_register\_eventFlag\_and\_callback* function
   3. Create a freeRTOS timer which is set to call *smbus\_app\_timer* every 10ms
   4. Create a FreeRTOS task with stack size 512 (TBD) bytes with *smbus\_main* as the call back function

**SKERN**

For hooking-in the smbus/i2c driver to SKERN, following sequence needs to be followed:

1. In cfg.h, create a task (task number can vary) for smbus as shown below:

#define TASK\_01 smbus

#define PRIORITY\_TASK\_01\_EVENTTASK HIGH

#define PRIORITY\_TASK\_01\_EVENTTIMEOUT HIGH

#define ENABLE\_TASK\_01\_EVENTINTR 1

#define ENABLE\_TASK\_01\_EVENTTASK 1

#define ENABLE\_TASK\_01\_EVENTTIMER 1

Note: This will automatically create function declarations for smbus\_init\_task and smbus\_main\_task.

1. In the irq12 interrupt service routine call *smb\_isr* function
2. In the irq13 interrupt service routine call *smb\_dma\_isr* function

### smb\_callback

Function Header

**void smb\_callback(const UINT8 channel,const UINT8 eventType,const UINT8 eventValue)**

Description

This function is the callback function which is invoked by smbus driver at various instances for notifications.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| channel | SMB\_CHANNEL\_x – x is 1 to 4 |
| eventType | the type of event for notification |
| eventValue | parameter for the notification, if any |

|  |  |
| --- | --- |
| eventType | |
| SMB\_CBK\_DISABLED | controller is disabled |
| SMB\_CBK\_HW\_ENABLED | controller is enabled |
| SMB\_CBK\_BER | bus error on the controller |
| SMB\_CBK\_BUSY | waiting for current transaction to complete before disabling |
| SMB\_CBK\_PORT\_ERROR\_SET | Error in port (clk/data not high) |
| SMB\_CBK\_PORT\_ERROR\_CLR | Port is good (clk/data high) |

Outputs

None

### smb\_register\_eventFlag\_and\_callback

Function Header

**void smb\_register\_eventFlag\_and\_callback(EventGroupHandle\_t \*ptr\_event\_flag\_handle, SMB\_CALLBACK\_FUNC\_PTR pCallback)**

Description

Register FreeRTOS event flag. This function needs to be called only in FreeRTOS framework.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ptr\_event\_flag | ptr\_event\_flag pointer to RTOS event flag structure |
| pCallback | pointer to callback function |

Outputs

None

### smb\_dma\_isr

Function Header

**void smb\_dma\_isr(void)**

Description

This function takes care of smbus dma interrupts. This needs to be called in irq13 interrupt routine from the application code. Smbus dma are dma controller instance 0 to 7.

Inputs

None

Outputs

None

### smb\_isr

Function Header

**void smb\_isr(void)**

Description

The main entry point for smbus interrupt service routine. This needs to be called in irq12 interrupt routine from the application code.

Inputs

None

Outputs

None

### smbus\_main\_task

Function Header (skern)

VOID smbus\_main\_task(enum EVENT\_TYPE call\_type)

Function Header (freeRTOS)

void smbus\_main(void \*pvParameters)

Description

This function performs all the SMBus tasks. This function is passed as the entry for FreeRTOS and is the main task for SKERN.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| Call\_type | Event Type – Interrupt/Task/Timer |

|  |  |
| --- | --- |
| Input Parameter | Description |
| pvParameters | Any parameters passed by application |

Outputs

None

### smbus\_app\_timer

Function Header

**void smbus\_app\_timer(void)**

Description

SMBus application timer function.

Note : This function internally sets flag to call smbus\_timer\_task. This function is only required for RTOS.

Inputs

None

Outputs

None

### smbus\_init\_task

Function Header

void smbus\_init\_task (void)

Description

This function initializes the smbus data structures.

Note: This function needs to be called before entering FreeRTOS.

Inputs

None

Outputs

None

## Configuring SMBus Controller

### smbus\_configure\_and\_enable

Function Header

**void smbus\_configure\_and\_enable ( UINT8 channel,**

**UINT8 own\_address,**

**UINT8 speed,**

**UINT8 port,**

**UINT8 configFlag)**

Description

This function can be used to start and enable the smbus controller

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| channel | SMB\_CHANNEL\_x – x is 1 to 4 |
| own\_address | 7-bit smb address |
| speed | SMBUS\_SPEED\_100KHZ SMBUS\_SPEED\_400KHZ  SMBUS\_SPEED\_1MHZ |
| **port** | default port on the controller |
| **configValue** | Bit 0 – set to enable  Bit 2 – enable Fairness |

Outputs

None

### smbus\_disable

Function Header

void smbus\_disable(UINT8 channel)

Description

This function can be used to disable the smbus controller.

Note: Make the sure the controller was enabled prior to using this function

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| channel | SMB\_CHANNEL\_x – x is 1 to 4 |

Outputs

None

### smb\_enable\_timeouts

Function Header

**void smb\_enable\_timeouts(const UINT8 channel, const UINT8 timeoutsFlag)**

Description

This function enables timeouts.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| channel | SMB\_CHANNEL\_x – x is 1 to 4 |
| timeoutsFlag | Timeout flag as per BYTE0 of completion register |

timeoutsFlag - timeoutsFlag is as per BYTE0 of completion register

BIT 2 – Device Timeout enable

BIT 3 – Master Cumulative Timeout enable

BIT 4 – Slave Cumulative Timeout enable

BIT 5 – Bus Idle Detect Timeout enable

Outputs

None

## MASTER APIs

The steps to initiate any master transaction are as follows:

1. Check if smbus resource is available using busy status API
2. If smbus resource is available:
   1. Change the port or speed (if desired)
   2. Initiate the master transaction using smb\_protocol\_execute API.

### smb\_busyStatus\_get

Function Header

**UINT8 smb\_busyStatus\_get (const UINT8 channel)**

Description

This function checks if master resource is available on the default port.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| channel | SMB\_CHANNEL\_x – x is 1 to 4 |

Outputs

MASTER\_BUSY - controller or default port is busy

MASTER\_AVAILABLE - controller is available for use on the default port

Usage

/\* Get SMBus resource status \*/

status = **smb\_busyStatus\_get**(SMB\_CHANNEL\_1);

if (MASTER\_BUSY == status)

{

trace0(0, SMB\_APP, 0, “smbApp\_master\_request: smbus is busy”);

/\* Steps for retry \*/

…

…

}

/\* Proceed to smb\_protocol\_execute to initiate master request \*/

### smb\_portBusyStatus\_get

Function Header

**UINT8 smb\_portBusyStatus\_get (const UINT8 channel,const UINT8 port)**

Description

This function checks if master resource is available on the queried port.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| channel | SMB\_CHANNEL\_x – x is 1 to 4 |
| port | port on the particular channel for which busy status is required |

Outputs

MASTER\_BUSY - controller or port is busy

MASTER\_AVAILABLE - controller is available for use on the port

Usage

/\* Get SMBus resource status \*/

status = **smb\_portBusyStatus\_get**(SMB\_CHANNEL\_1, SMB\_PORT\_1);

if (MASTER\_BUSY == status)

{

trace0(0, SMB\_APP, 0, “smbApp\_master\_request: smbus is busy”);

/\* Steps for retry \*/

…

…

}

/\* Proceed to smb\_protocol\_execute to initiate master request \*/

### smb\_change\_port

Function Header

**UINT8 smb\_change\_port(const UINT8 channel,const UINT8 port)**

Description

This function changes the controller port

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| channel | SMB\_CHANNEL\_x – x is 1 to 4 |
| port | port on the particular channel for which busy status is required |

Outputs

MASTER\_OK - port change success

MASTER\_ERROR - port or controller busy / port change error

### smb\_set\_speed

Function Header

**void smb\_set\_speed(const UINT8 channel,const UINT8 speed)**

Description

This function changes smbus speed

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| channel | SMB\_CHANNEL\_x – x is 1 to 4 |
| speed | SMBUS\_SPEED\_100KHZ SMBUS\_SPEED\_400KHZ  SMBUS\_SPEED\_1MHZ |

Outputs

None

Example Usage

None

### smb\_protocol\_execute

Function Header

**UINT8 smb\_protocol\_execute (const UINT8 channel,**

**UINT8 \*buffer\_ptr,**

**const UINT8 smb\_protocol,**

**const UINT8 writeCount,**

**const UINT8 pecEnable,**

**MASTER\_FUNC\_PTR func\_ptr,**

**const UINT8 readChainedFlag,**

**const UINT8 writeChainedFlag)**

Description

Initiates smbus master operation. This function is called by the application whenever it wants to initiate a master transaction on the smbus.

Note: For Read Block protocol the application should provide an 80 byte buffer.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| channel | SMB\_CHANNEL\_x – x is 1 to 4 |
| buffer\_ptr | Buffer for the smbus transaction |
| smb\_protocol | smbus protocol Byte |
| **writeCount** | Number of bytes to transmit |
| **pecEnable** | Flag to enable/disable PEC |
| **func\_ptr** | Function to call after success/failure of the transaction. This function will be invoked on completion of transaction with status. The function type is:  typedef void (MASTER\_FUNC\_PTR)(UINT8, UINT8 \*)  The first parameter is the status, the next is the buffer\_ptr that was passed. |
| **readChainedFlag** | flag to indicate if read needs to be done using dma chaining |
| **writeChainedFlag** | flag to indicate if write needs to be done using dma chaining |

|  |
| --- |
| smb\_protocol |
| SMB\_I2C\_WRITE |
| SMB\_I2C\_READ |
| SMB\_I2C\_COMBINED |
| SMB\_SEND\_BYTE |
| SMB\_RECEIVE\_BYTE |
| SMB\_WRITE\_BYTE |
| SMB\_WRITE\_WORD |
| SMB\_READ\_BYTE |
| SMB\_READ\_WORD |
| SMB\_WRITE\_BLOCK |
| SMB\_READ\_BLOCK |

Outputs

MASTER\_OK - success

MASTER\_ERROR - port or controller bus error

Note: If this function returns MASTER\_ERROR, application could retry after some time.

Usage

Sequence of events for smb\_protocol\_execute:

1. Application calls this function to initiate any master transaction. So this function is executed in the caller’s thread context
2. This function returns immediately after programming the smb hardware registers. Application is free to do any other tasks
3. Once the transaction completes on the bus, smbus driver calls the application callback with the transaction status – success or failure. The callback is invoked from the smbus driver thread context
4. Application can decide in the callback if it needs to retry (APP\_RETVAL\_RETRY) or initiate a new transaction (APP\_RETVAL\_NEW\_TX) or release control (APP\_RETVAL\_RELEASE\_SMBUS). See master callback function for details.

Example for sending a master Write Byte transaction:

UINT8 smbAppTxBuffer[10];

smbAppTxBuffer[0] = SMB\_APP\_SLAVE\_ADDRESS;

/\* Form the Write Byte information in Tx Buffer \*/

smbAppTxBuffer[1] = WRITE\_BYTE\_ADDR;

smbAppTxBuffer[2] = WRITE\_BYTE\_VALUE;

smb\_protocol = SMB\_WRITE\_BYTE;

smb\_writeCount = 3;

status = **smb\_protocol\_execute**(smb\_channel, &smbAppTxBuffer[0],

smb\_protocol,smb\_writeCount, FALSE, smbApp\_callback, FALSE);

Example for sending a master Read Byte transaction:

UINT8 smbAppTxBuffer[10];

smbAppTxBuffer[0] = SMB\_APP\_SLAVE\_ADDRESS;

/\* Form the Read Byte information in Tx Buffer \*/

smbAppTxBuffer[1] = READ\_BYTE\_ADDR;

smbAppTxBuffer[2] = SMB\_APP\_SLAVE\_ADDRESS;

smb\_protocol = SMB\_READ\_BYTE;

smb\_writeCount = 3;

status = **smb\_protocol\_execute**(smb\_channel,&smbAppTxBuffer[0],

smb\_protocol,smb\_writeCount, FALSE, smbApp\_callback, FALSE);

Example for sending a master Write Word transaction:

UINT8 smbAppTxBuffer[10];

smbAppTxBuffer[0] = SMB\_APP\_SLAVE\_ADDRESS;

/\* Form the Write Word information in Tx Buffer \*/

smbAppTxBuffer[1] = WRITE\_WORD\_ADDR;

smbAppTxBuffer[2] = WRITE\_WORD\_DATA\_BYTE\_1;

smbAppTxBuffer[3] = WRITE\_WORD\_DATA\_BYTE\_2;

smb\_protocol = SMB\_WRITE\_WORD;

smb\_writeCount = 4;

status = **smb\_protocol\_execute**(smb\_channel,&smbAppTxBuffer[0],

smb\_protocol,smb\_writeCount, FALSE, smbApp\_callback, FALSE);

Example for sending a master Read Word transaction:

UINT8 smbAppTxBuffer[10];

smbAppTxBuffer[0] = SMB\_APP\_SLAVE\_ADDRESS;

/\* Form the Read Word information in Tx Buffer \*/

smbAppTxBuffer[1] = READ\_WORD\_ADDR;

smbAppTxBuffer[2] = SMB\_APP\_SLAVE\_ADDRESS;

smb\_protocol = SMB\_READ\_WORD;

smb\_writeCount = 3;

status = **smb\_protocol\_execute**(smb\_channel,&smbAppTxBuffer[0],

smb\_protocol,smb\_writeCount, FALSE,smbApp\_callback, FALSE);

Example for sending a master Send Byte transaction:

UINT8 smbAppTxBuffer[10];

smbAppTxBuffer[0] = SMB\_APP\_SLAVE\_ADDRESS;

/\* Form the Send Byte information in Tx Buffer \*/

smbAppTxBuffer[1] = SEND\_BYTE\_DATA;

smb\_protocol = SMB\_SEND\_BYTE;

smb\_writeCount = 2;

status = **smb\_protocol\_execute**(smb\_channel,&smbAppTxBuffer[0],

smb\_protocol,smb\_writeCount, FALSE,smbApp\_callback, FALSE);

Example for sending a master Receive Byte transaction:

UINT8 smbAppTxBuffer[10];

smbAppTxBuffer[0] = SMB\_APP\_SLAVE\_ADDRESS;

/\* Form the Receive Byte information in Tx Buffer \*/

smb\_protocol = SMB\_RECEIVE\_BYTE;

smb\_writeCount = 1;

status = **smb\_protocol\_execute**(smb\_channel,&smbAppTxBuffer[0],

smb\_protocol, smb\_writeCount, FALSE,smbApp\_callback, FALSE);

Example for sending a master Write Block transaction:

UINT8 smbAppTxBuffer[10];

smbAppTxBuffer[0] = SMB\_APP\_SLAVE\_ADDRESS;

/\* Form the Write Block information in Tx Buffer \*/

smbAppTxBuffer[1] = WRITE\_BLOCK\_COMMAND\_CODE;

smbAppTxBuffer[2] = WRITE\_BLOCK\_DATA\_LEN; //(5)

smbAppTxBuffer[3] = WRITE\_BLOCK\_DATA\_1;

smbAppTxBuffer[4] = WRITE\_BLOCK\_DATA\_2;

smbAppTxBuffer[5] = WRITE\_BLOCK\_DATA\_3;

smbAppTxBuffer[6] = WRITE\_BLOCK\_DATA\_4;

smbAppTxBuffer[7] = WRITE\_BLOCK\_DATA\_5;

smb\_protocol = SMB\_WRITE\_BLOCK;

smb\_writeCount = 3 + WRITE\_BLOCK\_DATA\_LEN;

status = **smb\_protocol\_execute**(smb\_channel,&smbAppTxBuffer[0],

smb\_protocol,smb\_writeCount, FALSE, smbApp\_callback, FALSE);

### smb\_protocol\_execute\_blocking

Function Header

**UINT8 smb\_protocol\_execute\_blocking(const UINT8 channel,**

**UINT8 \*buffer\_ptr,**

**const UINT8 smb\_protocol,**

**const UINT8 writeCount,**

**const UINT8 pecEnable,**

**MASTER\_FUNC\_PTR func\_ptr,**

**const UINT8 readChainedFlag,**

**const UINT8 writeChainedFlag)**

Description

Initiates smbus master operation which is blocking i.e. returns only when the transaction is complete.

Note: For Read Block protocol the application should provide an 80 byte buffer.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| channel | SMB\_CHANNEL\_x – x is 1 to 4 |
| buffer\_ptr | Buffer for the smbus transaction |
| smb\_protocol | smbus protocol Byte |
| **writeCount** | Number of bytes to transmit |
| **pecEnable** | Flag to enable/disable PEC |
| **func\_ptr** | Function to call after success/failure of the transaction. This function will be invoked on completion of transaction with status. The function type is:  typedef void (MASTER\_FUNC\_PTR)(UINT8, UINT8 \*)  The first parameter is the status, the next is the buffer\_ptr that was passed. |
| **readChainedFlag** | flag to indicate if read needs to be done using dma chaining |
| **writeChainedFlag** | flag to indicate if write needs to be done using dma chaining |

Outputs

MASTER\_OK on success, MASTER\_ERROR if smbus is not ready for master mode operation.

Note: If this function returns MASTER\_ERROR, application could retry after some time.

Usage

This function is the blocking version of smb\_protocol\_execute function. The sequence of events is as follows:

1. Application calls this function to initiate any master transaction. So this function is executed in the caller’s thread context
2. This function blocks (waits on a hread event flag) after programming the smb hardware registers. As such application thread is blocked. It cannot do any other task.
3. Once the transaction completes on the bus, smbus driver calls the application callback with the transaction status – success or failure. The callback is invoked from the smbus driver thread context.
4. Application can decide in the callback if it needs to retry (APP\_RETVAL\_RETRY) or initiate a new transaction (APP\_RETVAL\_NEW\_TX) or release control (APP\_RETVAL\_RELEASE\_SMBUS). See master callback function for details.
5. If the application callback returns APP\_RETVAL\_RELEASE\_SMBUS, then smbus driver raises the event to release the blocking. This will cause this blocking function to return.

### Master callback function

Function Header

UINT8 smbApp\_master\_callback (UINT8 channel,

UINT8 status,

UINT8 \*buffer\_ptr, SMB\_MAPP\_CBK\_NEW\_TX \*newTxParams)

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| channel | SMB\_CHANNEL\_x – x is 1 to 4 |
| Status | Status table given below |
| buffer\_ptr | pointer to application passed buffer |
| newTxParams | pointer to structure defining new master transaction |

|  |  |
| --- | --- |
| *status* | |
| ERROR\_BER\_TIMEOUT | Bus Error |
| ERROR\_BER\_NON\_TIMEOUT | Bus Error (cause is timeout) |
| ERROR\_LAB | Lost Arbitration Error |
| ERROR\_MADDR\_NAKX | Address NAK |
| ERROR\_MDATA\_NAKX | Slave sent data NAK |
| ERROR\_SMB\_DISABLED | smbus is disabled |
| ERROR\_CLK\_DATA\_NOT\_HIGH | clk or data not high |
| ERROR\_PEC | PEC error |
| SUCCESS\_TX | Success TX operation |
| SUCCESS\_RX | Success RX operation |
| SUCCESS\_RX\_CHAINED | Successfully performed RX chained operation (intermediate status) |

-

Outputs

|  |  |
| --- | --- |
| *smbApp\_master\_callback return values* | |
| APP\_RETVAL\_RELEASE\_SMBUS | Application releases hold on smbus |
| APP\_RETVAL\_HOLD\_SMBUS | Application still acquires smbus |
| APP\_RETVAL\_RETRY | Application wants to retry the current transaction |
| APP\_RETVAL\_NEW\_TX | Application starts a new Master TX immediately |
| APP\_RETVAL\_CHAINED\_RX | Application is continuing a chained RX transaction |
| APP\_RETVAL\_CHAINED\_RX\_LAST | Last request for a chained RX transaction |

Description

This is the function defined in the application whose address will be passed as part of the master transaction request. Master callback function will be invoked from the smbus driver once the master transaction terminates either successfully or with any error. Application can decide in the callback function whether to retry the current transaction or initiate a new smbus transaction or release smbus resource for other applications.

If new transaction or retry is initiated from the callback when a blocking call (smb\_protocol\_execute\_blocking) was used then the blocking would continue until the new transaction or retry initiated from the callback is completed.

For a blocking request, only when the application returns APP\_RETVAL\_RELEASE\_SMBUS then the blocking would be released.

For a non-blocking request (smb\_protocol\_execute) returning a status APP\_RETVAL\_HOLD\_SMBUS allows the application to effectively block other application from using the controller, since they will get busy status.

A new transaction from callback (APP\_RETVAL\_NEW\_TX ) involving master RX chaining using DMA, is not supported.

For master RX chaining using DMA interrupts (initiated using readChainedFlag in

smb\_protocol\_execute functions) ->

1. The application would receive SUCCESS\_RX\_CHAINED status for intermediate set of bytes. Application can continue to receive more bytes using dma interrupt chaining by updating *newTxParams->buffer\_ptr* with the buffer to receive additional bytes.

The buffer should indicate the number of bytes to read in its first location.

*smbAppRxBuffer[0] = rx\_chained\_pk\_size; // readCount*

*newTxParams->buffer\_ptr = &smbAppRxBuffer[0];*

*return APP\_RETVAL\_CHAINED\_RX****;***

* 1. Once the application decides that it needs to receive the last set of bytes, it should

return status APP\_RETVAL\_CHAINED\_RX\_LAST. Smbus driver would complete the

transaction and notify application with status SUCCESS\_RX.

**Example code for handling smbus master completion status:**

If previous transaction is success, callback can initiate new transaction after SMB\_APP\_TRANSACTION\_INTERVAL without releasing smbus resource to other applications.

In case of success transaction, it can go to next transaction after SMB\_APP\_TRANSACTION\_INTERVAL.

Case SUCCESS\_TX:

/\* This status will be returned on successful write protocols:

\* SMB\_SEND\_BYTE, SMB\_WRITE\_BYTE, SMB\_WRITE\_WORD, SMB\_WRITE\_BLOCK \*/

/\* Note: buffer\_ptr will be pointing to smbAppTxBuffer[] \*/

/\* Go for next smbus transaction after 300 ms \*/

kTaskSetWakeTime(SMB\_APP\_TRANSACTION\_INTERVAL, SMBUS\_TASK\_ID);

retVal=APP\_RETVAL\_NEW\_TX;

In case of success transaction, it can initiate next transaction from this callback immediately.

/\* Next smbus transaction can be initiated from this callback immediately

without releasing smbus resource to other applications \*/

/\* Initiating READ BYTE transaction\*/

smbAppRetryCount = 0x0;

smbAppTxBuffer[0] = SMB\_APP\_SLAVE\_ADDRESS;

smbAppTxBuffer[1] = READ\_BYTE\_ADDR;

smbAppTxBuffer[2] = SMB\_APP\_SLAVE\_ADDRESS;

newTxParams->buffer\_ptr = &smbAppTxBuffer[0];

newTxParams->pecEnable = FALSE;

newTxParams->smb\_protocol = SMB\_READ\_BYTE;

newTxParams->WriteCount = 3;

retVal=APP\_RETVAL\_NEW\_TX;

If transaction terminated with PEC error, callback could retry previous transaction or else initiate next transaction after SMB\_APP\_TRANSACTION\_INTERVAL.

Case ERROR\_PEC: // PEC Errors

if (smbAppRetryCount < SMB\_APP\_RETRY\_COUNT)

{

/\* Retrying transaction\*/

smbAppRetryCount++;

retVal=APP\_RETVAL\_RETRY;

}

else

{

/\* Initiate next transaction after SMB\_APP\_TRANSACTION\_INTERVAL \*/

kTaskSetWakeTime(SMB\_APP\_TRANSACTION\_INTERVAL, SMBUS\_TASK\_ID);

}

If transaction encountered BER\_NON\_TIMEOUT i.e bus error due to non timeout (Invalid START and STOP conditions), callback initiates a new transaction after SMB\_APP\_TRANSACTION\_INTERVAL.

Case ERROR\_BER\_NON\_TIMEOUT:

//Bus Error due to non timeouts (e.g. invalid START/STOP conditions)

/\* Go for next smbus transaction after 300 ms \*/

kTaskSetWakeTime(SMB\_APP\_TRANSACTION\_INTERVAL, SMBUS\_TASK\_ID);

When transaction is successful and all received protocols are success.

Case SUCCESS\_RX:

/\* This status will be returned on successful receive protocols: SMB\_RECEIVE\_BYTE, SMB\_READ\_BYTE, SMB\_READ\_WORD, SMB\_READ\_BLOCK \*/

/\* For SMB\_RECEIVE\_BYTE protocol buffer\_ptr[1] will contain the

received byte \*/

/\* For SMB\_READ\_BYTE protocol buffer\_ptr[3] will contain the

received byte \*/

/\* For SMB\_READ\_WORD protocol buffer\_ptr[3] & buffer\_ptr[4] will

contain the received word \*/

/\* For SMB\_READ\_BLOCK protocol buffer\_ptr[3] will contain the

read block data length \*/

## SLAVE APIs

### smb\_register\_slave

Function Header

**UINT8 smb\_register\_slave(const UINT8 channel, SLAVE\_FUNC\_PTR slaveFuncPtr)**

Description

This function registers a smbus slave application.

Note: Whenever a application expects data from smbus (i.e acts as slave) it needs to register using this function.The application function that is registered should only copy the packet from smbus buffer, it should not the process the data in that function.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| channel | SMB\_CHANNEL\_x – x is 1 to 4 |
| slaveFuncPtr | The application function to call on receiving a packet |

Outputs

STATUS\_OK on successful registration, else error status.

Example Usage

None

### smb\_deregister\_slave

Function Header

**UINT8 smb\_deregister\_slave(const UINT8 channel, SLAVE\_FUNC\_PTR slaveFuncPtr)**

Description

This function is used to de-register a smbus slave application.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| channel | SMB\_CHANNEL\_x – x is 1 to 4 |
| slaveFuncPtr | The application function pointer |

Outputs

STATUS\_OK on successful de-registration, else error status.

Example Usage

None

### smbApp\_slave\_callback

Function Header

UINT8 smbApp\_slave\_callback (BUFFER\_INFO \*buffer\_info,

UINT8 slaveTransmitFlag)

Description

This function is the callback function which is registered with the smbus driver for getting any notification for slave packets.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| buffer\_info | buffer\_info structure containing buffer pointer and data length |
| slaveTransmitFlag | TRUE if application needs to provide data for slave transmit phase |

Outputs

|  |  |
| --- | --- |
| *smbApp\_slave\_callback return values* | |
| STATUS\_BUFFER\_NOT\_DONE | Application is busy, driver will discard the packet |
| STATUS\_BUFFER\_DONE | Application processed the packet |
| STATUS\_BUFFER\_ERROR | Packet not meant for this application |

**Note:**

Application can read the received data from the buffer\_info->buffer\_ptr, the data length

is specified by buffer\_info->Datalen.

If application wants to transmit any data (based on slaveTransmitFlag) it should update

the data in buffer\_info->buffer\_ptr and update buffer\_info->Datalen and

buffer\_info->pecFlagFor transmitting more than 255 bytes during the slave Transmit phase, the application can indicate by setting buffer\_info->slaveXmitDoneFlag = FALSE. This will cause the driver to use dma chaining. Application would be then notified again through the callback.

In the last sequence of data, application should not set buffer\_info->slaveXmitDoneFlag = FALSE to complete the transfer.

## Handling PEC

If PEC is enabled there will be an additional byte transferred (for all smbus protocols) which will be the PEC byte.

Based on the value of the PEC byte and the data contents, the master or slave can validate the data received. Both master and slave would need to understand that PEC is being used and act accordingly for the extra byte.

PEC is always transmitted by the device sending data, so for all write protocols the master will send the PEC byte and the slave can validate the data received.

For read protocols, the slave would send the data and the PEC byte, in which case the master can validate the data received.

Enabling PEC in Master APIs

In smbus master API’s smb\_protocol\_execute and smb\_protocol\_execute\_blocking, if the application wants to read or transmit the PEC byte, it can set the pecFlag as TRUE, all other parameters will remain same.

If PEC is enabled, then for write operations, the master would transmit the extra PEC byte. The slave should be ready to receive the extra byte.

For read operations, the master would read the extra PEC byte and validate it. If the PEC is invalid master callback would indicate using status ERROR\_PEC.

PEC in Slave Callback

While acting as slave, if master sends PEC byte, then in the slave callback, buffer\_info->buffer\_ptr[] would have the additional PEC byte and buffer\_info->Datalen will be incremented by 1.

Buffer\_info->pecFlag would then indicate if the PEC is valid or not.

For read protocols, where the slave is transmitting data (slaveTransmitFlag = TRUE), then application can indicate to transmit the PEC byte by setting buffer\_info->pecFlag as TRUE

## buffer\_info details for smbus protocols

In the smbus slave callback the contents of buffer\_info->buffer\_ptr is listed below for each smbus protocols:

BLOCK WRITE

|  |  |
| --- | --- |
| *Block Write* | |
| buffer\_info->buffer\_ptr[0] | 7-Bit Slave address + Write Bit |
| buffer\_info->buffer\_ptr[1] | Command Code |
| buffer\_info->buffer\_ptr[2] | Block Write Data Length (N) |
| buffer\_info->buffer\_ptr[3] | Data 1 |
| buffer\_info->buffer\_ptr[4] | Data 2 |
| buffer\_info->buffer\_ptr[5] | Data 3 |
| buffer\_info->buffer\_ptr[ ] | Data … |
| buffer\_info->buffer\_ptr[(N+3)-1] | Data (N) |
| buffer\_info->Datalen | N+3 |
| slaveTransmitFlag | FALSE |

BLOCK READ

|  |  |
| --- | --- |
| *Block Read* | |
| buffer\_info->buffer\_ptr[0] | 7-Bit Slave address + Write Bit |
| buffer\_info->buffer\_ptr[1] | Command Code |
| buffer\_info->buffer\_ptr[2] | 7-Bit Slave Address + Read Bit |
| buffer\_info->Datalen | 3 |
| slaveTransmitFlag | TRUE |

Application would need to fill-in buffer\_info->buffer\_ptr with the data to send and update buffer\_info->Datalen with the number of bytes to transmit.

WORD WRITE

|  |  |
| --- | --- |
| *Word Write* | |
| buffer\_info->buffer\_ptr[0] | 7-Bit Slave address + Write Bit |
| buffer\_info->buffer\_ptr[1] | Command Code |
| buffer\_info->buffer\_ptr[2] | Data Byte Low |
| buffer\_info->buffer\_ptr[3] | Data Byte High |
| buffer\_info->Datalen | 4 |
| slaveTransmitFlag | FALSE |

WORD READ

|  |  |
| --- | --- |
| *Word Read* | |
| buffer\_info->buffer\_ptr[0] | 7-Bit Slave address + Write Bit |
| buffer\_info->buffer\_ptr[1] | Command Code |
| buffer\_info->buffer\_ptr[2] | 7-Bit Slave Address + Read Bit |
| buffer\_info->Datalen | 3 |
| slaveTransmitFlag | TRUE |

Application would need to fill-in buffer\_info->buffer\_ptr[0] and in buffer\_info->buffer\_ptr[1] with the data to send and update buffer\_info->Datalen as 2 (number of bytes to transmit)

BYTE WRITE

|  |  |
| --- | --- |
| *Byte Write* | |
| buffer\_info->buffer\_ptr[0] | 7-Bit Slave address + Write Bit |
| buffer\_info->buffer\_ptr[1] | Command Code |
| buffer\_info->buffer\_ptr[2] | Data Byte |
| buffer\_info->Datalen | 3 |
| slaveTransmitFlag | FALSE |

BYTE READ

|  |  |
| --- | --- |
| *Byte Read* | |
| buffer\_info->buffer\_ptr[0] | 7-Bit Slave address + Write Bit |
| buffer\_info->buffer\_ptr[1] | Command Code |
| buffer\_info->buffer\_ptr[2] | 7-Bit Slave Address + Read Bit |
| buffer\_info->Datalen | 3 |
| slaveTransmitFlag | TRUE |

Application would need to fill-in buffer\_info->buffer\_ptr[0] with the data byte to send and update buffer\_info->Datalen as 1 (number of bytes to transmit)

SEND BYTE

|  |  |
| --- | --- |
| *Send Byte* | |
| buffer\_info->buffer\_ptr[0] | 7-Bit Slave address + Write Bit |
| buffer\_info->buffer\_ptr[1] | Data Byte |
| buffer\_info->Datalen | 2 |
| slaveTransmitFlag | FALSE |

RECEIVE BYTE

|  |  |
| --- | --- |
| *Receive Byte* | |
| buffer\_info->buffer\_ptr[0] | 7-Bit Slave address + Read Bit |
| buffer\_info->Datalen | 1 |
| slaveTransmitFlag | TRUE |

Application would need to fill-in buffer\_info->buffer\_ptr[0] with the data byte to send and update buffer\_info->Datalen as 1 (number of bytes to transmit)

# ADC

CEC1302 has sixteen ADC channels, provides 10-bit adc channels, supports two interrupts one for single mode interrupt and the other is for repeat mode interrupt.

User can use the below API’s to initiate the ADC channel using adc\_init routine.

User has options to select single or repeat mode.

User shall use the below API’s with valid ADC Channel Number.

**ADC API’s**

adc\_init

adc\_gpio\_configure

**Number of ADC channels**

ADC0

ADC1

ADC2

ADC3

ADC4

ADC5

ADC6

ADC7

ADC8

ADC9

ADC10

ADC11

ADC12

ADC13

ADC14

ADC15

**ADC instance**

ADCx Control Register

ADCx Delay Register

ADCx Status Register

ADCx Single Register

ADCx Repeat Register

ADC Channelx Reading Register.

**ADC Peripheral Functions**

p\_adc\_singlemode\_status

p\_adc\_singlemode\_status\_clear

p\_adc\_repeatmode\_status

p\_adc\_repeatmode\_status\_clear

*p\_adc\_adc\_block\_reset*

p\_adc\_power\_save\_control

p\_adc\_repeatmode\_control

p\_adc\_singlemode\_control

p\_adc\_block\_control

p\_adc\_repeat\_delay\_set

p\_adc\_start\_delay\_set

p\_adc\_status\_register\_read

p\_adc\_status\_register\_clear

p\_adc\_single\_enable\_control

p\_adc\_repeat\_enable\_control

p\_adc\_raw\_value\_read

PCR Peripheral Functions

GPIO Peripheral Functions

Interrupt Peripheral Functions

## ADC API’s

The list of ADC APIs

* adc\_init
* adc\_gpio\_configure

### ADC\_init

Function Header

**void adc\_init(uint8\_t adc\_ch,**

**uint8\_t adc\_mode,**

**)**

Description

Configure the ADC channel with required configuration.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| adc\_ch | adc ch – ADCx\_ch (x => 0-15) |
| adc\_mode | Mode selection, single conversion mode or Repeat conversion mode.  0 – single mode  1 – Repeat mode |

Outputs

None

### adc\_gpio\_configure

Function Header

**void adc\_gpio\_configure(uint8\_t adc\_ch )**

Description

Configure the GPIO control register for the requested ADC channel.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| adc\_ch | adc ch – ADCx\_ch (x => 0-15) |

Outputs

None

## ADC Peripheral Functions

The list of ADC peripheral functions are listed below:

* p\_adc\_singlemode\_status
* p\_adc\_singlemode\_status\_clear
* p\_adc\_repeatmode\_status
* p\_adc\_repeatmode\_status\_clear
* *p\_adc\_adc\_block\_reset*
* p\_adc\_power\_save\_control
* p\_adc\_repeatmode\_control
* p\_adc\_singlemode\_control
* p\_adc\_block\_control
* p\_adc\_repeat\_delay\_set
* p\_adc\_start\_delay\_set
* p\_adc\_status\_register\_read
* p\_adc\_status\_register\_clear
* p\_adc\_single\_enable\_control
* p\_adc\_repeat\_enable\_control
* p\_adc\_raw\_value\_read

### p\_adc\_singlemode\_status

Function Header

**Uint8\_t p\_adc\_singlemode\_status(void)**

Description

Returns the single mode conversion-complete status.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| Void | None. |

Outputs

* 1. – conversion not complete
     1. - Conversion complete.

### p\_adc\_singlemode\_status\_clear

Function Header

**void p\_adc\_singlemode\_status\_clear(void)**

Description

Clears the single mode conversion-complete status.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| void | None. |

Outputs

None.

### p\_adc\_repeatmode\_status

Function Header

**uint8\_t p\_adc\_repeatmode\_status(void)**

Description

Returns the repeat mode conversion-complete status.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| Void | None. |

Outputs

0 – conversion not complete

1 - Conversion complete.

### p\_adc\_repeatmode\_status\_clear

Function Header

**void p\_adc\_repeatmode\_status\_clear(void)**

Description

Clear the repeat mode conversion-complete status.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| Void | None. |

Outputs

None.

### p\_adc\_adc\_block\_reset

Function Header

**void p\_adc\_block\_reset(uint8\_t control)**

Description

Reset or move out of Reset.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| control | 1 – reset the block  0 – Move out of the reset. |

Outputs

None.

### p\_adc\_power\_save\_control

Function Header

**Void p\_adc\_power\_save\_control(uint8\_t control)**

Description

Enable or Disable the power saving feature.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| Control | 0 – enable power saving  1 – Disable power saving feature |

Outputs

None.

### p\_adc\_repeatmode\_control

Function Header

**void p\_adc\_repeatmode\_control(uint8\_t control)**

Description

Start or Stop conversion.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| Control | 1 – Start conversion  0 - Stop Conversion |

Outputs

None.

### p\_adc\_singlemode\_control

Function Header

**void p\_adc\_singlemode\_control(uint8\_t control)**

Description

Start or Stop the single mode conversion.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| Control | 1 – Start Conversion  0 - Stop Conversion. |

Outputs

None.

### p\_adc\_block\_control

Function Header

**void p\_adc\_block\_control (uint8\_t control)**

Description

Activate or De-activate the ADC Block.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| Control | 1 - Activate  0 – De-Activate the ADC Block. |

Outputs

None.

### p\_adc\_repeat\_delay\_set

Function Header

**void p\_adc\_repeat\_delay\_set(uint16\_t delay)**

Description

Load the required delay between repeat conversions.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| Delay | 1 unit is 40 microseconds;  max possible delay is 2.6 seconds. |

Outputs

None.

### p\_adc\_start\_delay\_set

Function Header

**void p\_adc\_start\_delay\_set (uint16\_t delay)**

Description

Load the required delay before start conversion.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| delay | 1 unit is 40 microseconds;  max possible delay is 2.6 seconds. |

Outputs

None.

### p\_adc\_status\_register\_read

Function Header

**Uint16\_t p\_adc\_status\_register\_read(uint8\_t adc\_ch)**

Description

Reads the conversion completion status of Individual channels.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| adc\_ch | 0 – 15 channels. |

Outputs

0 – conversion not complete

1 - Conversion complete.

### p\_adc\_status\_register\_clear

Function Header

**void p\_adc\_status\_register\_clear(void)**

Description

Clears the conversion completion status of all channels. Its not possible to clear individual channels.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| Void | None. |

Outputs

None.

### p\_adc\_single\_enable\_control

Function Header

**void p\_adc\_single\_enable\_control(uint8\_t adc\_ch, uint8\_t control)**

Description

Enable or Disable the required Channels in Single mode.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| adc\_ch | Channel 0 – 15. |
| Control | 1 – Enable  0 – Disable |

Outputs

None.

### p\_adc\_repeat\_enable\_control

Function Header

**void p\_adc\_repeat\_enable\_control(uint8\_t adc\_ch, uint8\_t control)**

Description

Enable or Disable the required Channels in Repeat mode.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| adc\_ch | Channel 0 – 15. |
| Control | 1 – Enable  0 – Disable |

Outputs

None.

### p\_adc\_raw\_value\_read

Function Header

**Uint16\_t p\_adc\_raw\_value\_read(uint8\_t adc\_ch)**

Description

Read converted adc raw value from Individual channels.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| Adc\_ch | Adc Channels 0 – 15. |

Outputs

ADC raw value of Individual Channel.

# PCR – Power, Clocks, Reset

## PCR APIs

The list of PCR APIs

1. Functions to program Sleep Enable, CLK Reqd Status, Reset Enable for a block

* pcr\_sleep\_enable
* pcr\_clock\_reqd\_status\_get
* pcr\_reset\_enable

2. Functions for entering low power modes

* pcr\_all\_blocks\_sleep
* pcr\_all\_blocks\_wake
* pcr\_system\_sleep

Other PCR APIs

pcr\_power\_reset\_status\_read

pcr\_power\_reset\_ctrl\_read

pcr\_pwr\_reset\_ctrl\_pwr\_inv\_set\_clr

pcr\_pwr\_reset\_ctrl\_host\_rst\_set\_clr

pcr\_system\_reset\_set

pcr\_pke\_clock\_write

pcr\_pke\_clock\_read

pcr\_osc\_cal\_write

pcr\_osc\_cal\_read

*Functions to program Sleep Enable, CLK Reqd Status, Reset Enable for a block*

### pcr\_sleep\_enable

Function Header

void pcr\_sleep\_enable(uint32\_t pcr\_block\_id, uint8\_t set\_clr\_flag);

Description

Sets or Clears block specific bit in PCR Sleep Enable Register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| pcr\_block\_id | pcr block id |
| set\_clr\_flag | Flag to set (1) or clear (0) bit in the PCR Sleep Enable Register |

Outputs

None

### pcr\_clock\_reqd\_status\_get

Function Header

uint8\_t pcr\_clock\_reqd\_status\_get (uint32\_t pcr\_block\_id);

Description

Get Clock Required Status for the block

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| pcr\_block\_id | pcr block id |

Outputs

1 if Clock Required Status set, else 0

### pcr\_reset\_enable

Function Header

void pcr\_sleep\_enable(uint32\_t pcr\_block\_id, uint8\_t set\_clr\_flag);

Description

Sets or Clears Reset Enable register bit for the block

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| pcr\_block\_id | pcr block id |
| set\_clr\_flag | Flag to set (1) or clear (0) bit in the PCR Reset Enable Register |

Outputs

None

*Functions for entering low power modes*

### p\_pcr\_all\_blocks\_sleep

Function Header

void pcr\_all\_blocks\_sleep(void);

Description

Instructs all blocks to sleep by setting the Sleep Enable bits

Inputs

None

Outputs

None

### p\_pcr\_all\_blocks\_wake

Function Header

void pcr\_all\_blocks\_wake(void);

Description

Clears the Sleep Enable bits for all blocks

Inputs

None

Outputs

None

### pcr\_system\_sleep

Function Header

void pcr\_system\_sleep (uint8\_t sleep\_mode);

Description

Programs required sleep mode in System Sleep Control Register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| sleep\_mode | Sleep mode – see enum SYSTEM\_SLEEP\_MODES |

Outputs

None

### pcr\_power\_reset\_status\_read

Function Header

uint16\_t pcr\_power\_reset\_status\_read(void);

Description

Reads the value of Power Reset Status Register

Inputs

None

Outputs

Power reset status register value

### pcr\_power\_reset\_ctrl\_read

Function Header

uint16\_t pcr\_power\_reset\_ctrl\_read (void);

Description

Reads the value of Power Reset Control Register

Inputs

None

Outputs

Power reset control register value

### pcr\_pwr\_reset\_ctrl\_pwr\_inv\_set\_clr

Function Header

void pcr\_pwr\_reset\_ctrl\_pwr\_inv\_set\_clr(uint8\_t set\_clr);

Description

Set the value of PWR\_INV bit to 1 or 0

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| set\_clr | 1 to set the PWR\_INV bit  0 to clear the PWR\_INV bit |

Outputs

None

### pcr\_pwr\_reset\_ctrl\_host\_rst\_set\_clr

Function Header

void pcr\_pwr\_reset\_ctrl\_host\_rst\_set\_clr(uint8\_t set\_clr);

Description

Set the value of HOST\_RESET bit to 1 or 0

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| set\_clr | 1 to set the HOST\_RESET bit  0 to clear the HOST\_RESET bit |

Outputs

None

### pcr\_system\_reset\_set

Function Header

void pcr\_system\_reset\_set (uint8\_t set\_clr);

Description

Set the system reset bit in PCR block

Inputs

None

Outputs

None

### pcr\_pke\_clock\_write

Function Header

void pcr\_pke\_clock\_write (uint8\_t pke\_clk\_val);

Description

Write the PKE Clock value in PKE clock register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| pke\_clk\_val | Clock value to be written to PKE Clock register |

Outputs

None

### pcr\_pke\_clock\_read

Function Header

uint8\_t pcr\_pke\_clock\_read (void);

Description

Read the PKE clock register

Inputs

None

Outputs

PKE clock register value

### pcr\_osc\_cal\_write

Function Header

void pcr\_osc\_cal\_write (uint8\_t pke\_clk\_val);

Description

Write the calibration value in OSC calibration register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| pke\_clk\_val | Calibration value 1 or 0 |

Outputs

None

### pcr\_pke\_clock\_read

Function Header

uint8\_t pcr\_osc\_cal\_read (void);

Description

Read the OSC calibration register

Inputs

None

Outputs

OSC calibration value

## PCR Peripheral Functions

The list of PCR Peripheral Functions

Generic functions to write and read 32-bit values from PCR Registers

* p\_pcr\_reg\_write
* p\_pcr\_reg\_read

Functions to set, clear and get bits in PCR Register

* p\_pcr\_reg\_set
* p\_pcr\_reg\_clr
* p\_pcr\_reg\_get
* p\_pcr\_reg\_update

Functions to operate on System Sleep Control Register

* p\_pcr\_system\_sleep\_ctrl\_write
* p\_pcr\_system\_sleep\_ctrl\_read

Function to program to CLK Divide Value

* p\_pcr\_processor\_clk\_ctrl\_write

Function to program the Slow Clock Control Register

* p\_pcr\_slow\_clk\_ctrl\_write

Function to read the Oscillator Lock Status

* p\_pcr\_oscillator\_lock\_sts\_get

Function to read the oscillator ID register

* p\_pcr\_oscillator\_id\_reg\_read

Functions to read various power status’ in Power reset status register

* p\_pcr\_pwr\_reset\_vcc\_reset\_sts\_get
* p\_pcr\_pwr\_reset\_host\_reset\_sts\_get
* p\_pcr\_pwr\_reset\_vbat\_reset\_sts\_get
* p\_pcr\_pwr\_reset\_vbat\_reset\_sts\_clr
* p\_pcr\_pwr\_reset\_vtr\_reset\_sts\_get
* p\_pcr\_pwr\_reset\_vtr\_reset\_sts\_clr
* p\_pcr\_pwr\_reset\_32K\_active\_sts\_get
* p\_pcr\_pwr\_reset\_pciclk\_active\_sts\_get
* p\_pcr\_pwr\_reset\_espilk\_active\_sts\_get
* p\_pcr\_pwr\_reset\_sts\_get

Functions to read/write various bits in Power Reset Control register

* p\_pcr\_pwr\_reset\_ctrl\_read
* p\_pcr\_pwr\_reset\_ctrl\_pwr\_inv\_set\_clr
* p\_pcr\_pwr\_reset\_ctrl\_host\_rst\_set\_clr

Function to set the system reset bit in system reset register

* p\_pcr\_system\_reset\_set

Functions to read/write PKE clock register

* p\_pcr\_pke\_clock\_write
* p\_pcr\_pke\_clock\_read

Functions to read/write oscillator calibration register

* p\_pcr\_osc\_cal\_write
* p\_pcr\_osc\_cal\_read

*Generic functions to write and read 32-bit values from PCR Registers*

### p\_pcr\_reg\_write

Function Header

void p\_pcr\_reg\_write(uint8\_t pcr\_reg\_id, uint32\_t value);

Description

Write 32-bit value in the PCR Register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| pcr\_reg\_id | PCR Register ID |
| value | 32-bit value |

Outputs

None

### p\_pcr\_reg\_read

Function Header

uint32\_t p\_pcr\_reg\_read(uint8\_t pcr\_reg\_id);

Description

Reads 32-bit value from the PCR Register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| pcr\_reg\_id | PCR Register ID |

Outputs

value – 32-bit value

*Functions to set, clear and get bits in PCR Register*

### p\_pcr\_reg\_set

Function Header

void p\_pcr\_reg\_set(uint8\_t pcr\_reg\_id, uint32\_t bit\_mask);

Description

Sets bits in a PCR Register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| pcr\_reg\_id | PCR Register ID |
| uint32\_bit\_mask | Bit mask of bits to Set |

Outputs

None

### p\_pcr\_reg\_clr

Function Header

void p\_pcr\_reg\_clr(uint8\_t pcr\_reg\_id, uint32\_t bit\_mask);

Description

Clears bits in a PCR Register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| pcr\_reg\_id | PCR Register ID |
| uint32\_bit\_mask | Bit mask of bits to clear |

Outputs

None

### p\_pcr\_reg\_get

Function Header

uint32\_t p\_pcr\_reg\_get(uint8\_t pcr\_reg\_id, uint32\_t bit\_mask);

Description

Read bits in a PCR Register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| pcr\_reg\_id | PCR Register ID |
| uint32\_bit\_mask | Bit mask of bits to read |

Outputs

None

### p\_pcr\_reg\_update

Function Header

void p\_pcr\_reg\_update(uint8\_t pcr\_reg\_id, uint32\_t bit\_mask, uint8\_t set\_clr\_flag);

Description

Sets or Clears bits in a PCR Register.

This function calls p\_pcr\_reg\_set / p\_pcr\_reg\_clr

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| pcr\_reg\_id | PCR Register ID |
| uint32\_bit\_mask | Bit mask of bits to update |
| set\_clr\_flag | Flag to set (1) or clear (0) bits in the PCR Register |

Outputs

None

*Functions to operate on System Sleep Control Register*

### p\_pcr\_system\_sleep\_ctrl\_write

Function Header

void p\_pcr\_system\_sleep\_ctrl\_write(uint8\_t sleep\_value);

Description

Writes required sleep mode in System Sleep Control Register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| sleep\_value | System Sleep control value – [D2, D1, D0] |

Outputs

None

### p\_pcr\_system\_sleep\_ctrl\_read

Function Header

uint8\_t p\_pcr\_system\_sleep\_ctrl\_read(void);

Description

Reads the System Sleep Control PCR Register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| value | byte 0 of the system sleep control PCR register |

Outputs

None

*Function to program the CLK Divide Value*

### p\_pcr\_processor\_clk\_ctrl\_write

Function Header

void p\_pcr\_processor\_clk\_ctrl\_write(uint8\_t clk\_divide\_value);

Description

Writes the clock divide value in the Processor Clock Control Register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| clk\_divide\_value | clk divide values, valid values in enum PROCESSOR\_CLK\_DIVIDE\_VALUE |

Outputs

None

*Function to program the Slow Clock Control Register*

### p\_pcr\_slow\_clk\_ctrl\_write

Function Header

void p\_pcr\_slow\_clk\_ctrl\_write(uint8\_t slow\_clk\_divide\_value);

Description

Write the slow clock divide value in the Slow Clock Control Register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| slow\_clk\_divide\_value | slow clk divide value |

Outputs

None

*Function to read the Oscillator Lock Status*

### p\_pcr\_oscillator\_lock\_sts\_get

Function Header

uint8\_t p\_pcr\_oscillator\_lock\_sts\_get(void);

Description

Reads the Oscillator Lock status bit in the Oscillator ID Register

Inputs

None

Outputs

1 if Status bit is set, else 0

*Function to read the oscillator ID register*

### p\_pcr\_oscillator\_id\_reg\_read

Function Header

uint16\_t p\_pcr\_oscillator\_id\_reg\_read(void);

Description

Reads the Oscillator ID in the Oscillator ID Register

Inputs

None

Outputs

Oscillator id value

*Functions to read various power status’ in Power Reset status register*

### p\_pcr\_pwr\_reset\_vcc\_reset\_sts\_get

Function Header

uint8\_t p\_pcr\_pwr\_reset\_vcc\_reset\_sts\_get(void);

Description

Reads the VCC Reset Status bit in the Power Reset Status Register

Inputs

None

Outputs

1 if Status bit is set, else 0

### p\_pcr\_pwr\_reset\_host\_reset\_sts\_get

Function Header

uint8\_t p\_pcr\_pwr\_reset\_sio\_reset\_sts\_get(void);

Description

Reads the Host Reset Status bit in the Power Reset Status Register

Inputs

None

Outputs

1 if Status bit is set, else 0

### p\_pcr\_pwr\_reset\_vbat\_reset\_sts\_clr

Function Header

void p\_pcr\_pwr\_reset\_vbat\_reset\_sts\_clr(void);

Description

Clears the VBAT Reset Status bit in the Power Reset Status Register

Inputs

None

Outputs

None

### p\_pcr\_pwr\_reset\_vtr\_reset\_sts\_get

Function Header

uint8\_t p\_pcr\_pwr\_reset\_vtr\_reset\_sts\_get(void);

Description

Reads the VTR Reset Status bit in the Power Reset Status Register

Inputs

None

Outputs

1 if Status bit is set, else 0

### p\_pcr\_pwr\_reset\_vtr\_reset\_sts\_clr

Function Header

void p\_pcr\_pwr\_reset\_vtr\_reset\_sts\_clr(void);

Description

Clears the VTR Reset Status bit in the Power Reset Status Register

Inputs

None

Outputs

None

### p\_pcr\_pwr\_reset\_32K\_active\_sts\_get

Function Header

uint8\_t p\_pcr\_pwr\_reset\_32K\_active\_sts\_get(void);

Description

Reads the 32K Active Status bit in the Power Reset Status Register

Inputs

None

Outputs

1 if Status bit is set, else 0

### p\_pcr\_pwr\_reset\_pciclk\_active\_sts\_get

Function Header

uint8\_t p\_pcr\_pwr\_reset\_pciclk\_active\_sts\_get(void);

Description

Reads the PCICLK\_ACTIVE status bit in the Power Reset Status Register

Inputs

None

Outputs

1 if Status bit is set, else 0

### p\_pcr\_pwr\_reset\_espiclk\_active\_sts\_get

Function Header

uint8\_t p\_pcr\_pwr\_reset\_espiclk\_active\_sts\_get(void);

Description

Reads the ESPICLK\_ACTIVE status bit in the Power Reset Status Register

Inputs

None

Outputs

1 if Status bit is set, else 0

### p\_pcr\_pwr\_reset\_ sts\_get

Function Header

uint16\_t p\_pcr\_pwr\_reset\_sts\_get(void);

Description

Reads the Power Reset Status Register

Inputs

None

Outputs

Power reset status register value

*Functions for Power Reset Control Register*

### p\_pcr\_pwr\_reset\_ctrl\_read

Function Header

uint16\_t p\_pcr\_pwr\_reset\_ctrl\_read(void);

Description

Reads the Power Reset Control Register

Inputs

None

Outputs

Power reset control register value

### p\_pcr\_pwr\_reset\_ctrl\_pwr\_inv\_set\_clr

Function Header

void p\_pcr\_ pwr\_reset\_ctrl\_pwr\_inv\_set\_clr (uint8\_t set\_clr);

Description

Sets/Clears the PWR\_INV bit in the Power Reset Control Register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| set\_clr | 1 Set bit; 0 – Clear the bit |

Outputs

None

### p\_pcr\_pwr\_reset\_ctrl\_host\_rst\_set\_clr

Function Header

void p\_pcr\_ pwr\_reset\_ctrl\_host\_rst\_set\_clr (uint8\_t set\_clr);

Description

Sets/Clears the HOST RESET bit in the Power Reset Control Register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| set\_clr | 1 Set bit; 0 – Clear the bit |

Outputs

None

*Functions for System Reset Register*

### p\_pcr\_system\_reset\_set

Function Header

void p\_pcr\_system\_reset\_set(void);

Description

Sets the system reset bit in the system reset register

Inputs

None

Outputs

None

### p\_pcr\_pke\_clock\_write

Function Header

void p\_pcr\_pke\_clock\_write (uint8\_t pke\_clk\_val);

Description

Write the PKE Clock value in PKE clock register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| pke\_clk\_val | Clock value to be written to PKE Clock register |

Outputs

None

### p\_pcr\_pke\_clock\_read

Function Header

uint8\_t p\_pcr\_pke\_clock\_read (void);

Description

Read the PKE clock register

Inputs

None

Outputs

PKE clock register value

### p\_pcr\_osc\_cal\_write

Function Header

void p\_pcr\_osc\_cal\_write (uint8\_t pke\_clk\_val);

Description

Write the calibration value in OSC calibration register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| pke\_clk\_val | Calibration value 1 or 0 |

Outputs

None

### p\_pcr\_pke\_clock\_read

Function Header

uint8\_t p\_pcr\_osc\_cal\_read (void);

Description

Read the OSC calibration register

Inputs

None

Outputs

OSC calibration value

# TACH

CEC1302 has three TACH input channels, this block is designed to monitor fans at fan speeds from 100RPMs to 30000 RPMs. It supports two modes, one is free running counter and the second one is clock pulses per revolution.

**Tach Peripheral Functions**

p\_tach\_outoflimit\_intp\_control

p\_tach\_control

p\_tach\_filter\_control

p\_tach\_reading\_mode\_select

*p\_tach\_edges\_configure*

p\_tach\_count\_ready\_inpt\_control

p\_tach\_toggle\_inpt\_control

p\_tach\_counter\_register\_read

p\_tach\_outoflimit\_status\_read

p\_tach\_outoflimit\_status\_clear

p\_tach\_pin\_status\_read

p\_tach\_toggle\_status\_read

p\_tach\_toggle\_status\_clear

p\_tach\_count\_ready\_status\_read

p\_tach\_high\_limit\_register\_write

p\_tach\_high\_limit\_register\_read

p\_tach\_low\_limit\_register\_write

p\_tach\_low\_limit\_register\_read

**Tach instance**

Tachx Control Register

Tachx Status Register

Tachx High Limit Register

Tachx Low limit Register

**Number of Tach channels**

Tach0

Tach1

Tach2

**Tach API’s**

tach\_init

tach\_limits\_init

tach\_pulse\_counter\_read

tach\_sleep\_enable

tach\_sleep\_disable

tach\_gpio\_configure

PCR Peripheral Functions

GPIO Peripheral Functions

Interrupt Peripheral Functions

## Tach API’s

The list of Tach APIs

* tach\_init
* tach\_limits\_init
* tach\_pulse\_counter\_read
* tach\_sleep\_enable
* tach\_sleep\_disable
* tach\_gpio\_configure

### tach\_init

Function Header

**void tach\_init(uint8\_t tach\_ch,**

**uint8\_t tach\_mode,**

**uint8\_t tach\_read\_mode,**

**uint8\_t tach\_edges**

**)**

Description

Configure the Tach channel with required configuration.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| tach\_ch | tach ch – tachx\_ch (x => 0-2) |
| tach\_mode | Mode selection,  0 – Free running mode  1 – Fan Revolutions mode |
| Tach\_read\_mode | Transition detection method  0 – Transition from low to High state  1 - Rising edge |
| Tach\_edges | Edges per one revolutions  0 – 2 Tach edges  1 – 3 tach edges  2 – 5 tach edges  3 - 9 tach edges |

Outputs

None

### tach\_limits\_init

Function Header

**void tach\_limits\_init(uint8\_t tach\_ch,**

**uint16\_t low\_limits,**

**uint16\_t high\_limits,**

**uint8\_t OutOfLimit\_interrupt\_control**

**)**

Description

Configure the High and Low Limit Registers.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| tach\_ch | tach ch – tachx\_ch (x => 0-2) |
| Low\_Limits | Low limit value |
| High\_Limits | High Limit value |
| Out of limit detect interrupt | 1 – Enable out of limit detect nterrupt  0 – Disable out of limit detect interrupt |

Outputs

None

### tach\_pulse\_counter\_read

Function Header

**void tach\_pulse\_counter\_read(uint8\_t tach\_ch,**

**uint16\_t Previous\_pulse\_count**

**)**

Description

Reads the tach pulse count from free running counter.

User has to read at least once before it overruns the total pulse count of 0xFFFF count.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| tach\_ch | tach ch – tachx\_ch (x => 0-2) |
| Previous pulse count | This is required to check the over flow and get the absolute count. |

Outputs

None

### tach\_sleep\_enable

Function Header

**void tach\_sleep\_enable(uint8\_t tach\_ch)**

Description

Enables the sleep mode of Tach Block channel.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| tach\_ch | tach ch – tachx\_ch (x => 0-2) |

Outputs

None

### tach\_sleep\_disable

Function Header

**void tach\_sleep\_disable(uint8\_t tach\_ch)**

Description

Disables the sleep mode of Tach Block channel.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| tach\_ch | tach ch – tachx\_ch (x => 0-2) |

Outputs

None

### tach\_gpio\_configure

Function Header

**void tach\_gpio\_configure(uint8\_t tach\_ch )**

Description

Configure the GPIO pin for Tach channel.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| tach\_ch | tach ch – tachx\_ch (x => 0-2) |

Outputs

None

## Tach Peripheral Functions

The list of Tach peripheral functions are listed below:

* p\_tach\_outoflimit\_intp\_control
* p\_tach\_control
* p\_tach\_filter\_control
* p\_tach\_reading\_mode\_select
* *p\_tach\_edges\_configure*
* p\_tach\_count\_ready\_inpt\_control
* p\_tach\_toggle\_inpt\_control
* p\_tach\_counter\_register\_read
* p\_tach\_outoflimit\_status\_read
* p\_tach\_outoflimit\_status\_clear
* p\_tach\_pin\_status\_read
* p\_tach\_toggle\_status\_read
* p\_tach\_toggle\_status\_clear
* p\_tach\_count\_ready\_status\_read
* p\_tach\_high\_limit\_register\_write
* p\_tach\_high\_limit\_register\_read
* p\_tach\_low\_limit\_register\_write
* p\_tach\_low\_limit\_register\_read

### p\_tach\_outoflimit\_intp\_control

Function Header

**void p\_tach\_outoflimit\_intp\_control(uint8\_t tach\_ch, uint8\_t control)**

Description

Configure the interrupt for out of limit detection.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| tach\_ch | tach ch – tachx\_ch (x => 0-2) |
| Control | 0 – Disable  1 — Enable |

Outputs

None

### p\_tach\_control

Function Header

**void p\_tach\_control(uint8\_t tach\_ch, uint8\_t control)**

Description

Enable or Disable Tach channel Block.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| tach\_ch | tach ch – tachx\_ch (x => 0-2) |
| Control | 0 – Disable  1 — Enable |

Outputs

None

### p\_tach\_filter\_control

Function Header

**void p\_tach\_filter\_control(uint8\_t tach\_ch, uint8\_t control)**

Description

This filter is used to remove high frequency glitches from Tach input. When this filter enabled, tach input pulses less than two 100kHz clock periods wide get filtered.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| tach\_ch | tach ch – tachx\_ch (x => 0-2) |
| Control | 0 – Filter Disabled  1 – Filter Enabled |

Outputs

None

### p\_tach\_reading\_mode\_select

Function Header

**void p\_tach\_reading\_mode\_select(uint8\_t tach\_ch, uint8\_t mode)**

Description

Select the pulse transition detection mode.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| tach\_ch | tach ch – tachx\_ch (x => 0-2) |
| Mode | 0 – Low state to High State transition  1 – Rising edge detection |

Outputs

None

### p\_tach\_edges\_configure

Function Header

**void p\_tach\_edges\_configure(uint8\_t tach\_ch, uint8\_t tach\_edges)**

Description

A tach signal is a square wave with a 50% duty cycle. Typically, two tach periods represents one revolution of the fan. A tach period consists of three tach edges.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| tach\_ch | tach ch – tachx\_ch (x => 0-2) |
| Edges | 0 – 2 tach edges  1 - 3 tach edges  2 – 5 tach edges  3 – 9 tach edges |

Outputs

None

### p\_tach\_count\_ready\_inpt\_control

Function Header

**void p\_tach\_count\_ready\_inpt\_control(uint8\_t tach\_ch, uint8\_t control)**

Description

Enable or Disable Count Ready interrupt.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| tach\_ch | tach ch – tachx\_ch (x => 0-2) |
| Control | 0 – Count Ready interrupt disabled  1 – Count Ready interrupt Enabled. |

Outputs

None

### p\_tach\_toggle\_inpt\_control

Function Header

**void p\_tach\_toggle\_inpt\_control(uint8\_t tach\_ch, uint8\_t control)**

Description

Enable or Disable the tach input toggle interrupt.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| tach\_ch | tach ch – tachx\_ch (x => 0-2) |
| Control | 0 – Tach input toggle interrupt Disabled  1 – Tach input toggled interrupt Enabled |

Outputs

None

### p\_tach\_counter\_register\_read

Function Header

**Uint16\_t p\_tach\_counter\_register\_read(uint8\_t tach\_ch)**

Description

Read the tach pulses counter register value.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| tach\_ch | tach ch – tachx\_ch (x => 0-2) |

Outputs

Returns the counter value

### p\_tach\_outoflimit\_status\_read

Function Header

**Uint8\_t p\_tach\_outoflimit\_status\_read(uint8\_t tach\_ch)**

Description

Reads the Out of Limit status.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| tach\_ch | tach ch – tachx\_ch (x => 0-2) |

Outputs

0 – Tach reading is within limits.

1 – Tach reading is out of Limits.

### p\_tach\_outoflimit\_status\_clear

Function Header

**void p\_tach\_outoflimit\_status\_clear(uint8\_t tach\_ch)**

Description

Clear the Out of Limit Status.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| tach\_ch | tach ch – tachx\_ch (x => 0-2) |

Outputs

None

### p\_tach\_pin\_status\_read

Function Header

**Uint8\_t p\_tach\_pin\_status\_read(uint8\_t tach\_ch)**

Description

Reads the tach pin status.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| tach\_ch | tach ch – tachx\_ch (x => 0-2) |

Outputs

0 – Tach input is Low

1 – Tach input is High.

### p\_tach\_toggle\_status\_read

Function Header

**Uint8\_t p\_tach\_toggle\_status\_read(uint8\_t tach\_ch)**

Description

Reads the Tach input toggle status.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| tach\_ch | tach ch – tachx\_ch (x => 0-2) |

Outputs

0 – Tach input is stable

1 – Tach input got toggled.

### p\_tach\_toggle\_status\_clear

Function Header

**void p\_tach\_toggle\_status\_clear(uint8\_t tach\_ch)**

Description

Clears the Toggle status bit.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| tach\_ch | tach ch – tachx\_ch (x => 0-2) |

Outputs

None

### p\_tach\_count\_ready\_status\_read

Function Header

**Uint8\_t p\_tach\_outoflimit\_intp\_control(uint8\_t tach\_ch)**

Description

Reads the status of Count Ready.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| tach\_ch | tach ch – tachx\_ch (x => 0-2) |

Outputs

0 – Reading not ready

1 – Reading Ready

### p\_tach\_high\_limit\_register\_write

Function Header

**void p\_tach\_high\_limit\_register\_write(uint8\_t tach\_ch, uint16\_t High\_limit)**

Description

Set the limit in High limit register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| tach\_ch | tach ch – tachx\_ch (x => 0-2) |
| High Limit | High limit value |

Outputs

None

### p\_tach\_high\_limit\_register\_read

Function Header

**Uint16\_t p\_tach\_high\_limit\_register\_read(uint8\_t tach\_ch)**

Description

Read the limit set in High Limit register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| tach\_ch | tach ch – tachx\_ch (x => 0-2) |

Outputs

Returns the value set in High limit register.

### p\_tach\_low\_limit\_register\_write

Function Header

**void p\_tach\_low\_limit\_register\_write(uint8\_t tach\_ch, uint16\_t Low\_limit)**

Description

Set the value in Low Limit Register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| tach\_ch | tach ch – tachx\_ch (x => 0-2) |
| Low Limit | Value to set in Low Limit Register |

Outputs

None

### p\_tach\_low\_limit\_register\_read

Function Header

**Uint16\_t p\_tach\_low\_limit\_register\_read(uint8\_t tach\_ch)**

Description

Reads the value set in Low Limit register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| tach\_ch | tach ch – tachx\_ch (x => 0-2) |

Outputs

Value set in Low Limit Register.

# LED

LED API’s

led\_pins\_init

led\_control

led\_toggle

led\_blink

led\_as\_general\_pwm

led\_breath

LED Peripheral Functions

p\_led\_configuration\_reg\_set

p\_led\_configuration\_reg\_get

p\_led\_control\_set

p\_led\_clk\_src\_set

p\_led\_sync\_set

p\_led\_pwm\_size\_set

p\_led\_update\_enable\_set

p\_led\_wdt\_reset

p\_led\_wdt\_reload

p\_led\_symmetry\_set

p\_led\_limits\_set

p\_led\_limits\_get

p\_led\_delay\_set

p\_led\_delay\_get

p\_led\_duty\_cycle\_set

p\_led\_prescalar\_set

p\_led\_stepsize\_set

p\_led\_updateInterval\_set

p\_led\_output\_delay\_set

LED registers

LED configuration register

LED limits register

LED delay register

LED update step size register

LED update interval register

LED output delay register

LED instances

LED 0

LED 1

LED 2

LED 3

## LED APIs

The list of LED APIs

* led\_pins\_init
* led\_control
* led\_toggle
* led\_blink
* led\_as\_general\_pwm
* led\_breath

### led\_pins\_init

Function Header

**uint8\_t gpio\_init( uint8\_t led\_id )**

Description

Initializes the gpio pin of the specified LED hardware instance for LED functionality and resets that LED hardware block.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****led\_id**** | 0-based LED ID |

Outputs

None

### led\_control

Function Header

**void led\_control(uint8\_t led\_id, enum LED\_CONTROL led\_state )**

Description

Drives the output pin of the specified LED instance high or low.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****led\_id**** | 0-based LED ID |
| ****led\_state**** | Desired state of the LED pin  LED\_OFF  LED\_ON |

Outputs

None

### led\_toggle

Function Header

**void led\_toggle( uint8\_t led\_id )**

Description

Toggles the output pin of the specified LED hardware instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****led\_id**** | 0-based LED ID |

Outputs

None

### led\_blink

Function Header

**void led\_blink( uint8\_t led\_id, uint8\_t duty\_cycle, float frequency )**

Description

Enables the hardware blinking of the specified LED instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****led\_id**** | 0-based LED ID |
| ****duty\_cycle**** | Desired duty cycle for the blinking  0% (0x00) to 100% (0xFF) |
| ****frequency**** | Desired frequency for blinking  0.03125Hz to 128Hz |

Outputs

None

### led\_as \_general\_pwm

Function Header

**void led\_as\_general\_pwm( uint8\_t led\_id, uint8\_t duty\_cycle, float frequency, uint8\_t wdt\_val )**

Description

Configures the specified LED instance to function as a general purpose 8-bit pwm.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****led\_id**** | 0-based LED ID |
| ****duty\_cycle**** | Desired duty cycle for the blinking  0%(0x00) to 100%(0xFF) |
| ****frequency**** | Desired frequency for blinking  187.5KHz to 46 Hz |
| ****wdt\_val**** | Watchdog timer reload value  LED\_PWM\_WDT\_DIS  LED\_PWM\_WDT\_200MS  LED\_PWM\_WDT\_400MS  LED\_PWM\_WDT\_600MS  LED\_PWM\_WDT\_800MS  LED\_PWM\_WDT\_4SEC – default value  LED\_PWM\_WDT\_51SEC |

Outputs

None

### led\_breath

Function Header

**void led\_breath( uint8\_t led\_id, uint8\_t psize, uint16\_t led\_limit, uint32\_t led\_delay \**

**uint32\_t led\_int, uint32\_t led\_step )**

Description

Enables the hardware breathing of the specified LED instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****led\_id**** | 0-based LED ID |
| ****psize**** | Type of PWM  LED\_CFG\_PWM\_WIDTH\_8  LED\_CFG\_PWM\_WIDTH\_7  LED\_CFG\_PWM\_WIDTH\_6 |
| ****sym\_disable**** | Enable / disable the symmetry mode  LED\_CFG\_SYMMETRY\_DIS  LED\_CFG\_SYMMETRY\_EN |
| ****led\_limit**** | Minimum and maximum limits for breathing |
| ****led\_delay**** | Delay values for the maximum and minimum limit |
| ****led\_int**** | Step size interval value |
| ****led\_step**** | Step size update value |

Outputs

None

## LED peripheral functions

The list of LED peripheral functions are –

* p\_led\_configuration\_reg\_set
* p\_led\_configuration\_reg\_get
* p\_led\_control\_set
* p\_led\_clk\_src\_set
* p\_led\_sync\_set
* p\_led\_pwm\_size\_set
* p\_led\_update\_enable\_set
* p\_led\_wdt\_reset
* p\_led\_wdt\_reload
* p\_led\_symmetry\_set
* p\_led\_limits\_set
* p\_led\_limits\_get
* p\_led\_delay\_set
* p\_led\_delay\_get
* p\_led\_duty\_cycle\_set
* p\_led\_prescalar\_set
* p\_led\_stepsize\_set
* p\_led\_updateInterval\_set
* p\_led\_output\_delay\_set

### p\_led\_configuration\_reg\_set

Function Header

**void p\_led\_configuration\_reg\_set( uint8\_t led\_id, uint32\_t new\_val )**

Description

Writes to the configuration register of the specified led instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****led\_id**** | 0-based LED ID |
| ****new\_val**** | 32-bit value for the new contents |

Outputs

None

### p\_led\_configuration\_reg\_get

Function Header

**uint32\_t p\_led\_configuration\_reg\_get( uint8\_t led\_id )**

Description

Reads the configuration register of the specified led instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****led\_id**** | 0-based LED ID |

Outputs

Configuration register contents, 0xBAAAAAAD – read fail

### p\_led\_control\_set

Function Header

**void p\_led\_control\_set( uint8\_t led\_id, uint8\_t new\_val )**

Description

Reads the configuration register of the specified led instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****led\_id**** | 0-based LED ID |
| ****new\_val**** | New configuration value  LED\_CFG\_CNTL\_LO  LED\_CFG\_CNTL\_BREATH  LED\_CFG\_CNTL\_BLINK  LED\_CFG\_CNTL\_HI |

Outputs

None

### p\_led\_clk\_src\_set

Function Header

**void p\_led\_clk\_src\_set( uint8\_t led\_id, uint8\_t new\_val )**

Description

Configure the base clock in the configuration register of the specified led instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****led\_id**** | 0-based LED ID |
| ****new\_val**** | Clock source selection  LED\_CFG\_CLK\_SRC\_MCLK – 48MHz  LED\_CFG\_CLK\_SRC\_32K – 32.768KHz |

Outputs

None

### p\_led\_sync\_set

Function Header

**void p\_led\_sync\_set( uint8\_t led\_id, uint8\_t new\_val )**

Description

Configure the synchronization of the breathing/blinking hardware between all the LED instances.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****led\_id**** | 0-based LED ID |
| ****new\_val**** | Enable / disable synchronization  LED\_CFG\_SYNC\_SET  LED\_CFG\_SYNC\_CLR |

Outputs

None

### p\_led\_pwm\_size\_set

Function Header

**void p\_led\_pwm\_size\_set( uint8\_t led\_id, uint8\_t new\_val )**

Description

Configures the pwm mode for the specified led instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****led\_id**** | 0-based LED ID |
| ****new\_val**** | Type of PWM  LED\_CFG\_PWM\_WIDTH\_8  LED\_CFG\_PWM\_WIDTH\_7  LED\_CFG\_PWM\_WIDTH\_6 |

Outputs

None

### p\_led\_update\_enable\_set

Function Header

**void p\_led\_update\_enable\_set( uint8\_t led\_id, uint8\_t new\_val )**

Description

Enables the hardware to update the values of the led\_delay, led\_step and led\_int registers.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****led\_id**** | 0-based LED ID |
| ****new\_val**** | Update mode  LED\_CFG\_EN\_UPDATE |

Outputs

None

### p\_led\_reset

Function Header

**void p\_led\_reset( uint8\_t led\_id )**

Description

Resets the hardware block of the specified hardware instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****led\_id**** | 0-based LED ID |

Outputs

None

### p\_led\_wdt\_reload

Function Header

**void p\_led\_wdt\_reload( uint8\_t led\_id, uint8\_t new\_val )**

Description

Loads the led wdt reload value in the configuration register of the specified led instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****led\_id**** | 0-based LED ID |
| ****new\_val**** | 8-bit watchdog reload value  LED\_PWM\_WDT\_DIS  LED\_PWM\_WDT\_200MS  LED\_PWM\_WDT\_400MS  LED\_PWM\_WDT\_600MS  LED\_PWM\_WDT\_800MS  LED\_PWM\_WDT\_4SEC – default value  LED\_PWM\_WDT\_51SEC |

Outputs

None

### p\_led\_symmetry\_set

Function Header

**void p\_led\_symmetry\_set( uint8\_t led\_id, uint32\_t new\_val )**

Description

Configures the symmetry configuration for the breathing mode of the specified led instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****led\_id**** | 0-based LED ID |
| ****new\_val**** | Symmetry mode  LED\_CFG\_SYMMETRY\_DIS  LED\_CFG\_SYMMETRY\_EN |

Outputs

None

### p\_led\_limits\_set

Function Header

**void p\_led\_limits\_set( uint8\_t led\_id, enum LED\_LIMIT limit\_type, uint16\_t new\_val )**

Description

Writes the maximum and minimum duty cycle values to the led limits register of the specified led instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****led\_id**** | 0-based LED ID |
| ****limit\_type**** | Limit type that needs to be modified  LED\_MIN\_LIMIT – minimum limit  LED\_MAX\_LIMIT – maximum limit  LED\_COMPLETE\_LIMIT – both |
| ****new\_val**** | Value for the limit setting |

Outputs

None

### p\_led\_limits\_get

Function Header

**uint16\_t p\_led\_limits\_get( uint8\_t led\_id )**

Description

Reads the maximum and minimum duty cycle values from the led limits register of the specified led instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****led\_id**** | 0-based LED ID |

Outputs

Current limit value, 0xBAAD – read failed

### p\_led\_delay\_set

Function Header

**void p\_led\_delay\_set( uint8\_t led\_id, enum LED\_DELAY delay\_type, uint32\_t new\_val )**

Description

Writes the maximum and minimum delay values to the led delay register of the specified led instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****led\_id**** | 0-based LED ID |
| ****limit\_type**** | Delay type that needs to be modified  LED\_LOW\_DELAY  LED\_HIGH\_DELAY  LED\_COMPLETE\_DELAY |
| ****new\_val**** | Value for the delay setting |

Outputs

None

### p\_led\_delay\_get

Function Header

**uint32\_t p\_led\_delay\_get( uint8\_t led\_id )**

Description

Reads the maximum and minimum delay values from the led delay register of the specified led instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****led\_id**** | 0-based LED ID |

Outputs

Current delay value, 0xBAAAAAAD – read failed

### p\_led\_duty\_cycle\_set

Function Header

**void p\_led\_duty\_cycle\_set( uint8\_t led\_id, uint8\_t new\_val )**

Description

Sets the duty cycle for the specified led instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****led\_id**** | 0-based LED ID |
| ****new\_val**** | New value (0x00 – 0% to 0xFF – 100%) |

Outputs

None

### p\_led\_prescalar\_set

Function Header

**void p\_led\_prescalar\_set ( uint8\_t led\_id, uint16\_t new\_val )**

Description

Sets the prescalar value for the specified led instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****led\_id**** | 0-based LED ID |
| ****new\_val**** | New value  Frequency = clock / (255 \* (prescalar + 1)) |

Outputs

None

### p\_led\_stepsize\_set

Function Header

**void p\_led\_stepsize\_set( uint8\_t led\_id, enum LED\_STEP step\_number, uint32\_t new\_val )**

Description

Sets the step size update values for segment 0-7of the specified led instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****led\_id**** | 0-based LED ID |
| ****step\_number**** | Step number that needs to be modified  LED\_STEP\_0  LED\_STEP\_1  LED\_STEP\_2  LED\_STEP\_3  LED\_STEP\_4  LED\_STEP\_5  LED\_STEP\_6  LED\_STEP\_7  LED\_STEP\_ALL |
| ****new\_val**** | Value for the step size setting |

Outputs

None

### p\_led\_updateInterval\_set

Function Header

**void p\_led\_updateInterval\_set( uint8\_t led\_id, enum LED\_INT interval\_number, uint32\_t new\_val)**

Description

Sets the interval period between the two successive updates for the current duty cycle of the specified led instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****led\_id**** | 0-based LED ID |
| ****step\_number**** | Interval number that needs to be modified  LED\_INT\_0  LED\_INT\_1  LED\_INT\_2  LED\_INT\_3  LED\_INT\_4  LED\_INT\_5  LED\_INT\_6  LED\_INT\_7  LED\_INT\_ALL |
| ****new\_val**** | Value for the interval setting |

Outputs

None

### p\_led\_output\_delay\_set

Function Header

**void p\_led\_output\_delay\_set( uint8\_t led\_id, uint8\_t new\_val )**

Description

Writes to the output delay register of the specified LED instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****led\_id**** | 0-based LED ID |
| ****new\_val**** | New value for the delay setting |

Outputs

None

# SPI

## SPI API’s

These SPI APIs are in ROM code, which can perform external SPI read only.

### Power SPI Controller On and Off

#### Declaration

void SPI\_Power\_OnOff(

uint8\_t spi\_id,

uint8\_t power\_on

);

#### Input Parameters

spi\_id A parameter to identify which SPI controller is to be used.

SPI\_BUS0 0 SPI controller 0 (shared SPI)

SPI\_BUS1 1 SPI controller 1 (private SPI)

power\_on A parameter to identify which SPI controller is to be used.

SPI\_OFF 0 Turn off SPI controller

SPI\_ON 1 Turn on SPI controller

#### Output

None

#### Description

This function can enable or disable either of the two SPI controllers on the CEC1302.

### Configure SPI Controller

#### Declaration

uint8\_t SPI\_Configure(

uint8\_t spi\_id,

uint32\_t clock\_speed

);

#### Input Parameters

spi\_id A parameter to identify which SPI controller is to be used.

SPI\_BUS0 0 SPI controller 0 (shared SPI)

SPI\_BUS1 1 SPI controller 1 (private SPI)

clock\_speed A parameter to configure the speed of the SPI interface

0 = 48 MHz

1 = 24 MHz

* 1. = 12 MHz
  2. = 8 MHz

Other values reserved

#### Output

Return code:

SPI\_ BAD\_CTRL 0 Command failed; bad SPI id

SPI\_ OK 1 Normal return

#### Description

This function configures the selected SPI controller for use by the ROM API, as well as configuring a specific speed.

### Set SPI Chip Select

#### Declaration

void SPI\_CS\_Select(

uint8\_t spi\_bus\_id,

uint8\_t spi\_cs\_id,

uint8\_t cs\_state

);

#### Input Parameters

spi\_bus\_id A parameter to identify which SPI controller is to be used.

SPI\_BUS0 0 SPI controller 0 (shared SPI)

SPI\_BUS1 1 SPI controller 1 (private SPI)

spi\_cs\_id Determines the chip select for the controller selected by spi\_cs\_id.

CS0 0 The GPIO for CS0

CS1 1 The GPIO for CS1

cs\_state The pin determined by spi\_bus\_id and spi\_cs\_id will be set to this value (0 or 1)

#### Output

None

#### Description

The GPIO pin associated with the chip select determined by spi\_bus\_id and spi\_cs\_id is set high or low according cs\_state. Only the least significant bit of all three parameters is examined.

### Transmit SPI Read Command

#### Declaration

uint8\_t SPI\_Transmit\_Read(

uint8\_t spi\_id,

uint32\_t cmd\_id,

uint32\_t spi\_addr

);

#### Input Parameters

spi\_id A parameter to identify which SPI controller is to be used.

SPI\_BUS0 0 SPI controller 0 (shared SPI)

SPI\_BUS1 1 SPI controller 1 (private SPI)

cmd\_id An index that selects an 8-bit SPI read command. Valid values are:

* + - 1. Normal read (0x03)
      2. Fast read (0x0B)
      3. Fast read with double data rate return (0x3B)
      4. Normal read with a 32-bit address (0x13)
      5. Fast read with a 32-bit address (0x0C)
      6. Fast read with a 32-bit address and DDR (0x3C)

spi\_addr The address to be sent to the SPI device. If the command type calls for a 24-bit address, the address occupies the least-significant 24-bits of this parameter.

#### Output

Return code:

SPI\_OK 0 Normal return

SPI\_BAD\_CTRL 1 Command failed; bad control data or bad SPI id

SPI\_BAD\_CS 2 Command failed; bad chip select

SPI\_BAD\_CMD 3 Command failed; bad command id

SPI\_BAD\_ADDR 4 Pointer points to a buffer that is outside the SRAM

SPI\_TX\_TIMEOUT 5 Transmit timeout failure

#### Description

This command starts a SPI read transaction on the SPI controller specified by the spi\_id parameter.

Caller must have asserted chip select to the target SPI device using SPI\_CS\_Select() API mentioned above.

### Read Data from SPI, Polled

#### Declaration

uint8\_t SPI\_Data\_Read\_Polled(

uint8\_t spi\_id,

uint32\_t num\_bytes,

uint8\_t \*pbuff

);

#### Input Parameters

spi\_id A parameter to identify which SPI controller is to be used.

SPI\_BUS0 0 SPI controller 0 (shared SPI)

SPI\_BUS1 1 SPI controller 1 (private SPI)

num\_bytes The number of bytes to transfer from an external SPI flash device to internal SRAM

\*pbuff A pointer to a region of SRAM that this function will write data read from the SPI flash device

#### Output

Return code:

SPI\_OK 0 Normal return

SPI\_BAD\_CTRL 1 Command failed; bad control data or bad SPI id

SPI\_TX\_TIMEOUT 5 Transmit timeout failure

SPI\_RX\_TIMEOUT 6 Read timeout failure

#### Description

This function will transfer num\_bytes of data between an external SPI Flash device and the internal SRAM memory. The function only performs the data transfers from the SPI device to the area in memory pointed to by pbuff and must be preceded by SPI\_Transmit\_Read(). Transfers are done byte at a time from the SPI controller and memory, under firmware control.

This function is blocking. Once invoked, it will not return until all num\_bytes have been transferred.

### Transfer Data from SPI, Polled

#### Declaration

uint32\_t SPI\_Data\_Transfer\_Polled(

uint8\_t spi\_id,

uint32\_t cmd\_id,

uint32\_t spi\_addr,

uint8\_t \*pbuff,

uint32\_t num\_bytes

);

#### Input Parameters

spi\_id A parameter to identify which SPI controller is to be used.

SPI\_BUS0 0 SPI controller 0 (shared SPI)

SPI\_BUS1 1 SPI controller 1 (private SPI)

cmd\_id The 8-bit command to be sent to the SPI device.

Spi\_addr The 24-bit address to be sent to the SPI device. The address occupies the least-significant 24-bits of this parameter.

\*pbuff A pointer to a region of SRAM that this function will write data read from the SPI flash device

num\_bytes The number of bytes to transfer from an external SPI flash device to internal SRAM

#### Output

Return code:

SPI\_OK 0 Normal return

SPI\_BAD\_CTRL 1 Command failed; bad control data or bad SPI id

SPI\_BAD\_CS 2 Command failed; bad chip select

SPI\_BAD\_CMD 3 Command failed; bad command id

SPI\_BAD\_ADDR 4 Pointer points to a buffer that is outside the SRAM

SPI\_TX\_TIMEOUT 5 Transmit timeout failure

SPI\_RX\_TIMEOUT 6 Read timeout failure

#### Description

This function combines SPI\_Transmit\_Read() with SPI\_Data\_Read\_Polled().

This function is blocking. Once invoked, it will not return until all num\_bytes have been transferred.

### Transfer Data from SPI, DMA

#### Declaration

uint32\_t SPI\_Data\_Transfer\_DMA(

uint8\_t spi\_id,

uint32\_t cmd\_id,

uint32\_t spi\_addr,

uint32\_t num\_bytes,

uint8\_t \*pbuff

);

#### Input Parameters

spi\_id A parameter to identify which SPI controller is to be used.

SPI\_BUS0 0 SPI controller 0 (shared SPI)

SPI\_BUS1 1 SPI controller 1 (private SPI)

cmd\_id The 8-bit command to be sent to the SPI device.

Spi\_addr The 24-bit address to be sent to the SPI device. The address occupies the least-significant 24-bits of this parameter.

Num\_bytes The number of bytes to transfer from an external SPI flash device to internal SRAM

\*pbuff A pointer to a region of SRAM that this function will write data read from the SPI flash device

#### Output

Return code:

SPI\_OK 0 Normal return

SPI\_BAD\_CTRL 1 Command failed; bad control data or bad SPI id

SPI\_BAD\_CS 2 Command failed; bad chip select

SPI\_BAD\_CMD 3 Command failed; bad command id

SPI\_BAD\_ADDR 4 Pointer points to a buffer that is outside the SRAM

SPI\_TX\_TIMEOUT 5 Transmit timeout failure

SPI\_RX\_TIMEOUT 6 Read timeout failure

SPI\_BAD\_DMA 7 Bad DMA; controller not enabled

#### Description

This function performs the SPI\_Transmit\_Read() and then sets up a DMA channel to transfer num\_bytes from an external SPI Flash device to a buffer in internal SRAM pointed to by \*pbuff.

The function is non-blocking. Invoking it starts the DMA controller, which will continue to transfer data from an external SPI device to the internal SRAM autonomously. Once started, engine will reject further attempts to start a decryption or verify operation until after it has completed. Firmware can check the status of the DMA using the SPI\_DMA\_Busy() function.

**Note:** This function uses DMA Read Channel 10 when spi\_id is 0 (for the Shared SPI) and DMA Read Channel 11 when spi\_id is 1 (for the Private SPI). Other firmware must not use the DMA channel used by SPI\_Data\_Transfer\_DMA() when this function is in operation.

### SPI DMA Done

#### Declaration

uint8\_t SPI\_DMA\_Busy(

uint8\_t spi\_id

);

#### Input Parameters

spi\_id A parameter to identify which SPI controller is to be used.

SPI\_BUS0 0 SPI controller 0 (shared SPI)

SPI\_BUS1 1 SPI controller 1 (private SPI)

#### Output

Return code:

SPI\_DMA\_BUSY 0 SPI DMA controller is busy

SPI\_DMA\_DONE 1 SPI DMA controller has completed the transfer

#### Description

This function reports the state of the DMA transaction started by SPI\_Data\_Read\_DMA(). When this function reports that the DMA is done, all bytes requested by the read request have been transferred from the SPI device to SRAM.

### Abort SPI Transaction

#### Declaration

void SPI\_Abort(

uint8\_t spi\_id

);

#### Input Parameters

spi\_id A parameter to identify which SPI controller is to be used.

SPI\_BUS0 0 SPI controller 0 (shared SPI)

SPI\_BUS1 1 SPI controller 1 (private SPI)

#### Output

None

#### Description

This function will terminate a DMA transaction started by SPI\_Data\_Read\_DMA(). There is no indication how much of the DMA transfer completed.

## Applications

In application level, user can use ROM SPI APIs mentioned above to perform SPI read, following describes a few example how to use these APIs to perform data read from external SPI chip.

### MACROs Definition

//

// Logical SPI ID for API calls

//

#define TRUE (1)

#define FALSE (0)

#define SPI\_BUS0\_ID (0)

#define SPI\_BUS1\_ID (1)

#define SPI\_BUS\_MAX (2)

#define SPI\_CS0 (0)

#define SPI\_CS1 (1)

#define SPI\_CS\_MAX (2)

#define SPI\_CS\_ASSERT (0UL)

#define SPI\_CS\_DEASSERT (1UL)

#define SPI\_CMD\_READ (0x03u)

#define SPI\_CMD\_READ32 (0x13u)

#define SPI\_CMD\_READ\_FAST (0x0Bu)

#define SPI\_CMD\_READ32\_FAST (0x0Cu)

#define SPI\_CMD\_READ\_FAST\_DDR (0x3Bu)

#define SPI\_CMD\_READ32\_FAST\_DDR (0x3Cu)

#define SPI\_CMD\_READ\_JEDEC\_ID (0x9Fu)

#define SPI\_CMD\_READ\_SFDP (0x5Au)

#define SPI\_READ (0UL)

#define SPI\_READ\_FAST (1UL)

#define SPI\_READ\_FAST\_DDR (2UL)

#define SPI\_READ32 (3UL)

#define SPI\_READ32\_FAST (4UL)

#define SPI\_READ32\_FAST\_DDR (5UL)

#define SPI\_READ\_JEDEC (6UL)

#define SPI\_READ\_SFDP (7UL)

#define SPI\_READ\_MAX (8UL)

// spi\_init spi\_config bits

//

// SPI clock b[6:0]

#define SPI\_CFG\_FREQ\_48M (0ul)

#define SPI\_CFG\_FREQ\_24M (1ul)

#define SPI\_CFG\_FREQ\_12M (2ul)

#define SPI\_CFG\_FREQ\_8M (3ul)

### Example 1 – general purpose read w/o DMA

Followings are the required APIs and calling sequence as application performs Private SPI chip JEDEC read,

/\* buffer to store data read from external SPI chip \*/

Uint8\_t SPI\_read\_buff[4];

/\* init PVT-SPI1 signals – SPI function; \*/

// need to initialize GPIOs as PVT SPI functionality and its CS#

/\* enable PVT-SPI1 block \*/

SPI\_Power\_OnOff(SPI\_BUS1\_ID, TRUE);

/\* configure SPI speed \*/

SPI\_Configure(SPI\_BUS1\_ID, SPI\_CFG\_FREQ\_24M);

/\* assert CS \*/

SPI\_CS\_Select(SPI\_BUS1\_ID, SPI\_CS0, GP\_SPI\_CS\_ASSERT);

/\* read SPI data – tx SPI command \*/

SPI\_Transmit\_Read(SPI\_BUS1\_ID, SPI\_READ\_JEDEC, 0ul);

/\* read SPI data – read data \*/

SPI\_Data\_Read\_Polled(SPI\_BUS1\_ID, 3, SPI\_read\_buff);

/\* de-assert CS \*/

API02\_SPI\_CS\_Select(SPI\_BUS1\_ID, SPI\_CS0, GP\_SPI\_CS\_DEASSERT);

Then 3-byte SPI chip’s JEDEC data is stored in SPI\_read\_buffer[0] ~ SPI\_read\_buffer[2].

### Example 2 – read SPI any location w/o DMA

Followings are the required APIs and calling sequence as application performs Private SPI chip any location read,

/\* buffer to store data read from external SPI chip \*/

Uint8\_t SPI\_read\_buff[64];

/\* init PVT-SPI1 signals – SPI function; \*/

// need to initialize GPIOs as PVT SPI functionality and its CS#

/\* enable PVT-SPI1 block \*/

SPI\_Power\_OnOff(SPI\_BUS1\_ID, TRUE);

/\* configure SPI speed \*/

SPI\_Configure(SPI\_BUS1\_ID, SPI\_CFG\_FREQ\_24M);

/\* assert CS \*/

SPI\_CS\_Select(SPI\_BUS1\_ID, SPI\_CS0, GP\_SPI\_CS\_ASSERT);

/\* read SPI data – tx SPI command and read data \*/

SPI\_Data\_Transfer\_Polled(SPI\_BUS1\_ID, SPI\_READ\_FAST\_DDR, 0xFF00, SPI\_read\_buff, 64);

/\* de-assert CS \*/

API02\_SPI\_CS\_Select(SPI\_BUS1\_ID, SPI\_CS0, GP\_SPI\_CS\_DEASSERT);

Then 64-byte data read from SPI chip’s 0xFF00 to 0xFF3F location is stored in SPI\_read\_buffer[0] ~ SPI\_read\_buffer[63].

### Example 3 – read SPI any location w/ DMA

Followings are the required APIs and calling sequence as application performs Private SPI chip any location read with DMA,

/\* buffer to store data read from external SPI chip \*/

Uint8\_t SPI\_read\_buff[64];

/\* init PVT-SPI1 signals – SPI function; \*/

// need to initialize GPIOs as PVT SPI functionality and its CS#

/\* power on DMA module for SPI/DMA read \*/

// need to activate DMA block

/\* enable PVT-SPI1 block \*/

SPI\_Power\_OnOff(SPI\_BUS1\_ID, TRUE);

/\* configure SPI speed \*/

SPI\_Configure(SPI\_BUS1\_ID, SPI\_CFG\_FREQ\_24M);

/\* assert CS \*/

SPI\_CS\_Select(SPI\_BUS1\_ID, SPI\_CS0, GP\_SPI\_CS\_ASSERT);

/\* read data via SPI / DMA: DMA interrupt is disabled, apply polling \*/

SPI\_Data\_Transfer\_DMA(SPI\_BUS1\_ID, SPI\_READ\_FAST\_DDR, 0xFF00, 64, SPI\_read\_buff);

/\* wait SPI/DMA read is done \*/

while( !SPI\_DMA\_Done(SPI\_BUS1\_ID))

{

\_\_NOP();

}

/\* de-assert CS \*/

API02\_SPI\_CS\_Select(SPI\_BUS1\_ID, SPI\_CS0, GP\_SPI\_CS\_DEASSERT);

Then 64-byte data read from SPI chip’s 0xFF00 to 0xFF3F location is stored in SPI\_read\_buffer[0] ~ SPI\_read\_buffer[63].

# WDT

The function of the Watchdog Timer is to provide a mechanism to detect if the internal embedded controller has failed. When enabled, the Watchdog Timer (WDT) circuit will generate a WDT Event and reset the embedded controller and it’s subsystem, if the user program fails to reload the WDT within a specified length of time known as the WDT Interval.

**WDT APIs**

wdt\_start

wdt\_stop

wdt\_kick

wdt\_sleep

wdt\_clk\_reqd\_sts\_getwdt\_reset\_on\_sleep

**WDT instance**

WDT control register

WDT kick register

WDT Load register

WDT Count register

**WDT Peripheral Functions**

p\_wdt\_enable\_set

p\_wdt\_enable\_clr

p\_wdt\_status\_get

p\_wdt\_status\_clr

p\_wdt\_kick

p\_wdt\_load\_write

p\_wdt\_load\_read

p\_wdt\_count\_read

PCR Peripheral Functions

## WDT API’s

The list of WDT APIs

* wdt\_start
* wdt\_stop
* wdt\_kick
* wdt\_sleep
* wdt\_clk\_reqd\_sts\_get
* wdt\_reset\_on\_sleep

### wdt\_start

Function Header

**void wdt\_start(uint16\_t delay\_value)**

Description

This API will load the WDT load register, reset the WDT status and start WDT.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| delay\_value | Delay value before the WDT fires. The value must be in milliseconds (1 – 65536 ms). |

Outputs

None

### wdt\_stop

Function Header

**void wdt\_stop(void)**

Description

This stops the WDT.

Inputs

None

Outputs

None

### wdt\_kick

Function Header

**void wdt\_kick(void)**

Description

This sets the WDT kick, thus reloading the WDT reload value. Thus, preventing the WDT from resetting the device.

Inputs

None

Outputs

None

### wdt\_sleep

Function Header

**void wdt\_sleep(uint8\_t sleep\_en)**

Description

Enable/Disable clock gating on WDT

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| sleep\_en | 1 = Sleep Enable  0 = Sleep Disable |

Outputs

None

### wdt\_clk\_reqd\_sts\_get

Function Header

**uint8\_t wdt\_clk\_reqd\_sts\_get (void)**

Description

Returns clk required status

Inputs

None

Outputs

0(CLK not required), Non-zero (CLK required)

### wdt\_reset\_on\_sleep

Function Header

**void wdt\_reset\_on\_sleep (uint8\_t reset\_en)**

Description

Enable/Disable WDT block reset on sleep

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| reset\_en | 1 = Enable Reset on Sleep  0 = Disable Reset on Sleep |

Outputs

None

## WDT Peripheral Functions

The list of PWM peripheral functions are listed below:

* p\_wdt\_enable\_set
* p\_wdt\_enable\_clr
* p\_wdt\_status\_get
* p\_wdt\_status\_clr
* p\_wdt\_kick
* p\_wdt\_load\_write
* p\_wdt\_load\_read
* p\_wdt\_count\_read

### p\_wdt\_enable\_set

Function Header

**void p\_wdt\_enable\_set**  **(void )**

Description

Enables and starts watchdog timer.

Inputs

None

Outputs

None

### p\_wdt\_enable\_clr

Function Header

**void p\_wdt\_enable\_clr**  **(void)**

Description

Disables and stop watchdog timer.

Inputs

None

Outputs

None

### p\_wdt\_status\_get

Function Header

**uint8\_t p\_wdt\_status\_get (void )**

Description

Get the WDT status.

Inputs

None

Outputs

Returns status. If the last reset of MEC device was caused by an underflow of the WDT, then this status is “1” else its “0”.

### p\_wdt\_status\_clr

Function Header

**void p\_wdt\_status\_clr (void)**

Description

Clears the WDT status.

Inputs

None

Outputs

None

### p\_wdt\_kick

Function Header

**void p\_wdt\_kick (void)**

Description

When the WDT Enable is set, this function causes the WDT to reload the WDT Load Register (WDT\_load/delay) value, thus preventing the WDT from resetting the device. When the WDT Enable is cleared to ‘0’, this function has no effect.

Inputs

None

Outputs

None

### p\_wdt\_load\_write

Function Header

**void p\_wdt\_load\_write (uint16\_t count)**

Description

Reloads WDT with the count value.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| count | Count value that will be reloaded to WDT |

Outputs

None

### p\_wdt\_load\_read

Function Header

**uint16\_t p\_wdt\_load\_read (void)**

Description

Reads the WDT load value.

Inputs

None

Outputs

Count value that will be reloaded to WDT

### p\_wdt\_count\_read

Function Header

**uint16\_t p\_wdt\_count\_read (void)**

Description

Reads the current WDT count.

Inputs

None

Outputs

Current WDT count value

# Interrupt

## Interrupt APIs

The list of Interrupt APIs

* interrupt\_init
* interrupt\_mode\_set
* interrupt\_reset
* interrupt\_device\_enable
* interrupt\_device\_disable
* interrupt\_device\_ecia\_source\_clear
* interrupt\_device\_ecia\_source\_get
* interrupt\_device\_ecia\_result\_get
* interrupt\_device\_nvic\_enable
* interrupt\_device\_nvic\_priority\_set
* interrupt\_device\_nvic\_priority\_get
* interrupt\_device\_nvic\_pending\_get
* interrupt\_device\_nvic\_pending\_set
* interrupt\_device\_nvic\_pending\_clear

Usage:

1. Interrupt vector table and ISRs will part of application
2. Initialization:

Application can initialize using *interrupt\_init* function, by specifying NVIC direct mode or fully aggregated mode. If in Direct Mode, certain interrupts could be still kept aggregated; which needs to be specified using girq\_bitmask in interrupt\_init function

Example for configuring direct mode:

girq\_bitmask\_aggregated = (MEC\_GIRQ12\_BITMASK |

MEC\_GIRQ13\_BITMASK | MEC\_GIRQ15\_BITMASK);

interrupt\_init(INTERRUPT\_MODE\_DIRECT, girq\_bitmask\_aggregated);

1. Individual blocks can enable their interrupts using interrupt\_device\_enable function

Example:

interrupt\_device\_enable (BTMR0\_IROUTE);

interrupt\_device\_enable (BTMR1\_IROUTE);

1. Application can use other functions, based on need basis.

Example to clear source in ECIA:

interrupt\_device\_ecia\_source\_clear(BTMR0\_IROUTE);

### interrupt\_init

Function Header

**void interrupt\_init(uint8\_t mode, uint32\_t girq\_bitmask)**

Description

Initialize and configure Interrupt mode

Note1: All GPIO’s and wake capable sources are always aggregated! GPIO’s interrupts will still work in direct mode. Block wakes are not be routed to the processor in direct mode.

Note2: This function disables and enables global interrupt

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| mode | 1 – Direct Mode, 0 – Fully aggregated mode |
| girq\_bitmask | Bitmask of GIRQ to be configured as aggregated. This parameter is only applicable in direct mode. |

Outputs

None

Example Usage

Examples for configuring interrupt mode. The effect of the interrupt configuration on Basic Timer 4 (timer4) interrupt is provided as an example.

A. If one is using fully aggregated mode, then we would need to call as follows:

interrupt\_init(0, 0);

In this case direct mode won’t work. And timer4 interrupt will be aggregated (GIRQ23 bit5)

B. If one is using direct mode; then one would to need to call as follows:

interrupt\_init(1, 0);

In this case timer4 interrupt would be direct.

C. If one is using direct mode; but wants timer4 to be aggregated to GIRQ23; then we would need to

specify GIRQ23 aggregated as follows:

interrupt\_init(1, MEC\_GIRQ23\_BITMASK);

In this case timer4 interrupt would be aggregated.

Note that *interrupt\_mode\_set* function is called internally *interrupt\_init* function; so application might not have a need to use *interrupt\_mode\_set* API.

### interrupt\_mode\_set

Function Header

**void interrupt\_mode\_set(uint8\_t mode)**

Description

Set interrupt routing mode to aggregated or direct.

*NVIC, ECIA Routing Policy*

In Direct Mode, some interrupts could be configured to be used as aggregated.

Configuration used for direct mode:

1. Set ECS Interrupt Direct enable bit.

2. If GIRQn aggregated set Block Enable bit.

3. If GIRQn direct then clear Block Enable bit and enable individual NVIC inputs.

Configuration used for aggregated mode:

1. Set ECS Interrupt Direct enable bit.

2. Set all Block Enable bits

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| mode | 1 – Direct Mode, 0 – Fully aggregated mode |

Outputs

None

### interrupt\_reset

Function Header

**void interrupt\_reset(void)**

Description

Clears all individual interrupts Enables and Source in ECIA, and clears all NVIC external enables and pending bits.

Inputs

None

Outputs

None

### interrupt\_device\_enable

Function Header

**void interrupt\_device\_enable(uint32\_t dev\_iroute)**

Description

Enables interrupt for a device

Note: This function disables and enables global interrupt

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| dev\_iroute | device IROUTING information |

Outputs

None

### interrupt\_device\_disable

Function Header

**void interrupt\_device\_disable(uint32\_t dev\_iroute)**

Description

Disables interrupt for a device

Note: This function disables and enables global interrupt

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| dev\_iroute | device IROUTING information |

Outputs

None

### interrupt\_device\_ecia\_source\_clear

Function Header

**void interrupt\_device\_ecia\_source\_clear(uint32\_t dev\_iroute)**

Description

Clear Source in the ECIA for the device

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| dev\_iroute | device IROUTING information |

Outputs

None

### interrupt\_device\_ecia\_source\_get

Function Header

**uint32\_t interrupt\_device\_ecia\_source\_get(uint32\_t dev\_iroute)**

Description

Get the Source bit in the ECIA for the device

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| dev\_iroute | device IROUTING information |

Outputs

0 if source bit not set; else non-zero value

### interrupt\_device\_ecia\_result\_get

Function Header

**uint32\_t interrupt\_device\_ecia\_result\_get(uint32\_t dev\_iroute)**

Description

Get the Result bit in the ECIA for the device

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| dev\_iroute | device IROUTING information |

Outputs

0 if result bit not set; else non-zero value

### interrupt\_device\_nvic\_enable

Function Header

**void interrupt\_device\_nvic\_enable(uint32\_t dev\_iroute)**

Description

Enable/Disable the NVIC (in the NVIC controller) for the device

Note 1: Recommended to use interrupt\_device\_enable, interrupt\_device\_disable to enable/disable interrupts for the device, since those APIs configure ECIA as well

Note 2: This function disables and enables global interrupt

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| dev\_iroute | device IROUTING information |
| en\_flag | 1 = Enable the NVIC IRQ, 0 = Disable the NVIC IRQ |

Outputs

None

### interrupt\_device\_nvic\_priority\_set

Function Header

**void interrupt\_device\_nvic\_priority\_set(uint32\_t dev\_iroute)**

Description

Set NVIC priority for specified peripheral interrupt

Note: If ECIA is in aggregated mode, the priority affects all interrupt sources in the GIRQ.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| dev\_iroute | device IROUTING information |
| nvic\_pri | NVIC Priority |

Outputs

None

### interrupt\_device\_nvic\_priority\_get

Function Header

**uint8\_t interrupt\_device\_nvic\_priority\_get(uint32\_t dev\_iroute)**

Description

Return NVIC priority for the device’s interrupt

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| dev\_iroute | device IROUTING information |

Outputs

NVIC Priority

### interrupt\_device\_nvic\_pending\_set

Function Header

**void interrupt\_device\_nvic\_pending\_set(uint32\_t dev\_iroute)**

Description

Set NVIC pending for interrupt source

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| dev\_iroute | device IROUTING information |

Outputs

None

### interrupt\_device\_nvic\_pending\_get

Function Header

**uint8\_t interrupt\_device\_nvic\_pending\_get (uint32\_t dev\_iroute)**

Description

Return NVIC pending for the device

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| dev\_iroute | device IROUTING information |

Outputs

0 (not pending), 1 (pending in NVIC)

### interrupt\_device\_nvic\_pending\_clear

Function Header

**void interrupt\_device\_nvic\_pending\_clear(uint32\_t dev\_iroute)**

Description

Clears NVIC pending for interrupt source

Note: This function disables and enables global interrupt

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| dev\_iroute | device IROUTING information |

Outputs

0 (not pending), 1 (pending in NVIC) — before clear

## Interrupt ECIA Peripheral Functions

The list of Interrupt ECIA (EC Interrupt Aggregator) Peripheral Functions

* p\_interrupt\_ecia\_block\_enable\_set
* p\_interrupt\_ecia\_block\_enable\_bitmask\_set
* p\_interrupt\_ecia\_block\_enable\_get
* p\_interrupt\_ecia\_block\_enable\_all\_set
* p\_interrupt\_ecia\_block\_enable\_clr
* p\_interrupt\_ecia\_block\_enable\_bitmask\_clr
* p\_interrupt\_ecia\_block\_enable\_all\_clr
* p\_interrupt\_ecia\_block\_irq\_status\_get
* p\_interrupt\_ecia\_block\_irq\_all\_status\_get
* p\_interrupt\_ecia\_girq\_source\_clr
* p\_interrupt\_ecia\_girq\_source\_get
* p\_interrupt\_ecia\_girq\_enable\_set
* p\_interrupt\_ecia\_girq\_enable\_clr
* p\_interrupt\_ecia\_girq\_enable\_get
* p\_interrupt\_ecia\_girq\_result\_get
* p\_interrupt\_ecia\_girqs\_source\_reset
* p\_interrupt\_ecia\_girqs\_enable\_reset
* p\_interrupt\_control\_set
* p\_interrupt\_control\_get

### p\_interrupt\_ecia\_block\_enable\_set

Function Header

**void p\_interrupt\_ecia\_block\_enable\_set(uint8\_t girq\_id)**

Description

Enable specified GIRQ in ECIA block

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| girq\_id | GIRQ Id |

Outputs

None

### p\_interrupt\_ecia\_block\_enable\_bitmask\_set

Function Header

void p\_interrupt\_ecia\_block\_enable\_bitmask\_set(uint32\_t girq\_bitmask)

Description

Enable GIRQs in ECIA block

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| girq\_bitmask | Bitmask of GIRQs to be enabled in ECIA Block |

Outputs

None

### p\_interrupt\_ecia\_block\_enable\_get

Function Header

uint8\_t p\_interrupt\_ecia\_block\_enable\_get(uint8\_t girq\_id)

Description

Check if specified GIRQ block enabled or not

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| girq\_id | GIRQ Id |

Outputs

1 if the particular GIRQ block enabled, else 0

### p\_interrupt\_ecia\_block\_enable\_all\_set

Function Header

void p\_interrupt\_ecia\_block\_enable\_all\_set(void)

Description

Set all GIRQ block enables

Inputs

None

Outputs

None

### p\_interrupt\_ecia\_block\_enable\_clr

Function Header

**void p\_interrupt\_ecia\_block\_enable\_clr(uint8\_t girq\_id)**

Description

Clear specified GIRQ in ECIA Block

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| girq\_id | GIRQ Id |

Outputs

None

### p\_interrupt\_ecia\_block\_enable\_bitmask\_clr

Function Header

void p\_interrupt\_ecia\_block\_enable\_bitmask\_clr(uint32\_t girq\_bitmask)

Description

Clear GIRQs in ECIA Block

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| girq\_bitmask | Bitmask of GIRQs to be cleared in ECIA Block |

Outputs

None

### p\_interrupt\_ecia\_block\_enable\_all\_clr

Function Header

void p\_interrupt\_ecia\_block\_enable\_all\_clr(void)

Description

Clears all GIRQ block enables

Inputs

None

Outputs

None

### p\_interrupt\_ecia\_block\_irq\_status\_get

Function Header

**uint32\_t p\_interrupt\_ecia\_block\_irq\_status\_get(uint8\_t girq\_id)**

Description

Get status of GIRQ in ECIA Block

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| girq\_id | GIRQ Id |

Outputs

0 if status bit not set; else non-zero value

### p\_interrupt\_ecia\_block\_irq\_all\_status\_get

Function Header

**uint32\_t p\_interrupt\_ecia\_block\_irq\_all\_status\_get(void)**

Description

Reads the Block IRQ Vector Register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| girq\_id | GIRQ Id |

Outputs

32-bit value

### p\_interrupt\_ecia\_girq\_source\_clr

Function Header

**void p\_interrupt\_ecia\_girq\_source\_clr(int16\_t girq\_id, uint8\_t bitnum)**

Description

Clear specified interrupt source bit in GIRQx

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| girq\_id | GIRQ Id |
| **bitnum** | Bit number —[0, 31] |

Outputs

None

### p\_interrupt\_ecia\_girq\_source\_get

Function Header

**uint32\_t p\_interrupt\_ecia\_girq\_source\_get(int16\_t girq\_id, uint8\_t bitnum)**

Description

Read the specified interrupt source bit in GIRQx

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| girq\_id | GIRQ Id |
| **bitnum** | Bit number —[0, 31] |

Outputs

0 if source bit not set; else non-zero value

### p\_interrupt\_ecia\_girq\_enable\_set

Function Header

**void p\_interrupt\_ecia\_girq\_enable\_set(uint16\_t girq\_id, uint8\_t bitnum)**

Description

Enable the specified interrupt in GIRQx

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| girq\_id | GIRQ Id |
| **bitnum** | Bit number —[0, 31] |

Outputs

None

### p\_interrupt\_ecia\_girq\_enable\_clr

Function Header

**void p\_interrupt\_ecia\_girq\_enable\_clr(uint16\_t girq\_id, uint8\_t bitnum)**

Description

Disable the specified interrupt in GIRQx

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| girq\_id | GIRQ Id |
| **bitnum** | Bit number —[0, 31] |

Outputs

None

### p\_interrupt\_ecia\_girq\_enable\_get

Function Header

**uint32\_t p\_interrupt\_ecia\_girq\_enable\_get (uint16\_t girq\_id, uint8\_t bitnum)**

Description

Read the status of the specified interrupt in GIRQx

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| girq\_id | GIRQ Id |
| **bitnum** | Bit number —[0, 31] |

Outputs

0 if enable bit not set; else non-zero value

### p\_interrupt\_ecia\_girq\_result\_get

Function Header

**uint32\_t p\_interrupt\_ecia\_girq\_result\_get (int16\_t girq\_id, uint8\_t bitnum)**

Description

Read the result bit of the interrupt in GIRQx

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| girq\_id | GIRQ Id |
| **bitnum** | Bit number —[0, 31] |

Outputs

0 if enable bit not set; else non-zero value

### p\_interrupt\_ecia\_girqs\_source\_reset

Function Header

void p\_interrupt\_ecia\_girqs\_source\_reset(void)

Description

Clear all aggregator GIRQn status registers

Inputs

None

Outputs

None

### p\_interrupt\_ecia\_girqs\_enable\_reset

Function Header

void p\_interrupt\_ecia\_girqs\_enable\_reset(void)

Description

Clear all aggregator GIRQn enables

Inputs

None

Outputs

None

### p\_interrupt\_control\_set

Function Header

void p\_interrupt\_control\_set(uint8\_t nvic\_en\_flag)

Description

Function to set interrupt control

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| nvic\_en\_flag | 0 = Alternate NVIC disabled, 1 = Alternate NVIC enabled |

Outputs

None

### p\_interrupt\_control\_get

Function Header

uint8\_t p\_interrupt\_control\_get(void)

Description

Read interrupt control

Inputs

0 = Alternate NVIC disabled, 1 = Alternate NVIC enabled

Outputs

None

## Interrupt NVIC Peripheral Functions

The list of Interrupt NVIC Peripheral Functions

* p\_interrupt\_nvic\_enable
* p\_interrupt\_nvic\_extEnables\_clr
* p\_interrupt\_nvic\_enpend\_clr
* p\_interrupt\_nvic\_priorities\_default\_set
* p\_interrupt\_nvic\_priorities\_set

### p\_interrupt\_nvic\_enable

Function Header

**void p\_interrupt\_nvic\_enable(IRQn\_Type nvic\_num, uint8\_t en\_flag)**

Description

Enable/Disable the NVIC IRQ in the NVIC interrupt controller

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| nvic\_num | NVIC number (see enum IRQn\_Type) |
| **en\_flag** | 1 = Enable the NVIC IRQ, 0 = Disable the NVIC IRQ |

Outputs

None

### p\_interrupt\_nvic\_extEnables\_clr

Function Header

**void p\_interrupt\_nvic\_extEnables\_clr(void)**

Description

Clear all NVIC external enables

Inputs

None

Outputs

None

### p\_interrupt\_nvic\_enpend\_clr

Function Header

**void p\_interrupt\_nvic\_enpend\_clr(void)**

Description

Clear all NVIC external enables and pending bits

Inputs

None

Outputs

None

### p\_interrupt\_nvic\_ priorities\_default\_set

Function Header

**void p\_interrupt\_nvic\_priorities\_default\_set(void)**

Description

Set NVIC external priorities to POR value

Inputs

None

Outputs

None

### p\_interrupt\_nvic\_priorities\_set

Function Header

**void p\_interrupt\_nvic\_priorities\_set(uint8\_t new\_pri)**

Description

Set NVIC external priorities to specified priority (0 — 7)

NVIC highest priority is the value 0, lowest is all 1’s. Each external interrupt has an 8-bit register and the priority is left justified in the registers. MECxxx implements 8 priority levels or bits [7:5] in the register. Lowest priority = 0xE0

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| new\_pri | New Priority |

Outputs

None

# Hibernation Timer

## Hibernation Timer APIs

The list of Hibernation Timer APIs

* htimer\_enable
* htimer\_disable
* htimer\_reload

### htimer\_enable

Function Header

**void htimer\_enable(uint8\_t htimer\_id, uint16\_t preload\_value, uint8\_t resolution\_mode)**

Description

This function enables the hibernation timer by programming the preload value.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **htimer\_id** | Hibernation Timer ID |
| preload\_value | 16-bit preload count value |
| resolution mode | 0 – The Hibernation Timer has a resolution of 30.5us per LSB, which yield a maximum time of ~2 seconds.  1 — The Hibernation Timer has a resolution of 0.125s per LSB, which yield a maximum time in excess of 2 hours. |

Outputs

None

### htimer\_disable

Function Header

**void htimer\_disable(uint8\_t htimer\_id)**

Description

This function disables the hibernation timer by programming the preload value as 0.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **htimer\_id** | Hibernation Timer ID |

Outputs

None

### htimer\_reload

Function Header

**void htimer\_reload(uint8\_t htimer\_id, uint16\_t reload\_value)**

Description

This function programs new preload value for the Hibernation Timer.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **htimer\_id** | Hibernation Timer ID |
| reload\_value | 16-bit reload count value |

Outputs

None

## Hibernation Timer Peripheral functions

The list of Hibernation Timer Peripheral functions

* p\_htimer\_preload\_set
* p\_htimer\_resolution\_set
* p\_htimer\_count\_get

### p\_htimer\_preload\_set

Function Header

**void p\_htimer\_preload\_set(uint8\_t htimer\_id, uint16\_t preload\_value)**

Description

This function is used to set the Hibernation Timer 16-bit Preload value.

Note: Setting the preload with a non-zero value starts the hibernation timer to down count.

Setting the preload to 0 disables the hibernation counter.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **htimer\_id** | Hibernation Timer ID |
| preload\_value | 16-bit preload count value |

Outputs

None

### htimer\_resolution\_set

Function Header

**void p\_htimer\_resolution\_set(uint8\_t htimer\_id, uint8\_t resolution\_mode)**

Description

This function is used to set the Hibernation Timer resolution.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **htimer\_id** | Hibernation Timer ID |
| resolution mode | 0 – The Hibernation Timer has a resolution of 30.5us per LSB, which yield a maximum time of ~2 seconds.  1 — The Hibernation Timer has a resolution of 0.125s per LSB, which yield a maximum time in excess of 2 hours. |

Outputs

None

### htimer\_count\_get

Function Header

**uint16\_t p\_htimer\_count\_get(uint8\_t htimer\_id)**

Description

This function returns the Hibernation Timer current count value

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **htimer\_id** | Hibernation Timer ID |

Outputs

16-bit count value

# RTC

## RTC Peripheral Functions

List of RTC peripheral Functions

* Get and Set seconds
  + P\_RTC\_seconds\_set
  + P\_RTC\_seconds\_get
* Get and Set Minutes
  + P\_RTC\_minutes\_set
  + P\_RTC\_minutes\_get
* Get and Set Hours
  + P\_RTC\_ hour\_ set
  + P\_RTC \_hour\_get
  + P\_RTC\_hour\_ampm\_get
* Get and Set Day of Week
  + P\_RTC \_dayofweek\_set
  + P\_RTC \_dayofweek\_get
* Get and Set Day of Month
  + P\_RTC \_dayofmonth\_set
  + P\_RTC \_dayofmonth\_get
* Get and Set Month
  + P\_RTC\_month\_set
  + P\_RTC\_month\_get
* Get and Set Year
  + P\_RTC \_year\_set
  + P\_RTC \_year\_get
* Set Alarm
  + P\_RTC \_seconds\_alarm\_set
  + P\_RTC \_minutes\_alarm\_set
  + P\_RTC \_hour\_alarm\_set
  + P\_RTC \_dayofweek\_alarm\_set
  + P\_RTC\_month\_alarm\_set
* RTC Control and Flags
  + P\_RTC\_Enable
  + P\_RTC\_SleepEnable
  + P\_RTC\_HostClk
  + P\_RTC\_Reset
  + P\_RTC\_alarm\_enable
  + P\_RTC\_ReadIntFlags
* Daylight savings
  + P\_RTC\_DaylightSavingsforward
  + P\_RTC\_DaylightSavingsBackward
* Time Datamode
  + P\_RTC \_datamode\_get
  + P\_RTC \_datamode \_set
* Time Format
  + P\_RTC \_hourformat\_set
  + P\_RTC \_hourformat\_get
* Update Busy
  + P\_RTC\_updateBusy

### p\_RTC \_seconds\_set

Function header

Uint8\_t p\_RTC \_seconds\_set (uint8\_t seconds);

Description

Sets the seconds of RTC

Inputs

|  |  |
| --- | --- |
| Input parameter | Description |
| Seconds | An unsigned 8 bit integer specifying the seconds of RTC. The values can be of the range 0 to 59. |

Output

RTC\_UPDATE\_SUCCESS – the requested operation was successful

RTC\_UPDATE\_FAIL – Unable to process requested operation

RTC\_HW\_BUSY – Unable to update value because RTC H/W is busy

### p\_RTC \_seconds\_get

Function Header

Uint8\_t p\_RTC \_seconds\_get (void);

Description

Get the seconds of RTC

Inputs

None

Output

Returns the value of seconds reflected by the RTC registers

### P\_RTC \_minutes\_set

Function header

Uint8\_t p\_RTC \_minutes\_set (uint8\_t minutes);

Description

Sets the minutes of RTC

Inputs

|  |  |
| --- | --- |
| Input parameter | Description |
| Seconds | An unsigned 8 bit integer specifying the minutes of RTC. The values can be of the range 0 to 59. |

Output

RTC\_UPDATE\_SUCCESS – the requested operation was successful

RTC\_UPDATE\_FAIL – Unable to process requested operation

RTC\_HW\_BUSY – Unable to update value because RTC H/W is busy

### p\_RTC \_minutes\_get

Function Header

Uint8\_t p\_RTC \_minutes\_get (void);

Description

Returns the minutes of RTC

Inputs

None

Output

Returns the value of minutes reflected by the RTC registers

### p\_RTC \_hour\_set

Function header

Uint8\_t p\_RTC \_hour\_set(uint8\_t hour, uint8\_t ampmMode);

Description

Sets the hour value of RTC

Inputs

|  |  |
| --- | --- |
| Input parameter | Description |
| hour | An unsigned 8 bit integer specifying the hour value of RTC. Based on the hour format, the value can range from 1 to 12 (for 12-hour mode) or 0-23 (for 24-hour mode) |
| ampmMode | An unsigned 8 bit integer specifying the AM/PM mode. The permitted values are  RTC\_HOUR\_AM  RTC\_HOUR\_PM  RTC\_HOUR\_MODE24 (for 24 hour mode, am/pm is immaterial) |

Output

RTC\_UPDATE\_SUCCESS – the requested operation was successful

RTC\_UPDATE\_FAIL – Unable to process requested operation

RTC\_HW\_BUSY – Unable to update value because RTC H/W is busy

### p\_RTC \_hour\_get

Function Header

Uint8\_t p\_RTC \_hour\_get (void);

Description

Get the hour value of RTC

Inputs

None

Output

Returns the value of hour reflected by the RTC registers

### p\_RTC\_hour\_ampm\_get

Function Header

Uint8\_t p\_RTC\_hour\_ampm\_get(void)

Description

Returns the HOUR\_AM\_PM value in RTC Registers.

Input

None

Output

An unsigned 8 bit integer which the HOUR\_AM\_PM value of RTC registers. If 0, it indicates AM and if 1 it indicates PM.

### p\_RTC \_dayofweek\_set

Function header

Uint8\_t p\_RTC \_dayofweek\_set (DAYS dayofweek);

Description

Sets the day of week of RTC

Inputs

|  |  |
| --- | --- |
| Input parameter | Description |
| dayofweek | An enumerated type indicating the day of the week.  Enum DAYS{SUNDAY = 1 , MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY}; |

Output

RTC\_UPDATE\_SUCCESS – the requested operation was successful

RTC\_UPDATE\_FAIL – Unable to process requested operation

RTC\_HW\_BUSY – Unable to update value because RTC H/W is busy

### p\_RTC \_dayofweek\_get

Function Header

DAYS p\_RTC \_dayofweek\_get (void);

Description

Get the day of the week from RTC registers.

Inputs

None

Output

Returns an enumerated type reflecting the day of the week of RTC registers.

### p\_RTC \_dayofmonth\_set

Function header

Uint8\_t p\_RTC \_dayofmonth\_set (uint8\_t dayofmonth);

Description

Sets the day of month of RTC

Inputs

|  |  |
| --- | --- |
| Input parameter | Description |
| dayofmonth | An unsigned 8 bit integer that indicates the day of the month. The values can range from 1 to 31. |

Output

RTC\_UPDATE\_SUCCESS – the requested operation was successful

RTC\_UPDATE\_FAIL – Unable to process requested operation

RTC\_HW\_BUSY – Unable to update value because RTC H/W is busy

### p\_RTC \_dayofmonth\_get

Function Header

Uint8\_t p\_RTC \_dayofmonth\_get (void);

Description

Get the day of the month of RTC

Inputs

None

Output

An unsigned 8 bit integer that indicates the day of the month. The values can range from 1 to 31.

### p\_RTC\_month\_set

Function header

Uint8\_t p\_RTC\_month\_set(uint8\_t month)

Description

Sets the month value in RTC Registers.

Input

|  |  |
| --- | --- |
| Input parameter | Description |
| month | An unsigned 8 bit integer that indicates the month. The values can range from 1 to 31. |

Output

RTC\_UPDATE\_SUCCESS – the requested operation was successful

RTC\_UPDATE\_FAIL – Unable to process requested operation

RTC\_HW\_BUSY – Unable to update value because RTC H/W is busy

### p\_RTC\_month\_get

Function Header

Uint8\_t p\_RTC\_month\_get(void)

Description

Gets the month value from RTC registers.

Input

None

Output

Returns the month value from RTC registers.

### p\_RTC \_year\_set

Function Header

Uint8\_t p\_RTC \_year\_set(uint8\_t year);

Description

Set the year value in RTC Regsiters.

Inputs

|  |  |
| --- | --- |
| Input parameter | Description |
| year | An unsigned 8 bit integer indicating the year. The value can range from 0 (2000) to 99 (2099) |

Outputs

RTC\_UPDATE\_SUCCESS – the requested operation was successful

RTC\_UPDATE\_FAIL – Unable to process requested operation

RTC\_HW\_BUSY – Unable to update value because RTC H/W is busy

### p\_RTC \_year\_get

Function Header

uint16\_t p\_RTC \_year\_get(void);

Description

Get the year value from RTC registers.

Inputs

None

Outputs

An unsigned 8 bit integer that returns the year value of RTC. . The value can range from 0 (2000) to 99 (2099).

### p\_RTC \_seconds\_alarm\_set

Function Header

uint8\_t p\_RTC \_seconds\_alarm\_set(uint8\_t seconds\_alarm);

Description

Programs the seconds’ value form which alarm must be triggered.

Inputs

|  |  |
| --- | --- |
| Input parameter | Description |
| seconds\_alarm | An unsigned 8 bit integer indicating the second value for which alarm should be triggered. The alarm value should be in the range 0 to 59. To disable this alarm, RTC\_ALARM\_DISABLE (0xC0) should be used. |

Output

RTC\_UPDATE\_SUCCESS – the requested operation was successful

RTC\_UPDATE\_FAIL – Unable to process requested operation

RTC\_HW\_BUSY – Unable to update value because RTC H/W is busy

### p\_RTC \_minutes\_alarm\_set

Function Header

uint8\_t p\_RTC\_set\_minutes\_alarm(uint8\_t minutes\_alarm);

Description

Programs the minutes’ value form which alarm must be triggered.

Inputs

|  |  |
| --- | --- |
| Input parameter | Description |
| minutes\_alarm | An unsigned 8 bit integer indicating the minute’s value for which alarm should be triggered. The alarm value should be in the range 0 to 59. To disable this alarm, RTC\_ALARM\_DISABLE (0xC0) should be used. |

Output

RTC\_UPDATE\_SUCCESS – the requested operation was successful

RTC\_UPDATE\_FAIL – Unable to process requested operation

RTC\_HW\_BUSY – Unable to update value because RTC H/W is busy

### p\_RTC \_hour\_alarm\_set

Function Header

uint8\_t p\_RTC \_hour\_alarm\_set(uint8\_t hour\_alarm);

Description

Programs the hour value form which alarm must be triggered.

Inputs

|  |  |
| --- | --- |
| Input parameter | Description |
| hour\_alarm | An unsigned 8 bit integer indicating the hour value for which alarm should be triggered. This value must be provided based on the way RTC is configured (12-Hour or 24-Hour Format). To disable this alarm, RTC\_ALARM\_DISABLE (0xC0) should be used. |

Output

RTC\_UPDATE\_SUCCESS – the requested operation was successful

RTC\_UPDATE\_FAIL – Unable to process requested operation

RTC\_HW\_BUSY – Unable to update value because RTC H/W is busy

### p\_RTC \_dayofweek\_alarm\_set

Function Header

uint8\_t p\_RTC \_dayofweek\_alarm\_set(uint8\_t dayofweek);

Description

Programs the week value form which alarm must be triggered.

Inputs

|  |  |
| --- | --- |
| Input parameter | Description |
| dayofweek | An unsigned 8 bit integer indicating the week value for which alarm should be triggered. To disable this alarm, RTC\_ALARM\_DISABLE (0xC0) should be used. |

Output

RTC\_UPDATE\_SUCCESS – the requested operation was successful

RTC\_UPDATE\_FAIL – Unable to process requested operation

RTC\_HW\_BUSY – Unable to update value because RTC H/W is busy

### p\_RTC\_month\_alarm\_set

Function Header

Uint8\_t p\_RTC\_month\_alarm\_set(uint8\_t month\_alarm)

Description

Configure the month alarm value in RTC registers.

Input

|  |  |
| --- | --- |
| Input parameter | Description |
| month\_alarm | An unsigned 8 bit integer indicating the month value for which alarm should be triggered. To disable this alarm, RTC\_ALARM\_DISABLE (0xC0) should be used. |

Output

RTC\_UPDATE\_SUCCESS – the requested operation was successful

RTC\_UPDATE\_FAIL – Unable to process requested operation

RTC\_HW\_BUSY – Unable to update value because RTC H/W is busy

### p\_RTC\_Enable

Function Header

Void p\_RTC\_enable(bool En)

Description

Enables or disables the block by setting the Block Enable bit in RTC Control register.

Input

|  |  |
| --- | --- |
| Input parameter | Description |
| En | A Boolean value to control enable disable of RTC block. If block is to be enabled, set the value to true, false otherwise. |

Output

None

### p\_RTC\_SleepEnable

Function Header

Void p\_RTC\_sleep(bool En)

Description

Triggers sleep mode of RTC by setting the Sleep bit in PCR registers.

Input

|  |  |
| --- | --- |
| Input parameter | Description |
| En | A Boolean value to control enable/disable sleep mode of RTC block. If sleep is to be enabled, set the value to true, false otherwise. |

Output

None

### p\_RTC\_HostClk

Function Header

Bool p\_RTC\_HostClk(void)

Description

Checks if the RTC block requires host clock.

Input

None

Output

Returns true if Host clock is required, false otherwise.

### p\_RTC\_Reset

Function Header

Void p\_RTC\_reset(void)

Description

Assert the soft reset of RTC. The Soft Reset bit is self-clearing and need not be cleared.

Input

None

Output

None

### p\_RTC\_alarm\_enable

Function Header

Void p\_RTC\_alarm\_enable(bool En, bool Ien)

Description

Enable or disables alarms and associated interrupts

Input

|  |  |
| --- | --- |
| Input parameter | Description |
| En | A Boolean value to control enable/disable alarms of RTC block. If alarm is to be enabled, set the value to true, false otherwise. |
| Ien | A Boolean value to control enable/disable interrupts associated with alarms of RTC block. If interrupt is to be enabled, set the value to true, false otherwise. |

Output

None

### p\_RTC\_ReadIntFlags

Function Header

Uint8\_t p\_RTC\_ReadIntFlags(void)

Description

Reads and returns the interrupt flags of RTC. The register is read-clear. So, once the status is read, the flags are cleared.

Input

None

Output

An 8 bit unsigned integer reflecting the interrupt flags of RTC. The bit definition of the return value is presented below.

|  |  |
| --- | --- |
| Bit Number | Description |
| 0-3 | Not Applicable |
| 4 | UPDATE\_ENDED\_INTERRUPT\_FLAG |
| 5 | ALARM\_FLAG |
| 6 | PERIODIC\_INTERRUPT\_FLAG |
| 7 | INTERRUPT\_REQUEST\_FLAG |

### p\_RTC\_daylight\_savings\_forward

Function Header

Uint8\_t p\_RTC\_daylight\_savings\_forward(DAYLIGHT\_SAVINGS forward)

Description

Loads data into registers for changing daylight savings forward.

Input

|  |  |
| --- | --- |
| Input parameter | Description |
| forward | A parameter of the predefined type DAYLIGHT\_SAVINGS wherein the values will reflect the amount of adjustment in time required.  The description of DAYLIGHT\_SAVINGS is presented below  struct DAYLIGHT\_SAVINGS  {  Bool am; // if true, sets the time to AM, else sets the time to PM  Uint8\_t hour; // hour value required for daylight saving compensation  Uint8\_t week; // week value required for daylight saving compensation  DAYS dayofweek; // day value required for daylight saving compensation  Uint8\_t month; // month value required for daylight saving compensation  }; |

Output

--Dunno—

### p\_RTC\_daylight\_savings\_backward

Function Header

uint8\_t p\_RTC\_daylight\_savings\_backward(DAYLIGHT\_SAVINGS backward)

Description

Loads data into registers for changing daylight savings backward.

Input

|  |  |
| --- | --- |
| Input parameter | Description |
| forward | A parameter of the predefined type DAYLIGHT\_SAVINGS wherein the values will reflect the amount of adjustment in time required.  The description of DAYLIGHT\_SAVINGS is presented below  struct DAYLIGHT\_SAVINGS  {  Bool am; // if true, sets the time to AM, else sets the time to PM  Uint8\_t hour; // hour value required for daylight saving compensation  Uint8\_t week; // week value required for daylight saving compensation  DAYS dayofweek; // day value required for daylight saving compensation  Uint8\_t month; // month value required for daylight saving compensation  }; |

Output

--Dunno—

### p\_RTC \_datamode\_get

Function header

bool p\_RTC \_datamode\_get(void)

Description

Returns the datamode configured in RTC.

Inputs

None

Output

Returns a Boolean value which, if true, indicates that the data mode is Binary, if false indicates that the data mode is in BCD.

### p\_RTC \_datamode\_set

Function Header

Void p\_RTC \_datamode\_set(bool format)

Description

Sets the data mode of RTC.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| Format | A Boolean value which, if true, indicates that the data mode should be set to Binary. If False, the data mode should be set to BCD. |

Output

None

### p\_RTC \_hourformat\_set

Function header

Void p\_RTC \_hourformat\_set(bool format)

Description

Sets the hour format to either 24-Hour mode or 12-Hour Mode.

Input

|  |  |
| --- | --- |
| Input Parameter | Description |
| Format | A Boolean value, if true, indicates that the hour format should be set to 24-hour and if false, indicates that the hour format should be set to 12-hour. |

Output

None

### p\_RTC\_get\_hourformat

Function Header

Bool p\_RTC \_hourformat\_get(void)

Description

Get the hour format configured in RTC.

Inputs

None

Outputs

Returns a Boolean value, if true, indicates that the hour format is 24-hour and if false, indicates that the hour format is 12-hour.

### p\_RTC\_DaylightSavingsForward

Function Header

Void p\_RTC\_DaylightSavingsForward(DAYLIGHT\_SAVINGS forward)

Description

Configures the daylight savings forward registers of RTC based on input provided

Input

|  |  |
| --- | --- |
| Input Parameter | Description |
| forward | A data structure of the type DAYLIGHT\_SAVINGS. Its description is presented below.  Struct DAYLIGHT\_SAVINGS {  uint8\_t am\_pm;  uint8\_t hour;  uint8\_t week;  DAYS dayofweek;  uint8\_t month; }; |

Output

None

### p\_RTC\_DaylightSavingsForward

Function Header

Void p\_RTC\_DaylightSavingsForward(DAYLIGHT\_SAVINGS forward)

Description

Configures the daylight savings forward registers of RTC based on input provided.

Input

|  |  |
| --- | --- |
| Input Parameter | Description |
| forward | A data structure of the type DAYLIGHT\_SAVINGS. Its description is presented below.  Struct DAYLIGHT\_SAVINGS {  uint8\_t am\_pm;  uint8\_t hour;  uint8\_t week;  DAYS dayofweek;  uint8\_t month; }; |

Output

None

### p\_RTC\_DaylightSavingsBackward

Function Header

Void p\_RTC\_DaylightSavingsBackward(DAYLIGHT\_SAVINGS backward)

Description

Configures the daylight savings backward registers of RTC based on input provided.

Input

|  |  |
| --- | --- |
| Input Parameter | Description |
| backward | A data structure of the type DAYLIGHT\_SAVINGS. Its description is presented below.  Struct DAYLIGHT\_SAVINGS {  uint8\_t am\_pm;  uint8\_t hour;  uint8\_t week;  DAYS dayofweek;  uint8\_t month; }; |

Output

Non

## RTC API’s

### RTC\_init

Function Header

void\_t RTC\_init(void)

Description

This api initializes the RTC by configuring datamode, hourformat and enables RTC Block.

Input

None

Output

None

### RTC\_start

Function Header

void RTC\_start(void)

Description

Disables sleep mode of RTC

Inputs

None

Output

None

### RTC\_sleep

Function Header

void RTC\_sleep(void)

Description

Suspends RTC Peripheral activity.

Input

None

Ouput

None

### RTC\_time\_set

Function Header

Uint8\_t RTC\_time\_set(TIME time)

Description

Configures the time of RTC

Input

|  |  |
| --- | --- |
| Input Parameters | Description |
| Time | A Parameter of the type TIME. The description of TIME is given below  struct TIME{  Uint8\_t seconds; // values from 0 to 59  Uint8\_t minutes; // values from 0 to 59  Uint8\_t hour; // based on hourformat, vales can be 1-12 or 0-23  Uint8\_t ampm\_mode; // Values can be RTC\_HOUR\_AM, RTC\_HOUR\_PM, RTC\_HOUR\_MODE24  } |

Output

RTC\_SUCCESS – If the requested operations were successful.

RTC\_FAIL – If the requested operations could not be completed.

### RTC\_dayofweek\_set

Function Header

Uint8\_t RTC\_dayofweek\_set(DAYS day)

Description

Sets the day of the week in RTC Registers.

Input

|  |  |
| --- | --- |
| Input parameter | Description |
| dayofweek | An enumerated type indicating the day of the week.  Enum DAYS{SUNDAY = 1 , MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY}; |

Output

RTC\_SUCCESS – If the requested operations were successful.

RTC\_FAIL – If the requested operations could not be completed.

### RTC\_dayofweek\_get

Function Header

DAYS RTC\_dayofweek\_get(void)

Description

Gets the Day of the week from RTC Registers.

Input

None

Output

An enumerated type indicating the day of the week.

### RTC\_date\_set

Function Header

Uint8\_t RTC\_date\_set(DATE date)

Description

Configures the date of RTC.

Input

|  |  |
| --- | --- |
| Input Parameters | Description |
| date | A Parameter of the type DATE. The description of DATE is given below  struct DATE{  uint8\_t dayofmonth; // Values from 1 to 31  Uint8\_t month; // values from 1 to 12  Uint8\_t year; // values form 0(2000) to 99 (2099)  } |

Output

RTC\_SUCCESS – If the requested operations were successful.

RTC\_FAIL – If the requested operations could not be completed.

### RTC\_time\_get

Function Header

TIME RTC\_time\_get(void)

Description

Returns the time as reflected by RTC peripheral

Input

None

Output

Returns a Data of the type TIME which indicates the current time as reflected in the RTC peripheral.

### RTC\_date\_get

Function Header

DATE RTC\_date\_get(void)

Description

Gets the date from the RTC peripheral.

Input

None

Output

Returns a Data of the type DATE which indicates the current date as reflected in the RTC peripheral.

### RTC\_AlarmEventOccured

Function Header

bool RTC\_AlarmEventOccured(unsigned char \* status)

Description

This API checks if an alarm event has occurred. Once this API is called, the flag bits of RTC are cleared.

Input

|  |  |
| --- | --- |
| Input Parameters | Description |
| status | A pointer to an 8 bit integer which will be loaded with the interrupt flag statuses. The bit definitions are given below.   |  |  | | --- | --- | | Bit Number | Description | | 0-3 | Not Applicable | | 4 | UPDATE\_ENDED\_INTERRUPT\_FLAG | | 5 | ALARM\_FLAG | | 6 | PERIODIC\_INTERRUPT\_FLAG | | 7 | INTERRUPT\_REQUEST\_FLAG | |

Output

A Boolean value which if true indicates that an alarm even has occurred.

### RTC\_AlarmEnable

Function Header

Void RTC\_Alarm\_enable(bool Enable)

Description

Enables or disables alarm and associated interrupts.

Input

|  |  |
| --- | --- |
| Input Parameters | Description |
| Enable | A Boolean parameter to indicate whether alarm and its associated interrupts are to be enabled or disabled. |

Output

None

### RTC\_AlarmSet

Function Header

Void RTC\_AlarmSet(ALARM value)

Description

Configures Alarms of RTC. Alarm needs to be enabled with a call to RTC\_AlarmEnable.

Input

|  |  |
| --- | --- |
| Input Parameters | Description |
| value | Data of the type ALARM specifying alarm values. The description of ALARM type is given below.  Struct ALARM{  Uint8\_t seconds;  Uint8\_t minutes;  Uint8\_t hours;  Uint8\_t hours;  Uint8\_t dayofweek;  Uint8\_t month;  };  To disable any of the alarms, load RTC\_ALARM\_DISABLE. |

Output

None

### RTC\_DaylightsavingConfig

Function Header

Uint8\_t RTC\_DaylightsavingConfig(DAYLIGHT\_SAVINGS dsCfg, bool forward)

Description

Configures daylight saving compensation of RTC

Input

|  |  |
| --- | --- |
| Input Parameter | Description |
| dsCfg | A data structure of the type DAYLIGHT\_SAVINGS. Its description is presented below.  Struct DAYLIGHT\_SAVINGS {  uint8\_t am\_pm;  uint8\_t hour;  uint8\_t week;  DAYS dayofweek;  uint8\_t month; }; |
| forward | A Boolean value which if true, configures RTC daylight savings forward. If false, configures RTC daylight savings backward. |

Output:

RTC\_SUCCESS – If the requested operations were successful.

RTC\_FAIL – If the requested operations could not be completed.

# UART

UART API’s

Uart\_pins\_init

uart\_hw\_init

uart\_protocol\_init

uart\_transmit

uart\_receive

UART registers

Activate Register

Configuration register

Receive buffer register

Transmit buffer register

Programmable baud rate generator – byte 0

Programmable baud rate generator – byte 1

Interrupt enable register

Interrupt identification register

FIFO control register

Line control register

Line status register

Modem control register

Modem status register

Scratchpad register

UART Peripheral Functions

p\_uart\_enable\_disable

p\_uart\_config\_sel\_reg\_set

p\_uart\_config\_sel\_reg\_get

p\_uart\_baud\_clk\_src\_set

p\_uart\_rx\_buff\_read

p\_uart\_tx\_buff\_write

p\_uart\_baud\_divisor\_set

p\_uart\_interrupt\_enable\_reg\_set

p\_uart\_interrupt\_enable\_reg\_get

p\_uart\_iir\_reg\_get

p\_uart\_fifo\_control\_reg\_set

p\_uart\_line\_control\_reg\_set

p\_uart\_line\_control\_reg\_get

p\_uart\_break\_control\_set

p\_uart\_line\_status\_reg\_get

p\_uart\_modem\_control\_reg\_set

p\_uart\_modem\_control\_reg\_get

p\_uart\_modem\_status\_reg\_get

p\_uart\_scratchpad\_write

p\_uart\_scratchpad\_read

UART instances

UART 0

UART 1

PCR Peripheral Functions

GPIO Peripheral Functions

Interrupt Peripheral Functions

## UART APIs

The list of UART APIs

* uart\_pins\_init
* uart\_hw\_init
* uart\_protocol\_init
* uart\_transmit
* uart\_receive

### uart\_pins\_init

Function Header

**void uart\_pins\_init( uint8\_t uart\_id );**

Description

Initializes the gpio pins that are associated with the specified UART instance for UART functionality.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****uart\_id**** | 0-based UART ID |

Outputs

None

### uart\_hw\_init

Function Header

**void uart\_hw\_init( uint8\_t uart\_id, uint8\_t polarity, uint8\_t power, enum UART\_CLK\_SRC \**

**clock\_sel, uint16\_t baud, uint8\_t operation\_mode, uint8\_t fifo\_tggr\_lvl )**

Description

Initializes the uart block hardware instance and enables it.

Note - While using the non - fifo mode; keep the fifo trigger level parameter as

UART\_FIFO\_INT\_LVL\_1.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****uart\_id**** | 0-based UART ID |
| ****polarity**** | Polarity setting for the uart pins  UART\_CFG\_SEL\_POL\_INV  UART\_CFG\_SEL\_POL\_NON\_INV |
| ****power**** | Power source settings for the uart block  UART\_CFG\_SEL\_PWR\_VCC  UART\_CFG\_SEL\_PWR\_V3S5 |
| ****clock\_sel**** | Clock source for baud rate generation  UART\_CLK\_INT\_1P84MHz  UART\_CLK\_INT\_24MHz  UART\_CLK\_EXT |
| ****baud**** | Desired baud rate (Refer the header file) |
| ****operation\_mode**** | FIFO or non – FIFO mode  UART\_FIFO\_EN  UART\_FIFO\_DIS |
| ****fifo\_tggr\_lvl**** | Interrupt trigger level setting for FIFO mode  UART\_FIFO\_INT\_LVL\_1  UART\_FIFO\_INT\_LVL\_4  UART\_FIFO\_INT\_LVL\_8  UART\_FIFO\_INT\_LVL\_14 |

Outputs

None

### uart\_protocol\_init

Function Header

**void uart\_protocol\_init( uint8\_t uart\_id, uint8\_t wrd\_len, uint8\_t stp\_bit, uint8\_t parity\_type, \**

**enum INT\_TYPE interrupt\_type )**

Description

Initializes the serial protocol and interrupt parameters.

Note –

1. In case interrupts are not being used; keep the interrupt source type parameter value

as UART\_INT\_DISABLED.

1. Refer to the datasheet for the valid word length and stop bits combinations.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****uart\_id**** | 0-based UART ID |
| ****wrd\_len**** | Word length setting for RS 232 packet frame  UART\_WRD\_LEN\_5\_BITS  UART\_WRD\_LEN\_6\_BITS  UART\_WRD\_LEN\_7\_BITS  UART\_WRD\_LEN\_8\_BITS |
| ****stp\_bit**** | Number of stop bits setting  UART\_STOP\_BIT\_1  UART\_STOP\_BIT\_1P5\_OR\_2 |
| ****parity\_type**** | Type of parity check setting  UART\_PARITY\_BIT\_AS\_SPACE  UART\_PARITY\_BIT\_AS\_MARK  UART\_PARITY\_AS\_EVEN  UART\_PARITY\_AS\_ODD  UART\_PARITY\_BIT\_NONE |
| ****Interrupt\_type**** | Types of interrupts to be enabled  UART\_RX\_DATA\_AVAILABLE  UART\_TX\_BUFF\_EMPTY  UART\_RX\_LINE\_STS  UART\_MODEM\_STS  UART\_RX\_TX\_BUFF  UART\_ALL\_INT\_EN  UART\_INT\_DISABLED |

Outputs

None

### uart\_transmit

Function Header

**void uart\_transmit( uint8\_t uart\_id, uint8\_t data )**

Description

Transmits serial data using the specified uart instance..

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****uart\_id**** | 0-based UART ID |
| ****data**** | Data character to be sent |

Outputs

None

### uart\_receive

Function Header

**uint8\_t uart\_receive( uint8\_t uart\_id )**

Description

Receives serial data using the specified uart instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****uart\_id**** | 0-based UART ID |

Outputs

Data received over uart

## UART peripheral functions

The list of UART peripheral functions is –

* p\_uart\_enable\_disable
* p\_uart\_config\_sel\_reg\_set
* p\_uart\_config\_sel\_reg\_get
* p\_uart\_baud\_clk\_src\_set
* p\_uart\_rx\_buff\_read
* p\_uart\_tx\_buff\_write
* p\_uart\_baud\_divisor\_set
* p\_uart\_interrupt\_enable\_reg\_set
* p\_uart\_interrupt\_enable\_reg\_get
* p\_uart\_fifo\_control\_reg\_set
* p\_uart\_iir\_reg\_get
* p\_uart\_line\_control\_reg\_set
* p\_uart\_line\_control\_reg\_get
* p\_uart\_break\_control\_set
* p\_uart\_line\_status\_reg\_get
* p\_uart\_modem\_control\_reg\_set
* p\_uart\_modem\_control\_reg\_get
* p\_uart\_modem\_status\_reg\_get
* p\_uart\_scratchpad\_write
* p\_uart\_scratchpad\_read

### p\_uart\_enable\_disable

Function Header

**void p\_uart\_enable\_disable( uint8\_t uart\_id, uint8\_t new\_val )**

Description

Enables or disables the uart hardware block

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****uart\_id**** | 0-based UART ID |
| ****new\_val**** | UART enable/disable setting  UART\_BLOCK\_EN  UART\_BLOCK\_DIS |

Outputs

None

### p\_uart\_config\_sel\_reg\_set

Function Header

**void p\_uart\_config\_sel\_reg\_set( uint8\_t uart\_id, uint8\_t new\_val )**

Description

Writes to the configuration select register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****uart\_id**** | 0-based UART ID |
| ****new\_val**** | New configuration value |

Outputs

None

### p\_uart\_config\_sel\_reg\_get

Function Header

**uint8\_t p\_uart\_config\_sel\_reg\_get( uint8\_t uart\_id )**

Description

Reads the configuration select register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****uart\_id**** | 0-based UART ID |

Outputs

Current register contents, 0xFF - read failed

### p\_uart\_baud\_clk\_src\_set

Function Header

**void p\_uart\_baud\_clk\_src\_set( uint8\_t uart\_id, uint8\_t new\_val )**

Description

Configures the clock source for baud rate generation of the specified uart instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****uart\_id**** | 0-based UART ID |
| ****new\_val**** | Clock source setting for the UART block  UART\_BAUD\_CLK\_24MHz  UART\_BAUD\_CLK\_1P84MHz |

Outputs

None

### p\_uart\_rx\_buff\_read

Function Header

**uint8\_t p\_uart\_rx\_buff\_read( uint8\_t uart\_id )**

Description

Reads the receive buffer register of the specified uart instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****uart\_id**** | 0-based UART ID |

Outputs

Contents of the receive data buffer

### p\_uart\_tx\_buff\_write

Function Header

**void p\_uart\_tx\_buff\_write( uint8\_t uart\_id, uint8\_t new\_val )**

Description

Writes to the tx buffer of the specified uart instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****uart\_id**** | 0-based UART ID |
| ****new\_val**** | Data to be transmitted |

Outputs

None

### p\_uart\_baud\_divisor\_set

Function Header

**void p\_uart\_baud\_divisor\_set( uint8\_t uart\_id, uint16\_t baud )**

Description

Function to set the baud rate divisor value for the specified uart instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****uart\_id**** | 0-based UART ID |
| ****baud**** | Desired baud rate value (Refer the header file) |

Outputs

None

### p\_uart\_interrupt\_enable\_reg\_set

Function Header

**void p\_uart\_interrupt\_enable\_reg\_set( uint8\_t uart\_id, uint8\_t new\_val )**

Description

Writes to the interrupt enable register of the specified uart instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****uart\_id**** | 0-based UART ID |
| ****new\_val**** | New configuration value |

Outputs

None

### p\_uart\_interrupt\_enable\_reg\_get

Function Header

**uint8\_t p\_uart\_interrupt\_enable\_reg\_get( uint8\_t uart\_id )**

Description

Reads the contents of interrupt enable register of the specified uart instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****uart\_id**** | 0-based UART ID |

Outputs

Contents of the interrupt enable register, 0xFF – read failed

### p\_uart\_iir\_reg\_get

Function Header

**uint8\_t p\_uart\_iir\_reg\_get( uint8\_t uart\_id )**

Description

Reads the contents of iir register of the specified uart instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****uart\_id**** | 0-based UART ID |

Outputs

Contents of the IIR register, 0xFF – read failed.

### p\_uart\_fifo\_control\_reg\_set

Function Header

**void p\_uart\_fifo\_control\_reg\_set( uint8\_t uart\_id, UART\_FIFO config\_type, uint8\_t new\_val )**

Description

Writes to the fifo control register of the specified uart instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****uart\_id**** | 0-based UART ID |
| ****config\_type**** | Configuration that needs to be changed  EN\_DIS\_FIFO – enable/disable fifo mode  CLR\_RCV\_FIFO – clear rx fifo  CLR\_XMIT\_FIFO – clear tx fifo  FIFO\_TRGGR\_LVL – set fifo trigger level  FIFO\_ALL – update all configurations |
| ****new\_val**** | New configuration data |

Outputs

None.

### p\_uart\_line\_control\_reg\_set

Function Header

**void p\_uart\_line\_control\_reg\_set( uint8\_t uart\_id, uint8\_t new\_val )**

Description

Writes to the line control register of the specified uart instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****uart\_id**** | 0-based UART ID |
| ****new\_val**** | New configuration data |

Outputs

None.

### p\_uart\_line\_control\_reg\_get

Function Header

**uint8\_t p\_uart\_line\_control\_reg\_get( uint8\_t uart\_id )**

Description

Reads the contents of line control register of the specified uart instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****uart\_id**** | 0-based UART ID |

Outputs

Current register contents.

### p\_uart\_break\_control\_set

Function Header

**void p\_uart\_break\_control\_set( uint8\_t uart\_id, uint8\_t new\_val )**

Description

Configures the uart to enable/disable break control.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****uart\_id**** | 0-based UART ID |
| ****new\_val**** | New configuration setting  UART\_BRK\_CNTRL\_EN  UART\_BRK\_CNTRL\_DIS |

Outputs

None.

### p\_uart\_line\_status\_reg\_get

Function Header

**uint8\_t p\_uart\_line\_status\_reg\_get( uint8\_t uart\_id, enum LINE\_STS\_TYPE flag )**

Description

Reads the contents of line status register of the specified uart instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****led\_id**** | 0-based LED ID |
| ****flag**** | Type of the status which is to be read  DATA\_READY  OVERRUN\_ERROR  PARITY\_ERROR  FRAME\_ERROR  BREAK\_INTERRUPT  TRANSMIT\_HOLDING\_REG\_EMPTY  TRANSMIT\_ERROR  FIFO\_ERROR  LINE\_STS\_ALL |

Outputs

Current register contents.

### p\_uart\_modem\_control\_reg\_set

Function Header

**void p\_uart\_modem\_control\_reg\_set( uint8\_t uart\_id, enum MODEM\_CTRL\_TYPE param,**

**uint8\_t new\_val )**

Description

Writes to the modem control register of the specified uart instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****uart\_id**** | 0-based UART ID |
| ****param**** | Type of the parameter that needs to be changed  DTR  RTS  OUT1  OUT2  LOOPBACK  MCR\_ALL – writes to the whole register |
| ****new\_val**** | New configuration data  **DTR**  UART\_MCR\_DTR\_SET  UART\_MCR\_DTR\_CLR  **RTS**  UART\_MCR\_RTS\_SET  UART\_MCR\_RTS\_CLR  **OUT1**  UART\_MCR\_OUT1\_EN  UART\_MCR\_OUT1\_DIS  **OUT2**  UART\_MCR\_OUT2\_EN  UART\_MCR\_OUT2\_DIS  **LOOPBACK**  UART\_MCR\_LOOPBACK\_EN  UART\_MCR\_LOOPBACK\_DIS  **MCR\_ALL**  Use combination of the above values |

Outputs

None.

### p\_uart\_modem\_control\_reg\_get

Function Header

**uint8\_t p\_uart\_modem\_control\_reg\_get( uint8\_t uart\_id )**

Description

Reads the contents of modem control register of the specified uart instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****uart\_id**** | 0-based UART ID |

Outputs

Current register contents, 0xFF – read failed

### p\_uart\_modem\_status\_reg\_get

Function Header

**uint8\_t p\_uart\_modem\_status\_reg\_get( uint8\_t uart\_id, enum MODEM\_STS\_TYPE flag )**

Description

Reads the contents of modem status register of the specified uart instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****uart\_id**** | 0-based uart ID |
| ****flag**** | Status which is to be read  CTS  DSR  RI  DCD  nCTS  nDSR  nRI  nDCD  MODEM\_STS\_ALL |

Outputs

Current register contents

### p\_uart\_scratchpad\_write

Function Header

**void p\_uart\_scratchpad\_write( uint8\_t uart\_id, uint8\_t new\_val )**

Description

Writes to the scratchpad register of the specified uart instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****uart\_id**** | 0-based UART ID |
| ****new\_val**** | New configuration data |

Outputs

None.

### p\_uart\_scratchpad\_read

Function Header

**uint8\_t p\_uart\_scratchpad\_read( uint8\_t uart\_id )**

Description

Reads the contents of scratchpad register of the specified uart instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****uart\_id**** | 0-based UART ID |

Outputs

Current register contents.

# QMSPI Functions

### spi\_port\_sel

Function Header:

void spi\_port\_sel (uint8\_t port, uint8\_t pin\_mask, bool en);

Description:

This function controls SPI port control. It facilitates the selection of ports and offers enable/disable control. By selection of ports, the GPIO’s and chip selects are configured as necessary.

If any port numbers other that the one’s mentioned below are used, the function will not perform any operation.

Inputs:

|  |  |
| --- | --- |
| Input Parameter | Description |
| Port | An 8 bit unsigned integer indicating port number. The permitted port numbers are   * 0 (Port 0, External shared) * 1 (Port 1, external private (Recovery)) * 2 (Port 2, Internal). |
| Pin\_mask | Specifies the pin(s) of the selected QMSPI port that needs to be modified  b[0]=chip-select, b[1]=clock, b[2]=IO0, b[3]=IO1, b[4]=IO2, b[5]=IO3. |
| En | A boolean input. The permitted values are   * 1 (Enable) * 0 (Disable) |

Outputs: None

### spi\_port\_drv\_slew

Function Header:

void spi\_port\_drv\_slew(uint8\_t port, uint8\_t pin\_mask , uint8\_t drv\_slew);

Description:

This function configures the drive strength and slew rate for GPIO’s based on selected port.

If any port numbers other that the one’s mentioned below are used, the function will not perform any operation.

Inputs:

|  |  |
| --- | --- |
| Input Parameters | Description |
| Port | An 8 bit unsigned integer indicating port number. The permitted port numbers are   * ROM\_PORT\_QMSPI\_SHD * ROM\_PORT\_QMSPI\_PVT * ROM\_PORT\_QMSPI\_INT |
| Pin\_mask | Specifies the pin(s) of the selected QMSPI port that needs to be modified  b[0]=chip-select, b[1]=clock, b[2]=IO0, b[3]=IO1, b[4]=IO2, b[5]=IO3. |
| Drv\_slew | An 8 bit unsigned integer indicating drv slew values. The permitted values for Drive strength and slew rate are Drive strength - GPIO\_DRV\_STR\_2MA, GPIO\_DRV\_STR\_4MA, GPIO\_DRV\_STR\_8MA, GPIO\_DRV\_STR\_12MA. Slew Rate - GPIO\_DRV\_SLEW\_SLOW, GPIO\_DRV\_SLEW\_FAST. The parameter drv\_slew corresponds to a hardware register. Please refer the User Manual of target device for description. |

Outputs: None

### rom\_dis\_lock\_shd\_spi

Function header:

void rom\_dis\_lock\_shd\_spi(uint8\_t lock\_shd\_spi);

Description:

Apply GPIO Locks as specified in customer section of EFUSE

Inputs:

|  |  |
| --- | --- |
| Input parameters | Description |
| lock\_shd\_spi | 0 (do not modify lock values)  1(insure Shared SPI GPIO's are disabled (tri-state input) and  these pins are locked. |

Outputs: None

### qmpsi\_init

Function Header:

void qmspi\_init(uint32\_t freqHz, uint8\_t spi\_signalling, uint8\_t ifctrl);

Description:

This function configures the frequency of SPI, the mode of operation and interface control.

The permitted frequencies for the SPI are 48 MHz, 24 MHz, 16 MHz, and 12 MHz

The SPI supports 4 modes of operation (SPI\_MODE\_0, SPI\_MODE\_1, SPI\_MODE\_2, SPI\_MODE\_3).

Inputs:

|  |  |
| --- | --- |
| Input parameters | Description |
| Freq\_hz | An unsigned 32 bit integer indicating frequency. The following frequencies are supported - 48 MHz, 24 MHz, 16 MHz, and 12 MHz. |
| Spi\_mode | An unsigned 8 bit integer indicating the mode. The following modes of operation are permitted   * SPI\_MODE\_0 * SPI\_MODE\_1 * SPI\_MODE\_2 * SPI\_MODE\_3. |
| If\_ctrl | An unsigned 8 bit integer indicating interface control. Refer the Data sheet of target for bit definitions. |

Macro values for Spi Modes field:

|  |  |
| --- | --- |
| Macro Name | Value |
| SPI\_MODE\_0 | (0x00u << 8) |
| SPI\_MODE\_1 | (0x06u << 8) |
| SPI\_MODE\_2 | (0x01u << 8) |
| SPI\_MODE\_3 | (0x07u << 8) |

Outputs: None

### qmspi\_freq\_get

Function Header:

uint32\_t qmspi\_freq\_get(void);

Description:

The function call is used to get the frequency of SPI.

Inputs: None

Outputs:

Returns the SPI operating frequency.

### qmspi\_freq\_set

Function Header:

void qmspi\_freq\_set (uint32\_t freq\_hz);

Description:

This function configures the frequency of SPI. The required frequency is passed to the function as an input parameter (freq\_hz). The permitted frequencies for the SPI are 48 MHz, 24 MHz, 16 MHz, and 12 MHz.

Inputs:

|  |  |
| --- | --- |
| Input Parameters | Description |
| Freq\_hz | A 32 bit unsigned integer indicating the required frequency of operation |

Outputs: None

### qmspi\_xfr\_done\_status

Function Header:

bool qmspi\_xfr\_done\_status(uint32\_t\* qmspi\_status);

Description:

This function gets the status of spi, updates the status into the pointer passed as argument, and returns the done status by evaluating the status register value. If done status is set, the bool value true is returned if not the value false is returned.

|  |  |
| --- | --- |
| Bit Number | Definition |
| 0 | XFR\_COMPLETE |
| 1 | DMA\_COMPLETE |
| 2 | TX\_BUFF\_ERR |
| 3 | RX\_BUFF\_ERR |
| 4 | PROG\_ERR |
| 8 | TX\_BUFF\_FULL |
| 9 | TX\_BUFF\_EMPTY |
| 10 | TX\_BUFF\_REQ |
| 11 | TX\_BUFF\_STALL |
| 12 | RX\_BUFF\_FULL |
| 13 | RX\_BUFF\_EMPTY |
| 15 | RX\_BUFF\_STALL |
| 16 | XFR\_ACTIVE |

Inputs:

|  |  |
| --- | --- |
| Input Parameters | Description |
| Qmspi\_status | A pointer to an unsigned 32 bit integer where the status of qmspi is stored |

Outputs:

TRUE if set, FALSE otherwise.

### qmspi\_start

Function Header:

void qmspi\_start(uint16\_t ien\_mask);

Description:

This function starts the SPI operation with the specified interrupt mask.

Inputs:

|  |  |
| --- | --- |
| Input Parameters | Description |
| Ien\_mask | An unsigned 16 bit integer specifying the interrupt mask. The bit definition of interrupt enable mask corresponds to Status register bit definitions mentioned in qmspi\_xfr\_done\_status. Refer data sheet for available interrupts. |

Outputs: None

### qmspi\_start\_dma

Function Header:

void qmspi\_start\_dma(uint8\_t dmach\_id, uint16\_t ien\_mask);

Description:

The function starts SPI operations along with a DMA channel. Ien\_mask represents the Interrupt Enable mask.

The dmach\_id is used to select the DMA Channel. There are 14 DMA channels and the channels along with their associated values are presented below.

|  |  |
| --- | --- |
| Channel name | Value |
| DMA\_CH00\_ID | 0 |
| DMA\_CH01\_ID | 1 |
| DMA\_CH02\_ID | 2 |
| DMA\_CH03\_ID | 3 |
| DMA\_CH04\_ID | 4 |
| DMA\_CH05\_ID | 5 |
| DMA\_CH06\_ID | 6 |
| DMA\_CH07\_ID | 7 |
| DMA\_CH08\_ID | 8 |
| DMA\_CH09\_ID | 9 |
| DMA\_CH10\_ID | 10 |
| DMA\_CH11\_ID | 11 |
| DMA\_CH12\_ID | 12 |
| DMA\_CH13\_ID | 13 |

Inputs:

|  |  |
| --- | --- |
| Input Parameters | Type |
| Dmach\_id | An 8 bit unsigned integer indicating the DMA channel. The available channels are present above. |
| Ien\_mask | An unsigned 16 bit integer specifying the interrupt mask. The bit definition of interrupt enable mask corresponds to Status register bit definitions mentioned in qmspi\_xfr\_done\_status. Refer data sheet for available interrupts. |

Outputs: None

### qmspi\_cfg\_spi\_cmd

Function Header:

uint8\_t qmspi\_cfg\_spi\_cmd(uint32\_t spi\_cmd, uint32\_t spi\_address);

Description:

This routine configures the QMSPI controller.

The bit definitions of the argument spi\_cmd are presented below.

1. b[7:0] = SPI op-code
2. b[15:8] = flags
3. b[9:8] = cmd bus width 0=1X, 1=2X, 2=4X
4. b[11:10] = address bus width
5. b[13:12] = data bus width
6. b[14] = 0 (24-bit address), 1(32-bit address)
7. b[15] = 1 use mode byte
8. b[23:16] = mode byte
9. b[31:24] = number of dummy clocks expressed as number of bytes where
   1. Clocks = bytes \* clocks/byte. Clocks per byte depend upon data bus width.
   2. Data bus width – 1X -> 8clocks/byte, 2X -> 4 clocks/byte, 4X -> 2 clocks/byte.
   3. Example: 4X 24bit read 0x6B requires 8 dummy clocks. At 2 clocks/byte, 4 bytes are required.

The SPI address can be either 24 bit address or 32 bit address.

Inputs:

|  |  |
| --- | --- |
| Input Parameter | Description |
| Spi\_cmd | An unsigned 32 bit integer. The bit definitions are presented above |
| Spi\_address | An unsigned 32 bit integer specifying the SPI address |

Outputs:

The function returns the ID of the Last Descriptor used (Descriptor is a Hardware register, refer data sheet for more details).

### qmspi\_read\_dma

Function Header:

uint32\_t qmspi\_ read\_dma( uint32\_t spi\_cmd,

uint32\_t spi\_address,

uint32\_t mem\_addr,

uint32\_t nbytes,

uint8\_t dmach\_id);

Description:

This routine configures the QMSPI controller to read a specified number of bytes form a specified address.

If nbytes is 0, the value returned will be zero.

If mem\_addr is specified as zero, the function will return a zero.

Inputs:

|  |  |
| --- | --- |
| Input Parameters | Description |
| Spi\_cmd | An unsigned 32 bit integer specifying the SPI Command. For spi\_cmd bit definitions, please refer qmspi\_cfg\_spi\_cmd. |
| Spi\_address | An unsigned 32 bit integer specifying the SPI address |
| Mem\_addr | An unsigned 32 bit integer which specifies the 32 bit address from where the data is to be read |
| Nbytes | An unsigned 32 bit integer which refers to the number of bytes to be read |
| Dmach\_id | An 8 bit unsigned integer which is used to refer to the DMA Channel to be used. Refer qmspi\_start\_dma section for description regarding dmach\_id. |

Outputs:

An unsigned 32 bit integer reflecting the number of bytes read.

### qmspi\_read\_fifo

Function Header:

Uint32\_t qmspi\_read\_fifo( uint8\_t \* data,

uint32\_t buff\_len);

Description:

The function is used to read data from the qmspi FIFO.

The number of bytes read will always be equal to or less than the buffer length specified.

Inputs:

|  |  |
| --- | --- |
| Input Parameters | Description |
| Data | An unsigned a-bit integer pointer to a buffer |
| Buff\_len | An unsigned 32 bit integer specifying the length |

Outputs:

The function returns a 32 bit value indicating the number of bytes read.

# Input Capture and Compare Timer

## ICCT APIs

The list of ICCT APIs

* icct\_freeRunningTimer\_init
* icct\_compare\_init
* icct\_capture\_init

### icct\_freeRunningTimer\_init

Function Header

**void icct\_freeRunningTimer\_init(uint8\_t tclk\_frequency, uint32\_t free\_running\_count)**

Description

Configure and enable Free Running Timer.

Note: This function resets and enables the free running timer.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **tclk\_frequency** | Clock source for the Free Running Counter |
| **free\_running\_count** | Free Running timer count value |

Use \_ICCT\_FREE\_RUN\_TMR\_TCLK\_ enum values from icct.h for tclk\_frequency

Outputs

None

### icct\_compare\_init

Function Header

**void icct\_compare\_init(uint8\_t cmp\_reg\_id, uint32\_t compare\_value, uint8\_t initial\_output\_state)**

Description

Initializes ICCT Compare

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **cmp\_reg\_id** | ICCT Compare ID |
| **compare\_value** | Compare Register Value |
| **initial\_output\_state** | Initial Compare O/P state |

use \_ICCT\_COMPARE\_INITIAL\_OUTPUT\_ enum macros from icct.h for initial\_output\_state

Outputs

None

### icct\_capture\_init

Function Header

**void icct\_capture\_init(uint8\_t cap\_reg\_id,**

**uint8\_t capture\_edge,**

**uint8\_t input\_filter\_byp,**

**uint8\_t input\_filter\_frequency)**

Description

Initializes ICCT Capture

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **cap\_reg\_id** | ICCT Capture ID |
| **capture\_edge** | Capture edge type - use enum \_ICCT\_CATPURE\_EDGE\_ |
| **input\_filter\_byp** | Enable/Bypass Input filter – use enum \_ICCT\_CAPTURE\_FILTER\_BYP\_ |
| **input\_filter\_frequency** | Input filter frequency – use enum \_ICCT\_CAPTURE\_FCLK\_SEL\_ |

Outputs

None

## ICCT Peripheral Functions

The list of ICCT APIs

* p\_icct\_activate
* p\_icct\_deactivate
* p\_icct\_freeRunningTimer\_reg\_read
* p\_icct\_freeRunningTimer\_reg\_write
* p\_icct\_freeRunningTimer\_tClk\_set
* p\_icct\_freeRunningTimer\_enable
* p\_icct\_freeRunningTimer\_disable
* p\_icct\_freeRunningTimer\_reset
* p\_icct\_freeRunningTimer\_reset\_read
* p\_icct\_compare\_reg\_read
* p\_icct\_compare\_reg\_write
* p\_icct\_compare\_timer\_enable
* p\_icct\_compare\_timer\_output\_clear
* p\_icct\_compare\_timer\_output\_set
* p\_icct\_compare\_timer\_output\_read
* p\_icct\_capture\_reg\_read
* p\_icct\_capture\_control\_reg\_write
* p\_icct\_capture\_control\_reg\_read

***Peripheral Functions for ICCT Activate***

### p\_icct\_activate

Function Header

**void p\_icct\_activate(void)**

Description

Activate ICCT block

Inputs

None

Outputs

None

### p\_icct\_deactivate

Function Header

void p\_icct\_deactivate(void)

Description

Deactivate ICCT block

Inputs

None

Outputs

None

***Peripheral Functions for Free Running Timer***

### p\_icct\_freeRunningTimer\_reg\_read

Function Header

**uint32\_t p\_icct\_freeRunningTimer\_reg\_read(void)**

Description

Read Free Running Timer Register

Inputs

None

Outputs

None

### p\_icct\_freeRunningTimer\_reg\_write

Function Header

**void p\_icct\_freeRunningTimer\_reg\_write(uint32\_t value);**

Description

Write Free Running Timer Register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **value** | Free Running Timer Value |

Outputs

None

### p\_icct\_freeRunningTimer\_tClk\_set

Function Header

**void p\_icct\_freeRunningTimer\_tClk\_set(uint8\_t tclk\_frequency)**

Description

Set Free Running Timer Frequency

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **tclk\_frequency** | Clock source for the Free Running Counter |

Use \_ICCT\_FREE\_RUN\_TMR\_TCLK\_ enums from icct.h for tclk\_frequency

Outputs

None

### p\_icct\_freeRunningTimer\_enable

Function Header

**void p\_icct\_freeRunningTimer\_enable(void)**

Description

Enable Free Running Timer

Inputs

None

Outputs

None

### p\_icct\_freeRunningTimer\_disable

Function Header

**void p\_icct\_freeRunningTimer\_disable(void)**

Description

Disable Free Running Timer

Inputs

None

Outputs

None

### icct\_freeRunningTimer\_init

Function Header

**void p\_icct\_freeRunningTimer\_reset(void)**

Description

Resets Free Running Timer

Inputs

None

Outputs

None

### p\_icct\_freeRunningTimer\_reset\_read

Function Header

**uint8\_t p\_icct\_freeRunningTimer\_reset\_read(void)**

Description

Read Free Running Timer Reset Bit

Inputs

None

Outputs

1 or 0

***Functions for ICCT Compare***

### p\_icct\_compare\_reg\_read

Function Header

**uint32\_t p\_icct\_compare\_reg\_read(uint8\_t cmp\_reg\_id)**

Description

Read Compare Register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **cmp\_reg\_id** | ICCT Compare ID |

Outputs

Compare Register Value

### p\_icct\_compare\_reg\_write

Function Header

**void p\_icct\_compare\_reg\_write(uint8\_t cmp\_reg\_id, uint32\_t value)**

Description

Write Compare Register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **cmp\_reg\_id** | ICCT Compare ID |
| **value** | Compare Register Value |

Outputs

None

### p\_icct\_compare\_timer\_enable

Function Header

**void p\_icct\_compare\_timer\_enable(uint8\_t cmp\_reg\_id)**

Description

Enables compare for Compare Register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **cap\_reg\_id** | Compare Register ID |

Outputs

None

### p\_icct\_compare\_timer\_disable

Function Header

**void p\_icct\_compare\_timer\_disable(uint8\_t cmp\_reg\_id)**

Description

Disable compare for Compare Register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **cap\_reg\_id** | Compare Register ID |

Outputs

None

### p\_icct\_compare\_timer\_output\_clear

Function Header

**void p\_icct\_compare\_timer\_output\_clear(uint8\_t cmp\_reg\_id)**

Description

Clears compare timer output state

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **cap\_reg\_id** | Compare Register ID |

Outputs

None

### p\_icct\_compare\_timer\_output\_set

Function Header

**void p\_icct\_compare\_timer\_output\_set(uint8\_t cmp\_reg\_id)**

Description

Set Compare Timer output state to '1'

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **cmp\_reg\_id** | Compare Register ID |

Outputs

None

### p\_icct\_compare\_timer\_output\_read

Function Header

**uint8\_t p\_icct\_compare\_timer\_output\_read(uint8\_t cmp\_reg\_id)**

Description

Get current state of Compare output state

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **cmp\_reg\_id** | Compare Register ID |

Outputs

Compare output state - 1 or 0

***Peripheral Functions for ICCT Capture***

### p\_icct\_capture\_reg\_read

Function Header

**uint32\_t p\_icct\_capture\_reg\_read(uint8\_t cap\_reg\_id)**

Description

Read Capture Register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **cap\_reg\_id** | Capture Register ID |

Outputs

Capture Register Value

### p\_icct\_capture\_control\_reg\_write

Function Header

**void p\_icct\_capture\_control\_reg\_write(uint8\_t cap\_reg\_id, uint8\_t value)**

Description

Write specific Capture Control Register value

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **cap\_reg\_id** | Capture Register ID |
| **value** | Compare Register value |

Outputs

None

### p\_icct\_capture\_control\_reg\_read

Function Header

**uint8\_t p\_icct\_capture\_control\_reg\_read(uint8\_t cap\_reg\_id)**

Description

Read specific capture Control register value

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **cap\_reg\_id** | Capture Register ID |

Outputs

Capture Control Register Value (8-bit)

# PS/2

PS/2

instances

PS2 0A

PS2 0B

PS2 1A

PS2 1B

PS2 2

PS/2 registers

PS2 Transmit Buffer Register

PS2 Receive Buffer Register

PS2 Control Register

PS2 Status Register

PS/2 API’s

ps2\_pins\_init

ps2\_protocol\_init

ps2\_enable\_disable

ps2\_send\_data

ps2\_receive\_data

PS/2 Peripheral Functions

p\_ps2\_control\_reg\_set

p\_ps2\_control\_reg\_get

p\_ps2\_enable\_disable

p\_ps2\_status\_reg\_get

p\_ps2\_tx\_buff\_write

p\_ps2\_rx\_buff\_read

PCR Peripheral Functions

GPIO Peripheral Functions

Interrupt Peripheral Functions

## PS/2 APIs

The list of PS/2 APIs

* ps2\_pins\_init
* ps2\_protocol\_init
* ps2\_enable\_disable
* ps2\_send\_data
* ps2\_receive\_data

### ps2\_pins\_init

Function Header

**void ps2\_pins\_init( uint8\_t ps2\_id )**

Description

Initializes the gpio pins that are associated with the specified PS/2 port for PS/2 functionality.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****ps2\_id**** | 0-based PS/2 ID |

Outputs

None

### ps2\_protocol\_init

Function Header

**void ps2\_protocol\_init( uint8\_t ps2\_id, uint8\_t stp\_bit\_type, uint8\_t parity\_type, uint8\_t dir )**

Description

Initializes all the parameters of the PS/2 communication for the specified PS/2 instance and enables that block.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****ps2\_id**** | 0-based PS/2 ID |
| ****stp\_bit\_type**** | Select the level for the stop bit  PS2\_STOP\_BIT\_AS\_HIGH  PS2\_STOP\_BIT\_AS\_LOW  PS2\_STOP\_BIT\_IGNORED |
| ****parity\_type**** | Select the type of parity check  PS2\_PARITY\_BIT\_AS\_ODD  PS2\_PARITY\_BIT\_AS\_EVEN  PS2\_PARITY\_BIT\_IGNORED |
| ****dir**** | Select the direction for the PS/2 transaction  PS2\_TRANSMIT  PS2\_RECEIVE |

Outputs

None

### ps2\_enable\_disable

Function Header

**void ps2\_enable\_disable( uint8\_t ps2\_id, uint8\_t new\_val )**

Description

Enables/disables the specified PS/2 hardware instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****ps2\_id**** | 0-based PS/2 ID |
| ****new\_val**** | New configuration setting  PS2\_BLOCK\_EN  PS2\_BLOCK\_DIS |

Outputs

None

### ps2\_send\_data

Function Header

**void ps2\_send\_data( uint8\_t ps2\_id, uint8\_t new\_val )**

Description

Transmits serial data via the specified PS/2 instance.

Note - Before sending the data; this API will handle the direction mode.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****ps2\_id**** | 0-based PS/2 ID |
| ****new\_val**** | Data to be sent |

Outputs

None

### ps2\_receive\_data

Function Header

**uint8\_t ps2\_receive\_data( uint8\_t ps2\_id )**

Description

Receives serial data via the specified PS/2 instance.

Note - Before receiving the data; this API will handle the direction mode.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****ps2\_id**** | 0-based PS/2 ID |

Outputs

Data byte received via the PS/2 port

## PS/2 peripheral functions

The list of PS/2 peripheral functions is –

* p\_ps2\_control\_reg\_set
* p\_ps2\_control\_reg\_get
* p\_ps2\_enable\_disable
* p\_ps2\_status\_reg\_get
* p\_ps2\_tx\_buff\_write
* p\_ps2\_rx\_buff\_read

### p\_ps2\_control\_reg\_set

Function Header

**void p\_ps2\_control\_reg\_set( uint8\_t ps2\_id, uint8\_t new\_val )**

Description

Writes to the control register of the specified ps2 hardware instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****ps2\_id**** | 0-based PS/2 ID |
| ****new\_val**** | Data that needs to be written into the register |

Outputs

None

### p\_ps2\_control\_reg\_get

Function Header

**uint8\_t p\_ps2\_control\_reg\_get( uint8\_t ps2\_id )**

Description

Reads the control register of the specified ps2 hardware instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****ps2\_id**** | 0-based PS/2 ID |

Outputs

Contents of the control register

### p\_ps2\_enable\_disable

Function Header

**void p\_ps2\_enable\_disable( uint8\_t ps2\_id, uint8\_t new\_val )**

Description

Writes to the PS2\_EN bit in the control register of specified ps2 hardware instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****ps2\_id**** | 0-based PS/2 ID |
| ****new\_val**** | New configuration setting  PS2\_BLOCK\_EN  PS2\_BLOCK\_DIS |

Outputs

None

### p\_ps2\_status\_reg\_get

Function Header

**uint32\_t p\_ps2\_status\_reg\_get( uint8\_t ps2\_id )**

Description

Reads the status register of the specified ps2 hardware instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****ps2\_id**** | 0-based PS/2 ID |

Outputs

Current contents of the status register

### p\_ps2\_tx\_buff\_write

Function Header

**void p\_ps2\_tx\_buff\_write( uint8\_t ps2\_id, uint8\_t new\_val )**

Description

Writes to the transmit buffer of the specified ps2 hardware instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****ps2\_id**** | 0-based PS/2 ID |
| ****new\_val**** | Data to be sent |

Outputs

None

### p\_ps2\_rx\_buff\_read

Function Header

**uint32\_t p\_ps2\_rx\_buff\_read( uint8\_t ps2\_id )**

Description

Reads the receive buffer of the specified ps2 hardware instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****ps2\_id**** | 0-based PS/2 ID |

Outputs

Data byte received via the PS/2 port

# Keyboard Scan Matrix

Scan Matrix

instances

KSM 0

Scan Matrix registers

KSO Select Register

KSI Input register

KSI Input Status Register

KSI Interrupt Enable Register

Keyscan Extended Control Register

Scan Matrix API’s

ksm\_block\_init

ksm\_predrive\_configure

Scan Matrix Peripheral Functions

p\_ksm\_kso\_select\_reg\_set

p\_ksm\_kso\_select\_reg\_get

p\_ksm\_block\_enable\_disable

p\_ksm\_kso\_line\_select

p\_ksm\_scan\_input\_reg\_get

p\_ksm\_scan\_input\_status\_get\_clr

p\_ksm\_scan\_interrupt\_enable\_disable

p\_ksm\_extended\_cntrl\_reg\_set

PCR Peripheral Functions

GPIO Peripheral Functions

Interrupt Peripheral Functions

## Keyboard Scan Matrix APIs

The list of Keyboard Scan Matrix APIs are

* ksm\_block\_init
* ksm\_predrive\_configure

### ksm\_block\_init

Function Header

**void ksm\_block\_init( uint8\_t out\_lvl, uint8\_t out\_cntrl, uint8\_t operating\_mode, uint8\_t int\_val )**

Description

Initializes the scan matrix block and its associated gpio pins followed by a block enable.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****out\_lvl**** | Output level of row lines when selected  KSO\_LINES\_NON\_INVERTED – driven low  KSO\_LINES\_INVERTED – driven high |
| out\_cntrl | Output control for the row lines  KSO\_OUPTPUT\_DRIVEN\_HIGH  KSO\_OUTPUT\_DRIVEN\_BY\_KSO\_SEL |
| ****operating\_mode**** | Operating mode for the scan matrix block  KSO\_PREDIRVE\_ENABLE  KSO\_PREDIRVE\_DISABLE |
| ****int\_val**** | 8 bit value where each bit represents a KSI line  0 – interrupt disabled  1 – interrupt enabled |

Outputs

None

### ksm\_predrive\_configure

Function Header

**void ksm\_predrive\_configure( uint8\_t predrive\_select )**

Description

Configures the scan matrix block for predrive/non-predrive mode.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****predrive\_select**** | Operating mode for the scan matrix block  KSO\_PREDIRVE\_ENABLE  KSO\_PREDIRVE\_DISABLE |

Outputs

None

## Keyboard Scan Matrix peripheral functions

The list of Keyboard Scan Matrix peripheral functions is –

* p\_ksm\_kso\_select\_reg\_set
* p\_ksm\_kso\_select\_reg\_get
* p\_ksm\_block\_enable\_disable
* p\_ksm\_kso\_line\_select
* p\_ksm\_scan\_input\_reg\_get
* p\_ksm\_scan\_input\_status\_get\_clr
* p\_ksm\_scan\_interrupt\_enable\_disable
* p\_ksm\_extended\_cntrl\_reg\_set

### p\_ksm\_kso\_select\_reg\_set

Function Header

**void p\_ksm\_kso\_select\_reg\_set( uint8\_t new\_val )**

Description

Writes to the KSO select register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | Data that needs to be written into the register |

Outputs

None

### p\_ksm\_kso\_select\_reg\_get

Function Header

**uint8\_t p\_ksm\_kso\_select\_reg\_get( void )**

Description

Reads the contents of the KSO select register.

Outputs

Contents of the KSO select register.

### p\_ksm\_block\_enable\_disable

Function Header

**void p\_ksm\_block\_enable\_disable( uint8\_t new\_val )**

Description

Writes to the KSEN bit of KSO select register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New Setting  KSM\_ENABLE  KSM\_DISABLE |

Outputs

None

### p\_ksm\_kso\_line\_select

Function Header

**void p\_ksm\_kso\_line\_select( uint8\_t line\_num )**

Description

Selects the output line that needs to be scanned.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****line\_num**** | New Setting  KSO\_LINE\_00 … KSO\_LINE\_17 |

Outputs

None

### p\_ksm\_scan\_input\_reg\_set

Function Header

**uint8\_t p\_ksm\_scan\_input\_reg\_get( void )**

Description

Reads the KSI input register.

Outputs

Contents of the KSI input register.

### p\_ksm\_scan\_input\_status\_get\_clr

Function Header

**uint8\_t p\_ksm\_scan\_input\_status\_get\_clr( void )**

Description

Reads the KSI status register and then clears it.

Outputs

Contents of the KSI status register

### p\_ksm\_interrupt\_enable\_disable

Function Header

**void p\_ksm\_scan\_interrupt\_enable\_disable( uint8\_t new\_val )**

Description

Writes to the KSI interrupt enable register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | 8 bit value where each bit represents a KSI line  0 – interrupt disabled  1 – interrupt enabled |

Outputs

None

### p\_ksm\_extended\_cntrl\_reg\_set

Function Header

**void p\_ksm\_extended\_cntrl\_reg\_set( uint8\_t new\_val )**

Description

Writes to the keyscan extended control register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New configuration setting  KSO\_PREDIRVE\_ENABLE  KSO\_PREDIRVE\_DISABLE |

Outputs

None

# Week Timer

## Week Timer APIs

The list of Week Timer APIs

* wtimer\_week\_alarm\_init
* wtimer\_subSecond\_init
* wtimer\_subWeek\_alarm\_init

### wtimer\_week\_alarm\_init

Function Header

void wtimer\_week\_alarm\_init(uint32\_t week\_counter, uint32\_t week\_compare)

Description

Initialize Week Alarm

Note:

1. To start counting after initializing, set the WT\_ENABLE bit in control register - use p\_wtimer\_ctrl\_enable API
2. The Week alarm counter must not be modified if Sub-Week Alarm Counter is using the Week Alarm Counter as its clock source

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **week\_counter** | week counter value |
| **week\_compare** | week compare value |

Outputs

None

### wtimer\_subSecond\_init

Function Header

void wtimer\_subSecond\_init(uint8\_t spisr\_value)

Description

Initialize Sub Second signals

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **spisr\_value** | sub seconds events rate |

Use enum \_WTIMER\_SPISR\_ENCODING\_ values for spisr value

Outputs

None

### wtimer\_subWeek\_alarm\_init

Function Header

void wtimer\_subWeek\_alarm\_init( uint16\_t subWeek\_tick, uint8\_t auto\_reload, uint16\_t subWeek\_counter)

Description

Initialize Sub-Week Alarm

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **subWeek\_tick** | Sub Week Tick Value - enum \_WTIMER\_SUBWEEK\_TICK\_ |
| **auto\_reload** | auto\_reload set or clr; enum \_WTIMER\_SUBWEEK\_AUTORELOAD\_ |
| **subWeek\_counter** | sub-week counter |

Outputs

None

## Week Timer Peripheral Functions

The list of Week Timer Peripheral functions

* p\_wtimer\_ctrl\_enable
* p\_wtimer\_ctrl\_disable
* p\_wtimer\_ctrl\_powerUpEventOutput\_enable
* p\_wtimer\_ctrl\_powerUpEventOutput\_disable
* p\_wtimer\_weekAlarmCounter\_reg\_write
* p\_wtimer\_weekAlarmCounter\_reg\_read
* p\_wtimer\_weekCompare\_reg\_write
* p\_wtimer\_weekCompare\_reg\_read
* p\_wtimer\_clkDivider\_reg\_read
* p\_wtimer\_SPISR\_reg\_write
* p\_wtimer\_SPISR\_reg\_read
* p\_wtimer\_subWeekControl\_reg\_write
* p\_wtimer\_subWeekControl\_reg\_read
* p\_wtimer\_WEEK\_TIMER\_POWERUP\_EVENT\_STATUS\_get
* p\_wtimer\_WEEK\_TIMER\_POWERUP\_EVENT\_STATUS\_clr
* p\_wtimer\_SUBWEEK\_TIMER\_POWERUP\_EVENT\_STATUS\_get
* p\_wtimer\_SUBWEEK\_TIMER\_POWERUP\_EVENT\_STATUS\_clr
* p\_wtimer\_subWeekCounter\_load
* p\_wtimer\_subWeekCounter\_sts\_read
* p\_bgpo\_data\_reg\_write
* p\_bgpo\_data\_reg\_read
* p\_bgpo\_power\_reg\_write
* p\_bgpo\_power\_reg\_read
* p\_bgpo\_reset\_reg\_write
* p\_bgpo\_reset\_reg\_read

### p\_wtimer\_ctrl\_enable

Function Header

void p\_wtimer\_ctrl\_enable(void)

Description

Enable Week Timer

Inputs

None

Outputs

None

### p\_wtimer\_ctrl\_disable

Function Header

void p\_wtimer\_ctrl\_disable(void)

Description

Disable Week Timer

Inputs

None

Outputs

None

### p\_wtimer\_ctrl\_powerUpEventOutput\_enable

Function Header

void p\_wtimer\_ctrl\_powerUpEventOutput\_enable(void)

Description

Enable Power-Up Event Output

Inputs

None

Outputs

None

### p\_wtimer\_ctrl\_powerUpEventOutput\_disable

Function Header

void p\_wtimer\_ctrl\_powerUpEventOutput\_disable(void)

Description

Disable Power-Up Event Output

Inputs

None

Outputs

None

### p\_wtimer\_weekAlarmCounter\_reg\_write

Function Header

void p\_wtimer\_weekAlarmCounter\_reg\_write(uint32\_t value)

Description

Write Week Alarm Counter Register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **value** | counter value |

Outputs

None

### p\_wtimer\_weekAlarmCounter\_reg\_read

Function Header

uint32\_t p\_wtimer\_weekAlarmCounter\_reg\_read(void)

Description

Read Week Alarm Counter Register

Inputs

None

Outputs

value - counter value

### p\_wtimer\_weekCompare\_reg\_write

Function Header

void p\_wtimer\_weekCompare\_reg\_write(uint32\_t value)

Description

Write Week Compare Register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **value** | counter value |

Outputs

None

### p\_wtimer\_weekCompare\_reg\_read

Function Header

uint32\_t p\_wtimer\_weekCompare\_reg\_read(void)

Description

Read Week Compare Register

Inputs

None

Outputs

counter value

### p\_wtimer\_clkDivider\_reg\_read

Function Header

uint32\_t p\_wtimer\_clkDivider\_reg\_read(void)

Description

Read Clock Divider Register

Inputs

None

Outputs

Clock Divider value

### p\_wtimer\_SPISR\_reg\_write

Function Header

void p\_wtimer\_SPISR\_reg\_write(uint8\_t spisr\_value)

Description

Write SPISR - Sub-second programmable interrupt select register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **spisr\_value** | SPISR value |

Outputs

None

### p\_wtimer\_SPISR\_reg\_read

Function Header

uint8\_t p\_wtimer\_SPISR\_reg\_read(void)

Description

Read SPISR - Sub-second programmable interrupt select register

Inputs

None

Outputs

SPISR value

### p \_wtimer\_subWeekControl\_reg\_write

Function Header

void p\_wtimer\_subWeekControl\_reg\_write(uint16\_t value)

Description

Write Sub-Week Control Register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **value** | Sub Week Control Value |

Outputs

None

### p\_wtimer\_subWeekControl\_reg\_read

Function Header

uint16\_t p\_wtimer\_subWeekControl\_reg\_read(void)

Description

Read Sub-Week Control Register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **value** | Sub Week Control Register Value |

Outputs

None

### p\_wtimer\_WEEK\_TIMER\_POWERUP\_EVENT\_STATUS\_get

Function Header

uint8\_t p\_wtimer\_WEEK\_TIMER\_POWERUP\_EVENT\_STATUS\_get(void)

Description

Read WEEK\_TIMER\_POWERUP\_EVENT\_STATUS bit

Inputs

None

Outputs

1 if set, else 0

### p\_wtimer\_WEEK\_TIMER\_POWERUP\_EVENT\_STATUS\_clr

Function Header

void p\_wtimer\_WEEK\_TIMER\_POWERUP\_EVENT\_STATUS\_clr(void)

Description

Clear WEEK\_TIMER\_POWERUP\_EVENT\_STATUS bit

Inputs

None

Outputs

None

### p\_wtimer\_SUBWEEK\_TIMER\_POWERUP\_EVENT\_STATUS\_get

Function Header

uint8\_t p\_wtimer\_SUBWEEK\_TIMER\_POWERUP\_EVENT\_STATUS\_get(void)

Description

Read SUBWEEK\_TIMER\_POWERUP\_EVENT\_STATUS

Inputs

None

Outputs

1 if set, else 0

### p\_wtimer\_SUBWEEK\_TIMER\_POWERUP\_EVENT\_STATUS\_clr

Function Header

void p\_wtimer\_SUBWEEK\_TIMER\_POWERUP\_EVENT\_STATUS\_clr(void)

Description

Clear SUBWEEK\_TIMER\_POWERUP\_EVENT\_STATUS bit

Inputs

None

Outputs

None

### p\_wtimer\_subWeekCounter\_load

Function Header

void p\_wtimer\_subWeekCounter\_load(uint16\_t value)

Description

Write SUBWEEK\_COUNTER\_LOAD value in sub-week alarm counter register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **value** | subweek counter value |

Outputs

None

### p\_wtimer\_subWeekCounter\_sts\_read

Function Header

uint16\_t p\_wtimer\_subWeekCounter\_sts\_read(void)

Description

Read SUBWEEK\_COUNTER\_STATUS from sub-week alarm counter Register

Inputs

None

Outputs

Sub Week alarm counter Status Value

### p\_bgpo\_data\_reg\_write

Function Header

void p\_bgpo\_data\_reg\_write(uint16\_t value)

Description

Write BGPO Data Register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **value** | Value to be written |

Outputs

None

### p\_bgpo\_data\_reg\_read

Function Header

uint16\_t p\_bgpo\_data\_reg\_read(void)

Description

Read BGPO Data Register

Inputs

None

Outputs

BGPO Data bitmask value

### p\_bgpo\_power\_reg\_write

Function Header

void p\_bgpo\_power\_reg\_write(uint8\_t value)

Description

Write BGPO Power Register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **value** | Value to be written |

Outputs

None

### p\_bgpo\_power\_reg\_read

Function Header

uint8\_t p\_bgpo\_power\_reg\_read(void)

Description

Read BGPO Power Register

Inputs

None

Outputs

BGPO Power bitmask value

### p\_bgpo\_reset\_reg\_write

Function Header

void p\_bgpo\_reset\_reg\_write(uint16\_t value)

Description

Write BGPO Reset Register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **value** | Value to be written |

Outputs

None

### p\_bgpo\_reset\_reg\_read

Function Header

uint16\_t p\_bgpo\_reset\_reg\_read(void)

Description

Read BGPO Reset Register

Inputs

None

Outputs

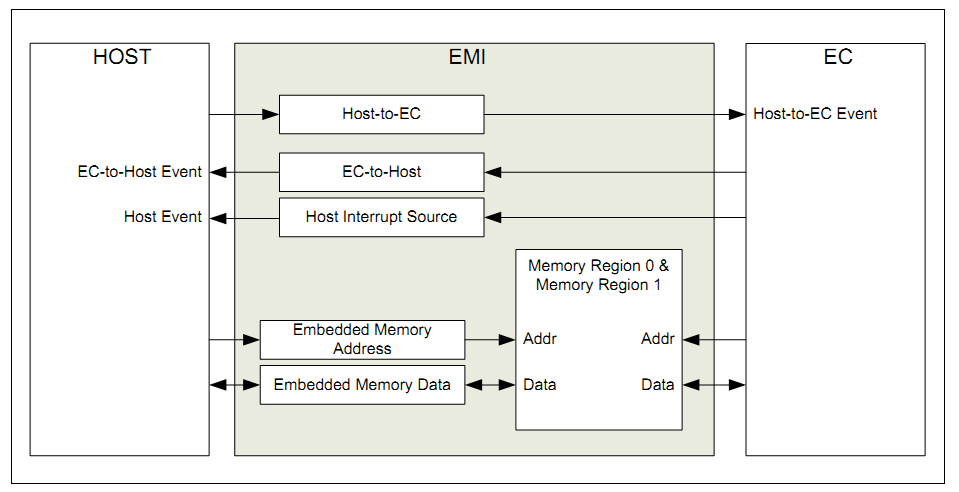
BGPO Power source bitmask

# EMI

The Embedded Memory Interface (EMI) provides a standard run-time mechanism for the system host to communicate with the Embedded Controller (EC) and other logical components. The Embedded Memory Interface can be used by the Host to access bytes of memory designated by the EC without requiring any assistance from the EC. The EC may configure these regions of memory as read-only, write-only, or read/write capable.

Important features

* Mailbox support for communication between host and EC
* Maximum two memory regions from EC internal SRAM address can be mapped in host address space
* Set separate read, write limits for regions
* Maximum length of a region can be 32K bytes
* Access type: 1, 2 or 4 byte
* Application ID support for controlling the ownership of EMI in multitasking applications
* Higher level protocol possible through use of mailbox



EMI instances

EMI 0

EMI 1

EMI 2

EMI registers

Host to EC Mailbox register

EC to Host Mailbox register

Memory Base Address 0 register

Memory Read Limit 0 register

Memory Write Limit 0 register

Memory Base Address 1 register

Memory Read Limit 1 register

Memory Write Limit 1 register

Interrupt Set register

Host Clear Enable register

EMI block Api’s

emi\_block\_init

emi\_block\_enable\_interrupt

emi\_block\_get\_mem\_base\_addr

emi\_block\_get\_read\_limit

emi\_block\_get\_write\_limit

emi\_block\_get\_app\_id

EMI block Peripheral Functions

p\_emi\_block\_set\_base\_address

p\_emi\_block\_set\_read\_limit

p\_emi\_block\_set\_write\_limit

p\_emi\_block\_set\_swi\_intr

p\_emi\_block\_write\_mbox

p\_emi\_block\_read\_mbox

p\_emi\_block\_read\_app\_id

## EMI block APIs

emi\_block\_init

emi\_block\_set\_access\_type

emi\_block\_enable\_interrupt

emi\_block\_get\_mem\_base\_addr

emi\_block\_get\_read\_limit

emi\_block\_get\_write\_limit

emi\_block\_set\_app\_id

emi\_block\_get\_app\_id

### emi\_block\_init

Function Header

**void** emi\_block\_init **(uint8\_t emi\_id, uint8\_t mem\_region, uint32\_t base\_addr, uint16\_t rlimit, uint16\_t wlimit)**

Description

The function initializes the emi block with following parameters

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| emi\_id | EMI instance  EMI\_0  EMI\_1  EMI\_2 |
| mem\_region | Memory region 0 or 1  MEM\_REGION\_0  MEM\_REGION\_1 |
| base\_addr | Base address value for selected memory region |
| rlimit | Read limit for the selected memory region |
| wlimit | Write limit for the selected memory region |

Outputs

None

### emi\_block\_enable\_interrupt

Function Header

**void** emi\_block\_enable\_swi\_set **(uint8\_t emi\_id)**

Description

The function enables the EC\_SWI interrupt

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| emi\_id | EMI instance  EMI\_0  EMI\_1  EMI\_2 |

Outputs

None

### emi\_block\_get\_mem\_base\_addr

Function Header

**uint32\_t** emi\_block\_get\_mem\_base\_addr (uint8\_t emi\_id)

Description

The function returns the memory base address register value

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| emi\_id | EMI instance  EMI\_0  EMI\_1  EMI\_2 |

Outputs

Memory base address value

### emi\_block\_get\_read\_limit

Function Header

**uint16\_t** emi\_block\_get\_read\_limit (uint8\_t emi\_id, uint8\_t mem\_region)

Description

The function returns the read limit for the selected memory region

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| emi\_id | EMI instance  EMI\_0 |
| mem\_region | Memory region 0 or 1  MEM\_REGION\_0  MEM\_REGION\_1 |

Outputs

Memory read limit value

### emi\_block\_get\_write\_limit

Function Header

**uint16\_t** emi\_block\_get\_write\_limit (uint8\_t emi\_id, uint8\_t mem\_region)

Description

The function returns the write limit for the selected memory region

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| emi\_id | EMI instance  EMI\_0  EMI\_1  EMI\_2 |
| mem\_region | Memory region 0 or 1  MEM\_REGION\_0  MEM\_REGION\_1 |

Outputs

Memory write limit value

### emi\_block\_get\_app\_id

Function Header

**uint8\_t** emi\_block\_set\_app\_id (uint8\_t emi\_id)

Description

The function returns the application id register value

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| emi\_id | EMI instance  EMI\_0  EMI\_1  EMI\_2 |

Outputs

Application id value

## EMI block peripheral functions

p\_emi\_block\_set\_base\_address

p\_emi\_block\_set\_read\_milit

p\_emi\_block\_set\_write\_limit

p\_emi\_block\_set\_swi\_intr

p\_emi\_block\_write\_mbox

p\_emi\_block\_read\_mbox

p\_emi\_block\_read\_app\_id

### p\_emi\_block\_set\_base\_addr

Function Header

**void\_t p\_**emi\_block\_set\_base\_addr (uint8\_t emi\_id, uint8\_t mem\_region, uint32\_t addr)

Description

The function sets the memory base address for selected memory region

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| emi\_id | EMI instance  EMI\_0  EMI\_1  EMI\_2 |
| mem\_region | Memory region 0 or 1  MEMORY\_REGION\_0  MEMORY\_REGION\_1 |

Outputs

None

### p\_emi\_block\_set\_read\_limit

Function Header

**uint16\_t p\_**emi\_block\_set\_read\_limit (uint8\_t emi\_id, uint8\_t mem\_region, uint16\_t rlimit)

Description

The function returns the read limit for the selected memory region

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| emi\_id | EMI instance  EMI\_0 |
| mem\_region | Memory region 0 or 1  MEM\_REGION\_0  MEM\_REGION\_1 |
| rlimit | Read limit value which is to be added to base address to get the final read limit address |

Outputs

Memory read limit value

### p\_emi\_block\_set\_write\_limit

Function Header

**void p\_**emi\_block\_set\_write\_limit (uint8\_t emi\_id, uint8\_t mem\_region, uint16\_t wlimit)

Description

The function sets the write limit for the selected memory region

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| emi\_id | EMI instance  EMI\_0  EMI\_1  EMI\_2 |
| mem\_region | Memory region 0 or 1  MEM\_REGION\_0  MEM\_REGION\_1 |
| wlimit | Write limit offset value to be added to base address to get the final write limit address |

Outputs

None

### p\_emi\_block\_write\_mbox

Function Header

**void p\_**emi\_block\_write\_mbox **(uint8\_t emi\_id, uint8\_t mbox\_id, uint8\_t data)**

Description

The function writes the 1 byte data to the mailbox selected by mbox\_id

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| emi\_id | EMI instance  EMI\_0  EMI\_1  EMI\_2 |
| mbox\_id | MBOX\_HOST\_TO\_EC  MBOX\_EC\_TO\_HOST |
| Data | Data value to be written |

Outputs

None

### p\_emi\_block\_read\_mbox

Function Header

**uint8\_t p\_**emi\_block\_read\_mbox **(uint8\_t emi\_id, uint8\_t mbox\_id)**

Description

The function reads the 1 byte data from the mailbox selected by mbox\_id

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| emi\_id | EMI instance  EMI\_0  EMI\_1  EMI\_2 |
| mbox\_id | MBOX\_HOST\_TO\_EC  MBOX\_EC\_TO\_HOST |

Outputs

Returns the value from the mailbox

### p\_emi\_block\_enable\_swi\_set

Function Header

**void p\_**emi\_block\_enable\_swi\_set **(uint8\_t emi\_id)**

Description

The function enables the EC\_SWI interrupt

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| emi\_id | EMI instance  EMI\_0  EMI\_1  EMI\_2 |

Outputs

None

### p\_emi\_block\_get\_mem\_base\_addr

Function Header

**uint32\_t p\_**emi\_block\_get\_mem\_base\_addr (uint8\_t emi\_id)

Description

The function returns the memory base address register value

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| emi\_id | EMI instance  EMI\_0  EMI\_1  EMI\_2 |

Outputs

Memory base address value

### p\_emi\_block\_get\_read\_limit

Function Header

**uint16\_t p\_**emi\_block\_get\_read\_limit (uint8\_t emi\_id, uint8\_t mem\_region)

Description

The function returns the read limit for the selected memory region

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| emi\_id | EMI instance  EMI\_0  EMI\_1  EMI\_2 |
| mem\_region | Memory region 0 or 1  MEM\_REGION\_0  MEM\_REGION\_1 |

Outputs

Memory read limit value

### p\_emi\_block\_get\_write\_limit

Function Header

**uint16\_t p\_**emi\_block\_get\_write\_limit (uint8\_t emi\_id, uint8\_t mem\_region)

Description

The function returns the write limit for the selected memory region

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| emi\_id | EMI instance  EMI\_0  EMI\_1  EMI\_2 |
| mem\_region | Memory region 0 or 1  MEM\_REGION\_0  MEM\_REGION\_1 |

Outputs

Memory write limit value

### p\_emi\_block\_get\_app\_id

Function Header

**uint8\_t p\_**emi\_block\_set\_app\_id (uint8\_t emi\_id)

Description

The function returns the application id register value

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| emi\_id | EMI instance  EMI\_0  EMI\_1  EMI\_2 |

Outputs

Application id value

# MailBox

MailBox instances

MAILBOX

MailBox registers

HOST-to-EC Mailbox Register

EC-to-Host Mailbox Register

Mailbox registers [0-1F]

MailBox Peripheral Functions

p\_mbx\_write\_reg

p\_mbx\_read\_reg

p\_mbx\_EC\_to\_HOST\_read

p\_mbx\_EC\_to\_HOST\_write

p\_mbx\_HOST\_to\_EC\_read

p\_mbx\_HOST\_to\_EC\_write

MailBox API’s

mbx\_write

mbx\_read

mbx\_read\_reg

mbx\_write\_reg

mbx\_HOST\_to\_EC\_write

mbx\_HOST\_to\_EC\_read

mbx\_EC\_to\_HOST\_write

mbx\_EC\_to\_HOST\_read

## Mailbox APIs

The list of Mailbox APIs

### mbx\_write

### mbx\_read

### mbx\_read\_reg

### mbx\_write\_reg

### mbx\_HOST\_to\_EC\_write

### mbx\_HOST\_to\_EC\_read

### mbx\_EC\_to\_HOST\_write

### mbx\_EC\_to\_HOST\_readmbx\_write

Function Header

**bool mbx\_write( uint8\_t length, uint8\_t\* data\_buffer)**

Description

Write buffer data into Mailbox registers

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****length**** | Number of bytes to be written |
| ****data\_buffer**** | Data buffer to be written into mailbox registers |

Outputs

Status of callback

### mbx\_read

Function Header

**bool mbx\_read( uint8\_t length, uint8\_t\* data\_buffer)**

Description

read buffer data from Mailbox registers

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****length**** | Number of bytes to be read |
| ****data\_buffer**** | Data buffer to store mailbox register data. |

Outputs

Status of callback

### mbx\_write\_reg

Function Header

**uint8\_t mbx\_write\_reg( uint8\_t offset, uint8\_t length, uint8\_t\* data\_buffer)**

Description

Write buffer data into Mailbox registers

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****offset**** | Register offset |
| ****length**** | Number of bytes to be written |
| ****data\_buffer**** | Data buffer to be written into mailbox registers |

Outputs

Number of bytes written

### mbx\_read\_reg

Function Header

**uint8\_t mbx\_read\_reg( uint8\_t offset, uint8\_t length, uint8\_t\* data\_buffer)**

Description

Read data into buffer from Mailbox registers

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****offset**** | Register offset |
| ****length**** | Number of bytes to be read |
| ****data\_buffer**** | Data buffer to store mailbox register data |

Outputs

Number of bytes read

### mbx\_HOST\_to\_EC\_write

Function Header

**void mbx\_HOST\_to\_EC\_write ( uint8\_t data)**

Description

Write data into **HOST\_to\_EC** Mailbox register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****data**** | Data Byte to be written |

Outputs

None

### mbx\_HOST\_to\_EC\_read

Function Header

**Uint8\_t mbx\_HOST\_to\_EC\_read ( void)**

Description

Read **HOST\_to\_EC** Mailbox register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****None**** |  |

Outputs

Data in **HOST\_to\_EC** Mailbox register

### mbx\_EC\_to\_HOST\_write

Function Header

**void mbx\_EC\_to\_HOST \_write ( uint8\_t data)**

Description

Write data into **EC\_to\_HOST**  Mailbox register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****data**** | Data Byte to be written |

Outputs

None

### mbx\_EC\_to\_HOST\_read

Function Header

**Uint8\_t mbx\_EC\_to\_HOST \_read ( void)**

Description

Read **EC\_to\_HOST Mailbox** register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****None**** |  |

Outputs

Data in **EC\_to\_HOST** Mailbox register

## Mailbox block peripheral functions

p\_mbx\_write\_reg

p\_mbx\_read\_reg

p\_mbx\_EC\_to\_HOST\_read

p\_mbx\_EC\_to\_HOST\_write

p\_mbx\_HOST\_to\_EC\_read

p\_mbx\_HOST\_to\_EC\_write

### p\_mbx\_read\_reg

Function Header

**uint8\_t p\_mbx\_read\_reg(uint8\_t offset,uint8\_t length, uint8\_t\* data\_buffer)**

Description

read mailbox register into buffer.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****Offset**** | offset of register to be read |
| ****Length**** | length of data to be read |
| ****Data\_buffer**** | output buffer to store data in mailbox |

Outputs

Number of bytes read.

### p\_mbx\_write\_reg

Function Header

**uint8\_t p\_mbx\_write\_reg(uint8\_t offset,uint8\_t length, uint8\_t\* data\_buffer)**

Description

write buffer data into mailbox register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****Offset**** | offset of register to write |
| ****Length**** | length of data to be written |
| ****Data\_buffer**** | input buffer data that needs to be written in mailbox registers |

Outputs

Number of bytes written.

### P\_mbx\_HOST\_to\_EC\_write

Function Header

**void p\_mbx\_HOST\_to\_EC\_write ( uint8\_t data)**

Description

Write data into **HOST\_to\_EC** Mailbox register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****data**** | Data Byte to be written |

Outputs

None

### P\_mbx\_HOST\_to\_EC\_read

Function Header

**Uint8\_t p\_mbx\_HOST\_to\_EC\_read ( void)**

Description

Read **HOST\_to\_EC** Mailbox register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****None**** |  |

Outputs

Data in **HOST\_to\_EC** Mailbox register

### P\_mbx\_EC\_to\_HOST\_write

Function Header

**void p\_mbx\_EC\_to\_HOST \_write ( uint8\_t data)**

Description

Write data into **EC\_to\_HOST**  Mailbox register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****data**** | Data Byte to be written |

Outputs

None

### p\_mbx\_EC\_to\_HOST\_read

Function Header

**Uint8\_t p\_mbx\_EC\_to\_HOST \_read ( void)**

Description

Read **EC\_to\_HOST Mailbox** register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****None**** |  |

Outputs

Data in **EC\_to\_HOST** Mailbox register

# BClink

BCLink registers

BC-Link Status

BC-Link Address

BC-Link Data

BC-Link Clock Select

BCLink Peripheral Functions

p\_BCLINK\_read\_data

p\_BCLINK\_write\_addr

p\_BCLINK\_write\_data

p\_BCLINK\_rst\_set

p\_BCLINK\_rst\_clr

p\_BCLINK\_err\_get

p\_BCLINK\_err\_clr

p\_BCLINK\_chk\_busy

p\_BCLINK\_clk\_freq\_set

BClink Api’s

bclink\_init

bclink\_set\_clock\_frequency

bclink\_write\_byte

bclink\_read\_byte

bclink\_check\_busy

BClink instances

BClink 0

BClink 1

## BClink APIs

The list of BClink APIs

* bclink\_init
* bclink\_set\_clock\_frequency
* bclink\_write\_byte
* bclink\_read\_byte
* bclink\_check\_busy

### bclink\_init

Function Header

**void bclink\_init(uint8\_t clk\_freq, uint8\_t int\_en, uint8\_t blk\_inst)**

Description

This function is used to initialize the BClink with the proper GPIO pin configuration and Clock frequency to operate and option for enabling interrupts

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **clk\_freq** | Clock frequency input for the slave device |
| **int\_en** | Interrupt 1 = Enable or 0 = disabled value |
| **blk\_inst** | Block instance ID for the BClink Used |

Outputs

None

### bclink\_set\_clock\_frequency

Function Header

**uint8\_t bclink\_set\_clock\_frequency(uint8\_t clk\_sel, uint8\_t blk\_inst)**

Description

This function set the given clock frequency to the respective clock register for the salve device to operate

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **clk\_freq** | Clock frequency input for the slave device |
| **blk\_inst** | Block instance ID for the BClink Used |

Outputs

Success = 1 or failure = 0

### bclink\_write\_byte

Function Header

**uint8\_t bclink\_write\_byte(uint8\_t addr, uint8\_t data, uint8\_t blk\_inst)**

Description

This function programs data to the corresponding address of the slave device

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **addr** | the address byte sent to bclink device |
| data | the data byte written to bclink davice |
| **blk\_inst** | Block instance ID for the BClink Used |

Outputs

Success = 1 or failure = 0

### bclink\_read\_byte

Function Header

**uint8\_t bclink\_read\_byte(uint8\_t addr, uint8\_t blk\_inst)**

Description

This function will read corresponding data from the address provided

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **addr** | the address byte sent to bclink device |
| **blk\_inst** | Block instance ID for the BClink Used |

Outputs

Data read from the slave device

### bclink\_check\_busy

Function Header

**uint8\_t bclink\_check\_busy(uint8\_t blk\_inst)**

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **blk\_inst** | Block instance ID for the BClink Used |

Description

This function Check the busy astatus bit of the slave device connected

Inputs

None

Outputs

Busy status returned from the device 1 = Not Busy 0 = Busy

## BClink Peripheral functions

The list of Hibernation Timer Peripheral functions

* p\_BCLINK\_write\_addr
* p\_BCLINK\_read\_data
* p\_BCLINK\_write\_data
* p\_BCLINK\_rst\_set
* p\_BCLINK\_rst\_clr
* p\_BCLINK\_err\_get
* p\_BCLINK\_err\_clr
* p\_BCLINK\_chk\_busy
* p\_BCLINK\_clk\_freq\_set

### p\_BCLINK\_write\_addr

Function Header

**void p\_BCLINK\_write\_addr(uint8\_t addr, uint8\_t blk\_inst)**

Description

This function is used write the Write address of the slave bclink device targeted in the address register

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **addr** | address of the slave device |

Outputs

None

### p\_BCLINK\_read\_data

Function Header

**uint8\_t p\_BCLINK\_read\_data(uint8\_t blk\_inst)**

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **blk\_inst** | Block instance ID for the BClink Used |

Description

This function is used to Read data from the BClink data register

Inputs

None

Outputs

Data read from the data register.

### p\_BCLINK\_write\_data

Function Header

**void p\_BCLINK\_write\_data(uint8\_t data, uint8\_t blk\_inst)**

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **blk\_inst** | Block instance ID for the BClink Used |

Description

This function returns the Hibernation Timer current count value

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **data** | data to be written to the data register |

Outputs

None

### p\_BCLINK\_rst\_set

Function Header

**void p\_BCLINK\_rst\_set(uint8\_t blk\_inst)**

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **blk\_inst** | Block instance ID for the BClink Used |

Description

This function is used to set the reset bit of the BClink to keep the block in reset

Inputs

None

Outputs

None

### p\_BCLINK\_rst\_clr

Function Header

**void p\_BCLINK\_rst\_clr(uint8\_t blk\_inst)**

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **blk\_inst** | Block instance ID for the BClink Used |

Description

p\_BCLINK\_rst\_clr is used to Clear Reset Bit.

Inputs

None

Outputs

None

### p\_BCLINK\_err\_get

Function Header

**uint8\_t p\_BCLINK\_err\_get(uint8\_t blk\_inst)**

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **blk\_inst** | Block instance ID for the BClink Used |

Description

p\_BCLINK\_err\_get is used to return Err status form the status register.

Inputs

None

Outputs

Return 1 if the error bit is set or 0 if no error

### p\_BCLINK\_err\_clr

Function Header

**void p\_BCLINK\_err\_clr(uint8\_t blk\_inst)**

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **blk\_inst** | Block instance ID for the BClink Used |

Description

p\_BCLINK\_err\_clr is used to Clear the error Bit if set in the status register

Inputs

None

Outputs

None

### p\_BCLINK\_chk\_busy

Function Header

**uint8\_t p\_BCLINK\_chk\_busy(uint8\_t blk\_inst)**

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **blk\_inst** | Block instance ID for the BClink Used |

Description

p\_BCLINK\_chk\_busy is used to Check busy status from the status register

Inputs

None

Outputs

Busy status bit = 1 if Busy or 0 if not busy

### p\_BCLINK\_clk\_freq\_set

Function Header

**void p\_BCLINK\_clk\_freq\_set(uint8\_t clk\_sel, uint8\_t blk\_inst)**

Description

p\_BCLINK\_clk\_select is used to Set Clock frequency to operate and set in the clock register associated with the bclink

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| **clk\_sel** | Clock frequency to set |
| **blk\_inst** | Block instance ID for the BClink Used |

Outputs

None

# VBAT Register Bank

VBAT

instances

VBAT 0

VBAT registers

Power Fail and Reset Status Register

Clock Enable Register

Monotonic Counter register

Counter HiWiord Register

VWire Backup Register

VBAT API’s

vbat\_32k\_clk\_configure

VBAT Peripheral Functions

p\_vbat\_32k\_clk\_src\_select

p\_vbat\_32k\_suppress\_configure

p\_vbat\_pfr\_sts\_reg\_get

p\_vbat\_pfr\_sts\_bit\_clr

p\_vbat\_monotonic\_counter\_read

p\_vbat\_counter\_hiword\_write

p\_vbat\_counter\_hiword\_read

p\_vbat\_vwire\_reg\_write

p\_vbat\_vwire\_reg\_read

PCR Peripheral Functions

Interrupt Peripheral Functions

## VBAT Register Bank API Function

The list of VBAT Register Bank API functions are

* vbat\_32k\_clk\_configure

### vbat\_32k\_clk\_configure

Function Header

**void vbat\_32k\_clk\_configure( enum CLK\_CONFIG new\_val )**

Description

Selects the base clock source for 32 KHz domain and executes the necessary steps for it to get stable.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | External clock source selection  PIN\_32KHz\_IN – single ended signal through 32KHz\_IN  INT\_SILICON\_OSC – internal RC oscillator  EXT\_RESONATOR – external crystal  EXT\_SINGLE\_ENDED\_SIG – single ended signal through  XTAL2 |

Outputs

None

## VBAT Register Bank Peripheral Function

The list of VBAT Register Bank peripheral functions are

* p\_vbat\_32k\_clk\_src\_select
* p\_vbat\_32k\_suppress\_configure
* p\_vbat\_pfr\_sts\_reg\_get
* p\_vbat\_pfr\_sts\_bit\_clr
* p\_vbat\_monotonic\_counter\_read
* p\_vbat\_counter\_hiword\_write
* p\_vbat\_counter\_hiword\_read
* p\_vbat\_vwire\_reg\_write
* p\_vbat\_vwire\_reg\_read

### p\_vbat\_32k\_clk\_src\_select

Function Header

**void p\_vbat\_32k\_clk\_src\_select( enum CLK\_CONFIG new\_val )**

Description

Selects the base clock source for the 32KHz domain.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | External clock source selection  PIN\_32KHz\_IN – single ended signal through 32KHz\_IN  INT\_SILICON\_OSC – internal RC oscillator  EXT\_RESONATOR – external crystal  EXT\_SINGLE\_ENDED\_SIG – single ended signal through  XTAL2 |

Outputs

None

### p\_vbat\_32k\_suppress\_configure

Function Header

**void p\_vbat\_32k\_suppress\_configure( uint8\_t new\_val )**

Description

Turns on/off the 32KHz clock domain while on vbat power only.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New configuration value  VBAT\_32K\_DOMAIN\_ON\_WHEN\_VTR\_OFF  VBAT\_32K\_DOMAIN\_OFF\_WHEN\_VTR\_OFF |

Outputs

None

### p\_vbat\_pfr\_sts\_reg\_get

Function Header

**uint8\_t p\_vbat\_pfr\_sts\_reg\_get( void )**

Description

Reads the VBAT status register.

Inputs

None

Outputs

Current contents of the PFR Status register.

### p\_vbat\_pfr\_sts\_bit\_clr

Function Header

**void p\_vbat\_pfr\_sts\_bit\_clr( enum VBAT\_STS\_TYPE new\_val )**

Description

Writes 1 to the specified status bit of the pfr register to clear it.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | Status bit selection setting  TAMPER  SOFT\_RST  RESETI  WDT\_RST  STSRESETREQ  VBAT\_RST |

Outputs

None

### p\_vbat\_monotonic\_counter\_read

Function Header

**uint32\_t p\_vbat\_monotonic\_counter\_read( void )**

Description

Reads the monotonic counter register.

Inputs

None

Outputs

Current contents of the monotonic counter register.

### p\_vbat\_counter\_hiword\_write

Function Header

**void p\_vbat\_counter\_hiword\_write( uint32\_t count )**

Description

Writes to the counter hiword register.

Note - This value combined with the monotonic counter register value will enable the monotonic

counter to act as a 64 bit counter.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New count value |

Outputs

None

### p\_vbat\_counter\_hiword\_read

Function Header

**uint32\_t p\_vbat\_counter\_hiword\_read( void )**

Description

Reads the counter hiword register.

Inputs

None

Outputs

Current contents of the counter hiword register.

### p\_vbat\_vwire\_reg\_write

Function Header

**void p\_vbat\_vwire\_reg\_write( uint8\_t new\_val )**

Description

Writes to the vwire backup register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New value |

Outputs

None

### p\_vbat\_vwire\_reg\_read

Function Header

**uint8\_t p\_vbat\_vwire\_reg\_read( void )**

Description

Reads the vwire backup register.

Inputs

None

Outputs

Current contents of the vwire backup register.

# RPM - PWM

RPM - PWM API’s

rpm\_pwm\_init

get\_current\_fan\_speed

RPM - PWM registers

Fan Setting Register

PWM Divide Register

Fan Configuration Register

Gain Register

Fan Spin Up Configuration Register

Fan Step Register

Fan Minimum Drive Register

Valid Tach Count Register

Fan Drive Fail Band Register

Tach Target Register

Tach Reading Register

PWM Driver Base Frequency Register

Fan Status Register

RPM - PWM Peripheral Functions

p\_rpm\_pwm\_fan\_setting\_reg\_set

p\_rpm\_pwm\_fan\_setting\_reg\_get

p\_rpm\_pwm\_pwm\_divide\_reg\_set

p\_rpm\_pwm\_fan\_config\_reg\_set

p\_rpm\_pwm\_fan\_config\_reg\_get

p\_rpm\_pwm\_mode\_select

p\_rpm\_pwm\_gain\_reg\_set

p\_rpm\_pwm\_spin\_up\_config\_reg\_set

p\_rpm\_pwm\_spin\_up\_config\_reg\_get

p\_rpm\_pwm\_fan\_step\_reg\_set

p\_rpm\_pwm\_minimum\_drive\_reg\_set

p\_rpm\_pwm\_valid\_tach\_count\_reg\_set

p\_rpm\_pwm\_fan\_drv\_fail\_band\_reg\_set

p\_rpm\_pwm\_tach\_target\_reg\_set

p\_rpm\_pwm\_tach\_reading\_reg\_get

p\_rpm\_pwm\_base\_freq\_set

p\_rpm\_pwm\_fan\_status\_reg\_get\_clr

RPM -PWM

instances

RPM- PWM 0

RPM – PWM 1

PCR Peripheral Functions

GPIO Peripheral Functions

Interrupt Peripheral Functions

## RPM - PWM API Functions

The list of VBAT Register Bank peripheral functions are

* rpm\_pwm\_init
* get\_current\_fan\_speed

### rpm\_pwm\_init

Function Header

**void rpm\_pwm\_init( uint8\_t fan\_id, uint8\_t poles, uint16\_t target\_speed )**

Description

Initializes the rpm pwm block for automatic functionality.

Note - This function only applies if the fan speed control algorithm is to be used

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****fan\_id**** | 0 based fan ID |
| ****poles**** | Number of poles that the fan contains (1, 2, 3, 4) |
| ****target\_speed**** | Target fan speed that the algorithm should maintain |

Outputs

None

### get\_current\_fan\_speed

Function Header

**uint16\_t get\_current\_fan\_speed( uint8\_t fan\_id )**

Description

Reads the tach reading register and returns the fan speed in terms of RPM.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****fan\_id**** | 0 based fan ID |

Outputs

None

## RPM - PWM Peripheral Functions

The list of VBAT Register Bank peripheral functions are

* p\_rpm\_pwm\_fan\_setting\_reg\_set
* p\_rpm\_pwm\_fan\_setting\_reg\_get
* p\_rpm\_pwm\_pwm\_divide\_reg\_set
* p\_rpm\_pwm\_fan\_config\_reg\_set
* p\_rpm\_pwm\_fan\_config\_reg\_get
* p\_rpm\_pwm\_gain\_reg\_set
* p\_rpm\_pwm\_spin\_up\_config\_reg\_set
* p\_rpm\_pwm\_spin\_up\_config\_reg\_get
* p\_rpm\_pwm\_mode\_select
* p\_rpm\_pwm\_fan\_step\_reg\_set
* p\_rpm\_pwm\_minimum\_drive\_reg\_set
* p\_rpm\_pwm\_valid\_tach\_count\_reg\_set
* p\_rpm\_pwm\_fan\_drv\_fail\_band\_reg\_set
* p\_rpm\_pwm\_tach\_target\_reg\_set
* p\_rpm\_pwm\_tach\_reading\_reg\_get
* p\_rpm\_pwm\_base\_freq\_set
* p\_rpm\_pwm\_fan\_status\_reg\_get\_clr

### p\_rpm\_pwm\_fan\_setting\_reg\_set

Function Header

**void p\_rpm\_pwm\_fan\_setting\_reg\_set( uint8\_t fan\_id, uint8\_t new\_val )**

Description

Writes the count value in the fan setting register for driving the fan.

Note - This function is applicable only for manual mode.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****fan\_id**** | 0 based fan ID |
| ****new\_val**** | 8 bit value for the fan driver setting |

Outputs

None

### p\_rpm\_pwm\_fan\_setting\_reg\_get

Function Header

**uint8\_t p\_rpm\_pwm\_fan\_setting\_reg\_get( uint8\_t fan\_id )**

Description

Reads the fan setting register.

Note - This function only applies if the fan speed control algorithm is enabled.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****fan\_id**** | 0 based fan ID |

Outputs

Current contents of the fan setting register.

### p\_rpm\_pwm\_pwm\_divide\_reg\_set

Function Header

**void p\_rpm\_pwm\_pwm\_divide\_reg\_set( uint8\_t fan\_id, uint8\_t new\_val )**

Description

Writes the divider value for the optional PWM driver of the RPM – PWM block.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****fan\_id**** | 0 based fan ID |
| ****new\_val**** | 8 bit value for the divider setting |

Outputs

None

### p\_rpm\_pwm\_fan\_config\_reg\_set

Function Header

**void p\_rpm\_pwm\_fan\_config\_reg\_set( uint8\_t fan\_id, uint16\_t new\_val )**

Description

Writes to the fan configuration register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****fan\_id**** | 0 based fan ID |
| ****new\_val**** | 16 bit value for the fan driver setting (Refer to the header file for each bit definitions) |

Outputs

None

### p\_rpm\_pwm\_fan\_config\_reg\_get

Function Header

**uint16\_t p\_rpm\_pwm\_fan\_config\_reg\_get( uint8\_t fan\_id )**

Description

Reads the fan configuration register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****fan\_id**** | 0 based fan ID |

Outputs

Current contents of the fan configuration register.

### p\_rpm\_pwm\_mode\_select

Function Header

**void p\_rpm\_pwm\_mode\_select( uint8\_t fan\_id, uint8\_t new\_val )**

Description

Sets the operating mode for the fan control block.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****fan\_id**** | 0 based fan ID |
| ****new\_val**** | New configuration value  RPM\_FAN\_CONTROL\_MANUAL  RPM\_FAN\_CONTROL\_AUTOMATIC |

Outputs

None

### p\_rpm\_pwm\_gain\_reg\_set

Function Header

**void p\_rpm\_pwm\_gain\_reg\_set( uint8\_t fan\_id, uint8\_t new\_val )**

Description

Writes the gain value settings for the P, I, D parameters of the fan control algorithm.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****fan\_id**** | 0 based fan ID |
| ****new\_val**** | Setting consisting of the gain values for P, I, D (Refer to the header file for each bit definitions) |

Outputs

None

### p\_rpm\_pwm\_spin\_up\_config\_reg\_set

Function Header

**void p\_rpm\_pwm\_spin\_up\_config\_reg\_set( uint8\_t fan\_id, uint8\_t new\_val )**

Description

Writes to the fan spin up configuration register for setting the advanced spin up configurations.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****fan\_id**** | 0 based fan ID |
| ****new\_val**** | New configuration value (Refer to the header file for each bit definitions) |

Outputs

None

### p\_rpm\_pwm\_spin\_up\_config\_reg\_get

Function Header

**uint8\_t p\_rpm\_pwm\_spin\_up\_config\_reg\_get( uint8\_t fan\_id )**

Description

Reads the spin up configuration register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****fan\_id**** | 0 based fan ID |

Outputs

Current contents of the spin up configuration register.

### p\_rpm\_pwm\_fan\_step\_reg\_set

Function Header

**void p\_rpm\_pwm\_fan\_step\_reg\_set( uint8\_t fan\_id, uint8\_t new\_val )**

Description

Writes the value of the maximum step size by which the fan speed can increase at a time during the spin up routine.

Note - This function only applies for no derivative and basic derivative action.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****fan\_id**** | 0 based fan ID |
| ****new\_val**** | New count value |

Outputs

None

### p\_rpm\_pwm\_minimum\_drive\_reg\_set

Function Header

**void p\_rpm\_pwm\_minimum\_drive\_reg\_set( uint8\_t fan\_id, uint8\_t new\_val )**

Description

Writes the minimum fan speed value to the pwm minimum drive register.

Note - This function only applies if the fan speed control algorithm is enabled.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****fan\_id**** | 0 based fan ID |
| ****new\_val**** | New count value |

Outputs

None

### p\_rpm\_pwm\_valid\_tach\_count\_reg\_set

Function Header

**void p\_rpm\_pwm\_valid\_tach\_count\_reg\_set( uint8\_t fan\_id, uint8\_t new\_val )**

Description

Writes the maximum fan speed value to the valid tach count register.

Note - This function only applies if the fan speed control algorithm is enabled.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****fan\_id**** | 0 based fan ID |
| ****new\_val**** | New count value |

Outputs

None

### p\_rpm\_pwm\_fan\_drive\_fail\_band\_reg\_set

Function Header

**void p\_rpm\_pwm\_fan\_drv\_fail\_band\_reg\_set( uint8\_t fan\_id, uint16\_t new\_val )**

Description

Writes the number of tach counts that is used by the fan drive fail detection circuitry.

Note - This function only applies if the fan speed control algorithm is enabled.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****fan\_id**** | 0 based fan ID |
| ****new\_val**** | New count value |

Outputs

None

### p\_rpm\_pwm\_tach\_target\_reg\_set

Function Header

**void p\_rpm\_pwm\_tach\_target\_reg\_set( uint8\_t fan\_id, uint16\_t new\_val )**

Description

Writes the target fan speed value to the tach target register.

Note - This function only applies if the fan speed control algorithm is enabled.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****fan\_id**** | 0 based fan ID |
| ****new\_val**** | New count value |

Outputs

None

### p\_rpm\_tach\_reading\_reg\_get

Function Header

**uint16\_t p\_rpm\_pwm\_tach\_reading\_reg\_get( uint8\_t fan\_id )**

Description

Returns the current fan speed by reading the tach reading register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****fan\_id**** | 0 based fan ID |

Outputs

Current contents of the tach reading register.

### p\_rpm\_pwm\_base\_freq\_set

Function Header

**void p\_rpm\_pwm\_base\_freq\_set( uint8\_t fan\_id, uint8\_t new\_val )**

Description

Sets the frequency range for the optional PWM fan driver.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****fan\_id**** | 0 based fan ID |
| ****new\_val**** | New configuration value  RPM\_FAN\_PWM\_FREQ\_26P8KHz  RPM\_FAN\_PWM\_FREQ\_23P4KHz  RPM\_FAN\_PWM\_FREQ\_4P67KHz  RPM\_FAN\_PWM\_FREQ\_2P34KHz |

Outputs

None

### p\_rpm\_pwm\_fan\_status\_get\_clr

Function Header

**uint8\_t p\_rpm\_pwm\_fan\_status\_get\_clr( uint8\_t fan\_id )**

Description

Reads the fan status register and clears if any of the status bits are set.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****fan\_id**** | 0 based fan ID |

Outputs

Current contents of the fan status register.

# ACPI – ECI

ACPI ECI

instances

ACPI 0

ACPI 1

ACPI 2

ACPI 3

ACPI 4

ACPI ECI registers

EC2OS Data Byte Registers 0 to 3

OS2EC Data Byte Registers 0 to 3

EC Status Register

EC Byte Control Register

ACPI ECI API’s

None

ACPI ECI Peripheral Functions

p\_acpi\_mode\_select

p\_acpi\_ec2os\_data\_byte\_reg\_set

p\_acpi\_os2ec\_data\_byte\_reg\_get

p\_acpi\_status\_bits\_set

p\_acpi\_status\_reg\_get

Interrupt Peripheral Functions

PCR Peripheral Functions

## ACPI ECI Peripheral Functions

The list of ACPI ECI peripheral functions are

* p\_acpi\_mode\_select
* p\_acpi\_ec2os\_data\_byte\_reg\_set
* p\_acpi\_os2ec\_data\_byte\_reg\_get
* p\_acpi\_status\_bits\_set
* p\_acpi\_status\_reg\_get

### p\_acpi\_mode\_select

Function Header

**void p\_acpi\_mode\_select( uint8\_t acpi\_id, uint8\_t mode\_config )**

Description

Sets the operating mode for the specified acpi block instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****acpi\_id**** | 0 based ACPI ID |
| ****mode\_config**** | Operationg mode selection  ACPI\_EC\_LEGACY\_MODE\_ENABLED  ACPI\_EC\_FOUR\_BYTE\_MODE\_ENABLED |

Outputs

None

### p\_acpi\_ ec2os\_data\_byte\_reg\_set

Function Header

**void p\_acpi\_ec2os\_data\_byte\_reg\_set( uint8\_t acpi\_id, uint32\_t data )**

Description

Writes either one byte or 4 bytes of data in to the EC2OS data byte registers depending upon the mode of operation.

Note - This is a common function for both legacy and 4 bytes mode. While using legacy mode;

user should pass only 1 byte of data.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****acpi\_id**** | 0 based ACPI ID |
| ****data**** | Data bytes that is to be written |

Outputs

None

### p\_acpi\_ os2ec\_data\_byte\_reg\_get

Function Header

**uint32\_t p\_acpi\_os2ec\_data\_byte\_reg\_get( uint8\_t acpi\_id )**

Description

Reads either one byte or 4 bytes of data from the OS2EC data byte registers.

Note - This is a common function for both legacy and 4 bytes mode as well as for command and

data bytes.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****acpi\_id**** | 0 based ACPI ID |

Outputs

Currently received command byte or data byte(s)

### p\_acpi\_ status\_bits\_set

Function Header

**void p\_acpi\_status\_bits\_set( uint8\_t acpi\_id, enum EC\_STS\_TYPE bit\_type, uint8\_t new\_val )**

Description

Writes to the specified status register bits for the specified acpi block instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****acpi\_id**** | 0 based ACPI ID |
| ****bit\_type**** | Bit selection setting  UD1A – user defined bit  BURST – configure burst mode  SCI\_EVT – SCI event  SMI\_EVT – SMI event  UD0A – user defined bit |
| ****new\_val**** | Option to set/clear the bit  1 – set the bit  0 – clear the bit |

Outputs

None

### p\_acpi\_ status\_reg\_get

Function Header

**uint8\_t p\_acpi\_status\_reg\_get( uint8\_t acpi\_id )**

Description

Reads the status register for the specified acpi block instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****acpi\_id**** | 0 based ACPI ID |

Outputs

Currently contents of the EC status register.

# 8042 Keyboard Emulator

Note – User will have to change the mux control of GPIO 135 for KBRST functionality using the following

p\_gpio\_mux\_set( PIN\_0135, GPIO\_MUX\_ALT\_FUNC1 );

8042

instances

KBC 0

8042 registers

EC2OS Data Byte Registers 0 to 3

OS2EC Data Byte Registers 0 to 3

EC Status Register

EC Byte Control Register

8042 API’s

None

8042 Peripheral Functions

p\_kbc\_block\_enable\_disable

p\_kbc\_host2ec\_data\_reg\_get

p\_kbc\_ec\_data\_reg\_set

p\_kbc\_ec\_keyboard\_status\_reg\_get

p\_kbc\_ec\_keyboard\_status\_reg\_set

p\_kbc\_keyboard\_control\_reg\_set

p\_kbc\_pcbof\_reg\_get

p\_kbc\_pcbof\_reg\_get

p\_port92\_enable\_disable

p\_port92\_gatea20\_reg\_set

p\_port92\_gatea20\_reg\_get

p\_port92\_setga20l\_reg\_set

p\_port92\_rstga20l\_reg\_set

Interrupt Peripheral Functions

## 

PCR Peripheral Functions

## 8042 Emulator Peripheral Functions

The list of ACPI ECI peripheral functions are

* p\_ kbc\_block\_enable\_disable
* p\_ kbc\_host2ec\_data\_reg\_get
* p\_ kbc\_ec\_data\_reg\_set
* p\_ kbc\_ec\_keyboard\_status\_reg\_get
* p\_ kbc\_ec\_keyboard\_status\_reg\_set
* p\_kbc\_keyboard\_control\_reg\_set
* p\_kbc\_pcbof\_reg\_get
* p\_kbc\_pcbof\_reg\_get
* p\_port92\_enable\_disable
* p\_port92\_gatea20\_reg\_set
* p\_port92\_gatea20\_reg\_get
* p\_port92\_setga20l\_reg\_set
* p\_port92\_rstga20l\_reg\_set

### p\_ kbc\_block\_enable\_disable

Function Header

**void p\_kbc\_block\_enable\_disable( uint8\_t new\_val )**

Description

Enables/disables the 8042 emulator block.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | Block enable/ disable setting  KBC\_BLOCK\_DISABLED  KBC\_BLOCK\_ENABLED |

Outputs

None

### p\_ kbc\_host2ec\_data\_reg\_get

Function Header

**uint8\_t p\_kbc\_host2ec\_data\_reg\_get( void )**

Description

Reads the HOST2EC Data Register.

Inputs

None

Outputs

Current contents of the HOST2EC Data register.

### p\_ kbc\_ ec\_data\_reg\_set

Function Header

**void p\_kbc\_** **ec\_data\_reg\_set ( uint8\_t new\_val )**

Description

Writes data byte to the EC Data Register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | 1 byte data that needs to be sent to host. |

Outputs

None

### p\_ kbc\_ ec\_keyboard\_status\_reg\_get

Function Header

**uint8\_t p\_kbc\_** **ec\_keyboard\_status\_reg\_get ( void )**

Description

Reads the EC Keyboard Status Register.

Inputs

None

Outputs

Current contents of the EC Keyboard Status Register.

### p\_ kbc\_ ec\_keyboard\_status\_reg\_set

Function Header

**void p\_kbc\_ec\_keyboard\_status\_reg\_set( uint8\_t new\_val )**

Description

Writes to the EC Keyboard Status Register.

Note - Only bits UD0(bit 2), UD1(bit 4) and UD2(bits[7:6]) are writable by EC.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | Data to be written into the above mentioned bits |

Outputs

None

### p\_ kbc\_ keyboard\_control\_reg\_set

Function Header

**void p\_kbc\_** **keyboard\_control\_reg\_set ( uint8\_t new\_val )**

Description

Writes to the Keyboard Control Register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New configuration value (Refer to the datasheet) |

Outputs

None

### p\_ kbc\_ pcbof\_reg\_set

Function Header

**void p\_kbc\_** **pcbof\_reg\_set ( uint8\_t new\_val )**

Description

Write to the PCBOF register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New configuration value  PCOBF\_RESET  PCOBF\_SET |

Outputs

None

### p\_ kbc\_ pcbof\_reg\_get

Function Header

**uint8\_t p\_kbc\_** kbc\_ pcbof\_reg\_get **( void )**

Description

Reads the PCBOF register.

Inputs

None

Outputs

Current contents of the PCBOF Register.

### p\_ port92\_block\_enable\_disable

Function Header

**void p\_port92\_block\_enable\_disable( uint8\_t new\_val )**

Description

Enables/disables the port92 block.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | Block enable/ disable setting  PORT92\_BLOCK\_DISABLED  PORT92\_BLOCK\_ENABLED |

Outputs

None

### p\_ port92\_gatea20\_reg\_set

Function Header

**void p\_port92\_gatea20\_reg\_set( uint8\_t new\_val )**

Description

Writes to the GATEA20 Control register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New configuration value  GATEA20\_OUTPUT\_DRIVE\_LOW  GATEA20\_OUTPUT\_DRIVE\_HIGH |

Outputs

None

### p\_ port92\_gatea20\_reg\_get

Function Header

**uint8\_t p\_** **port92\_gatea20\_reg**\_get **( void )**

Description

Reads the GATEA20 register.

Inputs

None

Outputs

Current contents of the GATEA20 Register.

### p\_ setga20l\_reg\_set

Function Header

**void p\_port92\_setga20l\_reg\_set ( uint8\_t new\_val )**

Description

Writes to the SETGA20L register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New configuration value |

Outputs

None

### p\_ rstga20l\_reg\_set

Function Header

**void p\_port92\_rstga20l\_reg\_set ( uint8\_t new\_val )**

Description

Writes to the RSTGA20L register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New configuration value |

Outputs

None

# Port 80

Port 80

instances

Port80 0

Port80 1

Port 80 registers

Activate Register

EC Data Register

Configuration Register

Status Register

Count Register

Port 80 API’s

None

Port 80 Peripheral Functions

p\_port80\_block\_enable\_disable

p\_port80\_host\_data\_read

p\_port80\_configuration\_reg\_write

p\_port80\_status\_reg\_read

p\_port80\_count\_reg\_write

p\_port80\_count\_reg\_read

Interrupt Peripheral Functions

PCR Peripheral Functions

## Port 80 Peripheral Functions

The list of Port 80 peripheral functions are

* p\_port80\_block\_enable\_disable
* p\_port80\_host\_data\_read
* p\_port80\_configuration\_reg\_write
* p\_port80\_status\_reg\_read
* p\_port80\_count\_reg\_write
* p\_port80\_count\_reg\_read

### p\_ port80\_block\_enable\_disable

Function Header

**void p\_port80\_block\_enable\_disable( uint8\_t port80\_id, uint8\_t new\_val )**

Description

Enables/disables the specified Port 80 instance.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****port80\_id**** | 0 based Port 80 ID |
| ****new\_val**** | New Configuration Value  PORT\_80\_DISABLED  PORT\_80\_ENABLED |

Outputs

None

### p\_ port80\_host\_data\_read

Function Header

**uint32\_t p\_port80\_host\_data\_read( uint8\_t port80\_id )**

Description

Reads the data received from the host using EC Data Register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****port80\_id**** | 0 based Port 80 ID |

Outputs

Current contents of the EC Data register.

### p\_ port80\_configuration\_reg\_write

Function Header

**void p\_port80\_configuration\_reg\_write( uint8\_t port80\_id, uint32\_t new\_val )**

Description

Writes to the Configuration Register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****port80\_id**** | 0 based Port 80 ID |
| ****new\_val**** | New configuration value (Refer to the datasheet and header file for possible values) |

Outputs

None

### p\_ port80\_status\_reg\_read

Function Header

**uint32\_t p\_port80\_status\_reg\_read( uint8\_t port80\_id )**

Description

Reads the Status Register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****port80\_id**** | 0 based Port 80 ID |

Outputs

Current contents of the Status Register.

### p\_port80\_count\_reg\_write

Function Header

**void p\_port80\_count\_reg\_write( uint8\_t port80\_id, uint32\_t new\_val )**

Description

Writes the timer reload value to the Count Register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****port80\_id**** | 0 based Port 80 ID |
| ****new\_val**** | 24 bit count value |

Outputs

None

### p\_ port80\_count\_reg\_read

Function Header

**uint32\_t p\_port80\_count\_reg\_read( uint8\_t port80\_id )**

Description

Reads the Count Register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****port80\_id**** | 0 based Port 80 ID |

Outputs

Current contents of the Status Register.

# RTOS Timer

RTOS Timer instances

RTOS Timer

RTOS Timer registers

RTOS Timer Count Register

RTOS Timer Preload Register

RTOS Timer Control Register

Soft Interrupt Register

RTOS Timer API’s

rtos\_timer\_init

rtos\_timer\_change\_

preload\_value

RTOS Timer Peripheral Functions

p\_rtos\_block\_enable\_disable p\_rtos\_timer\_start\_stop

p\_rtos\_timer\_pause\_unpause

p\_rtos\_timer\_mode\_select

p\_rtos\_timer\_ext\_halt\_configure

p\_rtos\_timer\_counter\_reg\_get

p\_rtos\_timer\_preload\_reg\_set

p\_rtos\_timer\_control\_reg\_get

p\_rtos\_timer\_soft\_interrupt\_reg\_set

Interrupt Peripheral Functions

PCR Peripheral Functions

## RTOS Timer API Functions

The list RTOS Timer API functions are

* rtos\_timer\_init
* rtos\_timer\_change\_preload\_value

### rtos\_timer\_init

Function Header

**void rtos\_timer\_init( uint32\_t preload\_val, uint8\_t mode, uint8\_t ext\_halt )**

Description

Initializes the rtos timer block and starts the counting.

Note –

1. By default the ext\_halt parameter should be set to RTOS\_TIMER\_EXT\_HARDWARE\_HALT\_DIS.
2. 1 count of the timer is equal to 30.51 us.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****preload\_val**** | 32 bit preload count value |
| ****mode**** | Operating mode selection  RTOS\_TIMER\_IN\_ONE\_SHOT\_MODE  RTOS\_TIMER\_IN\_CONTINUOUS\_MODE |
| ****ext\_halt**** | External hardware halt selection  RTOS\_TIMER\_EXT\_HARDWARE\_HALT\_DIS  RTOS\_TIMER\_EXT\_HARDWARE\_HALT\_EN |

Outputs

None

### rtos\_ timer\_change\_preload\_value

Function Header

**void rtos\_timer\_change\_preload\_value( uint32\_t preload\_val )**

Description

Sets new preload value for the RTOS timer during its operation.

Note - 1 count of the timer is equal to 30.51 us.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****preload\_val**** | 32 bit preload count value |

Outputs

None

## RTOS Timer Peripheral Functions

The list of RTOS Timer peripheral functions are

* p\_rtos\_block\_enable\_disable
* p\_rtos\_timer\_start\_stop
* p\_rtos\_timer\_pause\_unpause
* p\_rtos\_timer\_mode\_select
* p\_rtos\_timer\_ext\_halt\_configure
* p\_rtos\_timer\_counter\_reg\_get
* p\_rtos\_timer\_preload\_reg\_set
* p\_rtos\_timer\_control\_reg\_get
* p\_rtos\_timer\_soft\_interrupt\_reg\_set

### p\_rtos\_block\_enable\_disable

Function Header

**void p\_rtos\_block\_enable\_disable( uint8\_t new\_val )**

Description

Enables/disables the RTOS timer block.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New Configuration Value  RTOS\_TIMER\_BLOCK\_DISABLE  RTOS\_TIMER\_BLOCK\_ENABLE |

Outputs

None

### p\_rtos\_timer\_start\_stop

Function Header

**void p\_rtos\_timer\_start\_stop( uint8\_t new\_val )**

Description

Starts/stops the RTOS timer.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New Configuration Value  RTOS\_TIMER\_STOP  RTOS\_TIMER\_START |

Outputs

None

### p\_rtos\_timer\_pause\_unpause

Function Header

**void p\_rtos\_timer\_pause\_unpause( uint8\_t new\_val )**

Description

Pauses/unpauses the RTOS timer count.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New Configuration Value  RTOS\_TIMER\_RUN  RTOS\_TIMER\_PAUSE |

Outputs

None

### p\_rtos\_timer\_ mode\_select

Function Header

**void p\_rtos\_timer\_mode\_select( uint8\_t new\_val )**

Description

Configures the operating mode for the RTOS timer.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New Configuration Value  RTOS\_TIMER\_IN\_ONE\_SHOT\_MODE  RTOS\_TIMER\_IN\_CONTINUOUS\_MODE |

Outputs

None

### p\_rtos\_ext\_halt\_configure

Function Header

**void p\_rtos\_timer\_ext\_halt\_configure( uint8\_t new\_val )**

Description

Enables/disables the external hardware halt functionality for the RTOS timer.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New Configuration Value  RTOS\_TIMER\_EXT\_HARDWARE\_HALT\_DIS  RTOS\_TIMER\_EXT\_HARDWARE\_HALT\_EN |

Outputs

None

### p\_rtos\_counter\_reg\_get

Function Header

**uint32\_t p\_rtos\_timer\_counter\_reg\_get( void )**

Description

Reads the RTOS timer counter register.

Inputs

None

Outputs

Current count value in the counter register.

### p\_rtos\_preload\_reg\_set

Function Header

**void p\_rtos\_timer\_preload\_reg\_set( uint32\_t new\_val )**

Description

Writes to the RTOS timer preload register.

Note - 1 count of the timer is equal to 30.51us.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New count value |

Outputs

None

### p\_rtos\_ control\_reg\_get

Function Header

**uint32\_t p\_rtos\_timer\_control\_reg\_get( void )**

Description

Reads the RTOS timer control register.

Inputs

None

Outputs

Current count value in the control register.

### p\_rtos\_ soft\_interrupt\_reg\_set

Function Header

**void p\_rtos\_timer\_soft\_interrupt\_reg\_set( uint32\_t new\_val )**

Description

Writes to the RTOS timer soft interrupt register.

Note - Only bits[3:0] are defined. Writing a 1 to each of these bits will generate SWI\_3, SWI\_2,

SWI\_1, SWI\_0 software interrupt signals to the EC respectively.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New configuration value |

Outputs

None

# EEPROM

EEPROM API’s

eeprom\_block\_init

EEPROM Peripheral Functions

p\_eeprom\_enable\_disable

p\_eeprom\_soft\_reset

p\_eeprom\_execute\_reg\_set

p\_eeprom\_status\_reg\_get

p\_eeprom\_status\_reg\_set

p\_eeprom\_interrupt\_enable\_reg\_set

p\_eeprom\_password\_reg\_set

p\_eeprom\_unlock\_reg\_set

p\_eeprom\_lock\_reg\_set

p\_eeprom\_data\_buffer\_write

p\_eeprom\_data\_buffer\_read

EEPROM instances

EEPROM 0

EEPROM registers

EEPROM Mode Register

EEPROM Execute register

EEPROM Status Register

EEPROM Interrupt Enable register

EEPROM Password register

EEPROM Unlock Register

EEPROM Lock register

EEPROM Buffer register

## 

Interrupt Peripheral Functions

PCR Peripheral Functions

## EEPROM API Functions

The list EEPROM API functions are

* eeprom\_block\_init

### eeprom\_block\_init

Function Header

**void eeprom\_block\_init( uint8\_t interrupt\_setting )**

Description

Initializes and activates the eeprom block.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****interrupt\_setting**** | Interrupt enable selection  EEPROM\_TRANSFER\_COMPLETE\_IE\_ENABLED  EEPROM\_EXECUTION\_ERROR\_IE\_ENABLED  EEPROM\_ALL\_INTERRUPTS\_ENABLED |

Outputs

None

## EEPROM Peripheral Functions

The list EEPROM peripheral functions are

* p\_eeprom\_enable\_disable
* p\_eeprom\_soft\_reset
* p\_eeprom\_execute\_reg\_set
* p\_eeprom\_status\_reg\_get
* p\_eeprom\_status\_reg\_set
* p\_eeprom\_interrupt\_enable\_reg\_set
* p\_eeprom\_password\_reg\_set
* p\_eeprom\_unlock\_reg\_set
* p\_eeprom\_lock\_reg\_set
* p\_eeprom\_data\_buffer\_write
* p\_eeprom\_data\_buffer\_read

### p\_eeprom\_block\_init

Function Header

**void p\_eeprom\_enable\_disable( uint8\_t new\_val )**

Description

Enables/disables the eeprom block.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | Block enable/disable selection  EEPROM\_BLOCK\_DISABLED  EEPROM\_BLOCK\_ENABLED |

Outputs

None

### p\_eeprom\_soft\_reset

Function Header

**void p\_eeprom\_soft\_reset( void )**

Description

Performs a soft reset of the eeprom block.

Inputs

None

Output

None

### p\_eeprom\_block\_init

Function Header

**void p\_eeprom\_execute\_reg\_set( uint32\_t transfer\_size, uint32\_t command, \**

**uint16\_t eeprom\_addr )**

Description

Writes to the eeprom execute register.

Notes –

1. The transfer\_size and eeprom\_addr fields are valid for read and write commands only. For other commands; set them to 0.
2. eeprom\_addr is valid from 0x0000 to 0x07FF only.
3. A maximum of 32 bytes can be written at a time only.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****transfer\_size**** | Number of bytes to be transferred between eeprom fabric and the data buffer.  Writing a 0 will transfer 32 bytes |
| ****command**** | command that is to be executed  EEPROM\_READ  EEPROM\_WRITE  EEPROM\_READ\_STATUS  EEPROM\_WRITE\_STATUS |
| ****eeprom\_addr**** | start address in the eeprom where the data has to be written |

Outputs

None

### p\_eeprom\_status\_reg\_get

Function Header

**uint32\_t p\_eeprom\_status\_reg\_get( void )**

Description

Reads the eeprom status register.

Inputs

None

Outputs

Current contents of the register.

### p\_eeprom\_status\_reg\_set

Function Header

**void p\_eeprom\_status\_reg\_set( uint32\_t new\_val )**

Description

Writes to the eeprom status register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | Status bits that needs to be cleared. |

Outputs

None

### p\_eeprom\_interrupt\_enable\_reg\_set

Function Header

**void p\_eeprom\_interrupt\_enable\_reg\_set( uint32\_t new\_val )**

Description

Writes to the eeprom interrupt enable register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | Interrupt enable selection  EEPROM\_TRANSFER\_COMPLETE\_IE\_ENABLED  EEPROM\_EXECUTION\_ERROR\_IE\_ENABLED  EEPROM\_ALL\_INTERRUPTS\_ENABLED |

Outputs

None

### p\_eeprom\_password\_reg\_set

Function Header

**void p\_eeprom\_password\_reg\_set( uint32\_t new\_val )**

Description

Writes to the eeprom password register.

Note – Bit 31 is reserved.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New password value |

Outputs

None

### p\_eeprom\_unlock\_reg\_set

Function Header

**void p\_eeprom\_unlock\_reg\_set( uint32\_t new\_val )**

Description

Writes to the eeprom unlock register.

Note – Bit 31 is reserved.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | Value matching to the Password register |

Outputs

None

### p\_eeprom\_lock\_reg\_set

Function Header

**void p\_eeprom\_lock\_reg\_set( uint32\_t new\_val )**

Description

Writes to the eeprom lock register.

Note – Bits[31:2] are reserved.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New configuration value  EEPROM\_JTAG\_LOCKED  EEPROM\_ACCESS\_IS\_LOCKED |

Outputs

None

### p\_eeprom\_data\_buffer\_write

Function Header

void p\_eeprom\_data\_buffer\_write( uint8\_t transfer\_size, uint8\_t \*source\_ptr )

Description

Writes data to the eeprom hardware buffer from the specified source.

Note – A maximum of 32 bytes can be written at a time only.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****transfer\_size**** | Number of bytes to be transferred. (0 to 32) |
| ****source\_ptr**** | Pointer to the buffer containing the data |

Outputs

None

### p\_eeprom\_data\_buffer\_read

Function Header

**void p\_eeprom\_data\_buffer\_read( uint8\_t transfer\_size, uint8\_t \*destination\_ptr )**

Description

Reads data from the eeprom hardware buffer to the specified destination.

Note – A maximum of 32 bytes can be read at a time only.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****transfer\_size**** | Number of bytes to be transferred. (0 to 32) |
| ****destination\_ptr**** | Pointer to the buffer to store the data |

Outputs

None

# eSPI

eSPI Global Peripheral Functions

p\_espi\_device\_id\_read

p\_espi\_global\_capabilities\_0\_reg\_write

p\_espi\_global\_capabilities\_1\_reg\_read

p\_espi\_global\_capabilities\_1\_reg\_write

p\_espi\_pc\_capabilities\_reg\_write

p\_espi\_vwire\_channel\_capabilities\_reg\_write

p\_espi\_oob\_channel\_capabilities\_reg\_write

p\_espi\_flash\_channel\_capabilities\_reg\_write

p\_espi\_pc\_ready\_reg\_write

p\_espi\_oob\_channel\_ready\_reg\_write

p\_espi\_flash\_channel\_ready\_reg\_write

p\_espi\_reset\_status\_get\_clr

p\_espi\_reset\_interrupt\_enable\_reg\_write

p\_espi\_pltrst\_source\_reg\_write

eSPI Global registers

eSPI Capabilities ID register

eSPI Global capabilities 0 register

eSPI Global capabilities 1 register

eSPI Peripheral Channnel capabilities register

eSPI virtual channel capabilities register

eSPI OOB channel capabilities register

eSPI flash channel capabilities register

eSPI peripheral channel ready register

eSPI OOB channel ready register

eSPI flash channel ready register

eSPI interrupt status register

eSPI interrupt enable register

PLTRST source register

eSPI

instances

eSPI 0

eSPI Global API’s

espi\_block\_init

espi\_alert\_signal\_mode\_get

Interrupt Peripheral Functions

PCR Peripheral Functions

## eSPI Global API Functions

The list eSPI Global API functions are

* espi\_block\_init
* espi\_alert\_signal\_mode\_get

### espi\_block\_init

Function Header

**espi\_block\_init( ESPI\_CHANNELS channel\_sel, uint8\_t io\_mode, uint8\_t max\_freq,**

**uint8\_t pltrst\_src, uint8\_t espi\_rst\_int\_en )**

Description

Configures all the necessary hardware parameters for the eSPI block.

Note –

1. By default, the 'espi\_rst\_int\_en' parameter should be set to ESPI\_RESET\_INTERRUPT\_DISABLE.
2. This function configures the eSPI gpio pins as well.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****channel\_sel**** | eSPI channel activation selection  NO\_CHANNELS\_SUPPORTED  ONLY\_PERIPHERAL\_CHANNEL  ONLY\_VWIRE\_CHANNEL  PERIPHERAL\_AND\_VWIRE  ONLY\_OOB\_CHANNEL  OOB\_AND\_PERIPHERAL  OOB\_AND\_VWIRE  OOB\_VWIRE\_PERIPHERAL  ONLY\_FLASH\_CHANNEL  FLASH\_AND\_PERIPHERAL  FLASH\_AND\_VWIRE  FLASH\_VWIRE\_PERIPHERAL  FLASH\_AND\_OOB  FLASH\_OOB\_PERIPHERAL  FLASH\_OOB\_VWIRE  ALL\_CHANNELS\_SUPPORTED |
| ****io\_mode**** | I/O modes supported by the slave  ESPI\_SLV\_SINGLE\_IO  ESPI\_SLV\_SINGLE\_DOUBLE\_IO  ESPI\_SLV\_SINGLE\_QUAD\_IO  ESPI\_SLV\_SINGLE\_DOUBLE\_QUAD\_IO |
| ****max\_freq**** | Maximum operating frequency for eSPI block  ESPI\_SLV\_MAX\_FREQ\_20MHZ  ESPI\_SLV\_MAX\_FREQ\_25MHZ  ESPI\_SLV\_MAX\_FREQ\_33MHZ  ESPI\_SLV\_MAX\_FREQ\_50MHZ  ESPI\_SLV\_MAX\_FREQ\_66MHZ |
| ****pltrst\_src**** | Source selection for the PLTRST# reset event  PLTRST\_RST\_FROM\_PLTRST\_VWIRE  PLTRST\_RST\_FROM\_EXTERNAL\_SIGNAL |
| ****espi\_rst\_int\_en**** | eSPI\_RESET# interrupt enable selection  ESPI\_RST\_INT\_ENABLE |

Outputs

None

### espi\_alert\_signal\_mode\_get

Function Header

**uint8\_t espi\_alert\_signal\_mode\_get( void )**

Description

Returns the current operating mode for the ALERT# signal set by the host.

Inputs

None

Output

0(ALERT# muxed on IO[1]), 1(ALERT# on dedicated pin)

## eSPI Global Peripheral Functions

The list eSPI Global peripheral functions are

* p\_espi\_device\_id\_read
* p\_espi\_global\_capabilities\_0\_reg\_write
* p\_espi\_global\_capabilities\_1\_reg\_read
* p\_espi\_global\_capabilities\_1\_reg\_write
* p\_espi\_pc\_capabilities\_reg\_write
* p\_espi\_vwire\_channel\_capabilities\_reg\_write
* p\_espi\_oob\_channel\_capabilities\_reg\_write
* p\_espi\_flash\_channel\_capabilities\_reg\_write
* p\_espi\_pc\_ready\_reg\_write
* p\_espi\_oob\_channel\_ready\_reg\_write
* p\_espi\_flash\_channel\_ready\_reg\_write
* p\_espi\_reset\_status\_get\_clr
* p\_espi\_reset\_interrupt\_enable\_reg\_write
* p\_espi\_pltrst\_source\_reg\_write

### p\_espi\_device\_id\_read

Function Header

**uint8\_t p\_espi\_device\_id\_read( void )**

Description

Reads the espi capabilities id register.

Note – The default value should not be changed.

Inputs

None

Output

Current eSPI device ID value.

### p\_espi\_global\_capabilities\_0\_reg\_write

Function Header

**void p\_espi\_global\_capabilities\_0\_reg\_write( ESPI\_CHANNELS new\_val )**

Description

Writes to espi global capabilities 0 register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****channel\_sel**** | eSPI channel activation selection  NO\_CHANNELS\_SUPPORTED  ONLY\_PERIPHERAL\_CHANNEL  ONLY\_VWIRE\_CHANNEL  PERIPHERAL\_AND\_VWIRE  ONLY\_OOB\_CHANNEL  OOB\_AND\_PERIPHERAL  OOB\_AND\_VWIRE  OOB\_VWIRE\_PERIPHERAL  ONLY\_FLASH\_CHANNEL  FLASH\_AND\_PERIPHERAL  FLASH\_AND\_VWIRE  FLASH\_VWIRE\_PERIPHERAL  FLASH\_AND\_OOB  FLASH\_OOB\_PERIPHERAL  FLASH\_OOB\_VWIRE  ALL\_CHANNELS\_SUPPORTED |

Outputs

None

### p\_espi\_global\_capabilities\_1\_reg\_write

Function Header

**void p\_espi\_global\_capabilities\_1\_reg\_write( uint8\_t new\_val )**

Description

Writes to espi global capabilities 1 register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New configuration value (Refer to the header file for corresponding macros) |

Outputs

None

### p\_espi\_pc\_capabilities\_reg\_write

Function Header

**void p\_espi\_pc\_capabilities\_reg\_write( uint8\_t new\_val )**

Description

Writes to espi peripheral channel capabilities register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New configuration value (Refer to the header file for corresponding macros) |

Outputs

None

### p\_espi\_vwire\_channel\_capabilities\_reg\_write

Function Header

**void p\_espi\_vwire\_channel\_capabilities\_reg\_write( uint8\_t new\_val )**

Description

Writes to espi vwire channel capabilities register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New configuration value (Refer to the header file for corresponding macros) |

Outputs

None

### p\_espi\_oob\_channel\_capabilities\_reg\_write

Function Header

**void p\_espi\_oob\_channel\_capabilities\_reg\_write( uint8\_t new\_val )**

Description

Writes to espi oob channel capabilities register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New configuration value (Refer to the header file for corresponding macros) |

Outputs

None

### p\_espi\_flash\_channel\_capabilities\_reg\_write

Function Header

**void p\_espi\_flash\_channel\_capabilities\_reg\_write( uint8\_t new\_val )**

Description

Writes to espi flash channel capabilities register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New configuration value (Refer to the header file for corresponding macros) |

Outputs

None

### p\_espi\_pc\_ready\_reg\_write

Function Header

**void p\_espi\_pc\_ready\_reg\_write( uint8\_t new\_val )**

Description

Writes to espi peripheral channel ready register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New configuration value (Refer to the header file for corresponding macros) |

Outputs

None

### p\_espi\_oob\_channel\_ready\_reg\_write

Function Header

**void p\_espi\_oob\_channel\_ready\_reg\_write( uint8\_t new\_val )**

Description

Writes to espi oob channel ready register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New configuration value (Refer to the header file for corresponding macros) |

Outputs

None

### p\_espi\_flash\_channel\_ready\_reg\_write

Function Header

**void p\_espi\_flash\_channel\_ready\_reg\_write( uint8\_t new\_val )**

Description

Writes to espi flash channel ready register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New configuration value (Refer to the header file for corresponding macros) |

Outputs

None

### p\_espi\_reset\_interrupt\_enable\_reg\_write

Function Header

**void p\_espi\_reset\_interrupt\_enable\_reg\_write( uint8\_t new\_val )**

Description

Writes to espi reset interrupt enable register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | eSPI\_RESET# interrupt enable selection  ESPI\_RST\_INT\_ENABLE |

Outputs

None

### p\_espi\_pltrst\_source\_reg\_write

Function Header

**void p\_espi\_pltrst\_source\_reg\_write( uint8\_t new\_val )**

Description

Writes to espi pltrst source register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | Source selection for the PLTRST# reset event  PLTRST\_RST\_FROM\_PLTRST\_VWIRE  PLTRST\_RST\_FROM\_EXTERNAL\_SIGNAL |

Outputs

None

eSPI PC Peripheral Functions

p\_espi\_activate\_reg\_write

p\_espi\_pc\_last\_cycle\_reg\_read

p\_espi\_pc\_error\_address\_reg\_read

p\_espi\_pc\_status\_reg\_get\_clr

p\_espi\_pc\_interrupt\_enable\_reg\_write

p\_espi\_bar\_inhibit\_reg\_write

p\_espi\_bar\_init\_reg\_write

p\_espi\_ec\_irq\_reg\_write

p\_espi\_io\_bar\_config\_reg\_write

p\_espi\_io\_bar\_read

p\_espi\_io\_bar\_write

p\_espi\_ltr\_status\_reg\_get\_clr

p\_espi\_ltr\_peripheral\_enable\_reg\_write

p\_espi\_ltr\_control\_reg\_write

p\_espi\_ltr\_tx\_start

p\_espi\_ltr\_peripheral\_msg\_reg\_write

p\_espi\_mem\_bar\_read

p\_espi\_mem\_bar\_write

p\_espi\_sram\_bar\_write

p\_espi\_bus\_mstr\_status\_get\_clr

p\_espi\_bus\_mstr\_int\_en\_reg\_write

p\_espi\_bus\_mstr\_config\_reg\_write

p\_espi\_bus\_mstr\_ctrl\_reg\_write

p\_espi\_bus\_mstr\_tx\_start

p\_espi\_bus\_mstr\_tx\_abort

p\_espi\_bus\_mstr\_host\_addr\_reg\_write

p\_espi\_bus\_mstr\_intrnl\_addr\_reg\_write

eSPI PC registers

eSPI Activate Register

I/O BAR Configuration Register

PC Last Cycle Register

PC Error Address Register

PC Status Register

PC Interrupt Enable Register

BAR Inhibit Register

eSPI BAR Init Register

EC IRQ Register

I/O BAR Internal Component

LTR Peripheral Status Register

LTR Peripheral Enable Register

LTR Peripheral Control Register

LTR Peripheral Message Register

eSPI

instances

eSPI 0

eSPI PC API’s

espi\_pc\_is\_enabled

espi\_pc\_mastering\_is\_enabled

espi\_ltr\_not\_busy

espi\_bus\_mstr\_not\_busy

Interrupt Peripheral Functions

PCR Peripheral Functions

## eSPI PC API Functions

The list eSPI PC API functions are -

* espi\_pc\_is\_enabled
* espi\_pc\_mastering\_is\_enabled
* espi\_ltr\_not\_busy
* espi\_bus\_mstr\_not\_busy

### espi\_pc\_is\_enabled

Function Header

**uint8\_t espi\_pc\_is\_enabled( void )**

Description

Checks if the Peripheral channel has been enabled by the host, and clears all the necessary hardware bits.

Inputs

None

Outputs

0(PC is disabled), 1(PC is enabled)

### espi\_pc\_mastering\_is\_enabled

Function Header

**uint8\_t espi\_pc\_mastering\_is\_enabled( void )**

Description

Checks if the Peripheral channel is enabled for EC mastering by the host and clears the necessary hardware bits.

Inputs

None

Outputs

0(PC mastering is disabled), 1(PC mastering is enabled)

### espi\_ltr\_is\_not\_busy

Function Header

**uint32\_t espi\_ltr\_not\_busy( void )**

Description

This function does the following –

* checks the status of the TRANSMIT\_BUSY bit in the LTR Status register and returns
  + ‘0b’, if TRANSMIT\_BUSY == 1
  + Current contents of the status register, if TRANSMIT\_BUSY == 0
* Clears all the necessary status bits in the hardware

Inputs

None

Outputs

0(LTR transfer ongoing), non-zero(transfer complete)

### espi\_bus\_mstr\_not\_busy

Function Header

**uint16\_t espi\_bus\_mstr\_not\_busy( ESPI\_BUS\_MASTERS bus\_master\_sel )**

Description

This function does the following –

* checks the status of the BM\_BUSY bit in the LTR Status register and returns
  + ‘0b’, if BM\_BUSY == 1
  + Current contents of the status bits, if BM\_BUSY == 0
* Clears all the necessary status bits in the hardware

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****bus\_master\_sel**** | Bus master selection  BUS\_MASTER\_1  BUS\_MASTER\_2 |

Outputs

0(channel busy), non-zero(transfer complete)

## eSPI PC Peripheral Functions

The list eSPI PC Peripheral functions are -

* p\_espi\_activate\_reg\_write
* p\_espi\_pc\_last\_cycle\_reg\_read
* p\_espi\_pc\_error\_address\_reg\_read
* p\_espi\_pc\_status\_reg\_get\_clr
* p\_espi\_pc\_interrupt\_enable\_reg\_write
* p\_espi\_bar\_inhibit\_reg\_write
* p\_espi\_bar\_init\_reg\_write
* p\_espi\_ec\_irq\_reg\_write
* p\_espi\_io\_bar\_config\_reg\_write
* p\_espi\_io\_bar\_read
* p\_espi\_io\_bar\_write
* p\_espi\_ltr\_status\_reg\_get\_clr
* p\_espi\_ltr\_peripheral\_enable\_reg\_write
* p\_espi\_ltr\_control\_reg\_write
* p\_espi\_ltr\_tx\_start
* p\_espi\_ltr\_peripheral\_msg\_reg\_write
* p\_espi\_mem\_bar\_read
* p\_espi\_mem\_bar\_write
* p\_espi\_sram\_bar\_write
* p\_espi\_bus\_mstr\_status\_get\_clr
* p\_espi\_bus\_mstr\_int\_en\_reg\_write
* p\_espi\_bus\_mstr\_config\_reg\_write
* p\_espi\_bus\_mstr\_ctrl\_reg\_write
* p\_espi\_bus\_mstr\_tx\_start
* p\_espi\_bus\_mstr\_tx\_abort
* p\_espi\_bus\_mstr\_host\_addr\_reg\_write
* p\_espi\_bus\_mstr\_intrnl\_addr\_reg\_write

### p\_espi\_activate\_reg\_write

Function Header

**void p\_espi\_activate\_reg\_write( uint32\_t new\_val )**

Description

Writes to the eSPI activate register.

Note - This register should be configured before RMRST# is released and eSPI\_RESET# signal

is de - asserted.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | eSPI block activate selection  ESPI\_BLOCK\_ENABLE |

Outputs

None

### p\_espi\_pc\_last\_cycle\_reg\_read

Function Header

**void p\_espi\_pc\_last\_cycle\_reg\_read( uint32\_t \*destination\_ptr )**

Description

Returns the address, length, cycle type and tag values of the most recent PC transaction.

Note - This is a 96 bit register. A single call to this function will return all 96 bits LSB first.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****destination\_ptr**** | Pointer to the destination buffer |

Outputs

None

### p\_espi\_pc\_error\_address\_reg\_read

Function Header

**void p\_espi\_pc\_error\_address\_reg\_read( uint32\_t \*destination\_ptr )**

Description

Returns the address of the most recent PC transaction that incurred an error.

Note - This is a 64 bit register. A single call to this function will return all the 64 bits LSB first.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****destination\_ptr**** | Pointer to the destination buffer |

Outputs

None

### p\_espi\_pc\_status\_reg\_get\_clr

Function Header

**uint32\_t p\_espi\_pc\_status\_reg\_get\_clr( void )**

Description

Reads the peripheral channel status register and clears any of the status bits, if set.

Inputs

None

Outputs

Current contents of the register.

### p\_espi\_flash\_channel\_ready\_reg\_write

Function Header

**void p\_espi\_pc\_interrupt\_enable\_reg\_write( uint32\_t new\_val )**

Description

Writes to espi peripheral channel interrupt enable register.

Note - By default, 'new\_val' should be set to ESPI\_PC\_ALL\_INT\_DISABLED.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New configuration value (Refer to the header file for corresponding macros) |

Outputs

None

### p\_espi\_bar\_inhibit\_reg\_write

Function Header

**void p\_espi\_bar\_inhibit\_reg\_write( uint8\_t logical\_device )**

Description

Disables the BAR of the specified logical device by writing a '1b' to the bar inhibit register bit of that device.

Note –

1. This register should be configured before RMRST# is released and eSPI\_RESET# signal is de - asserted.
2. The permissible values for logical\_device is 0x00 to 0x3F only.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****logical\_device**** | eSPI logical device selection |

Outputs

None

### p\_espi\_bar\_init\_reg\_write

Function Header

**void p\_espi\_bar\_inhibit\_reg\_write( uint8\_t logical\_device )**

Description

Writes to espi BAR init register.

Note –

1. Only bits[15:0] are currently valid for MEC2016.
2. By default, the BAR\_Init value should be set to 0x002E.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | Host I/O address value |

Outputs

None

### p\_espi\_ec\_irq\_reg\_write

Function Header

**void p\_espi\_ec\_irq\_reg\_write( uint32\_t new\_val )**

Description

Writes to ec irq register.

Note – Only bit[0] is currently valid for MEC2016.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New configuration value (Refer to the header file for corresponding macros) |

Outputs

None

### p\_espi\_io\_bar\_config\_reg\_write

Function Header

**void p\_espi\_io\_bar\_config\_reg\_write( ESPI\_IO\_LOGICAL\_DEVICES device\_type,**

**uint32\_t host\_io\_addr, uint8\_t valid\_en )**

Description

writes to the I/O BAR Configuration register of the specified logical device.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****device\_type**** | New configuration value (Refer to the header file for corresponding macros) |
| ****host\_io\_addr**** | Host I/O address value for the selected logical device |
| ****valid\_en**** | BAR valid selection  ESPI\_IO\_BAR\_VALID |

Outputs

None

### p\_espi\_io\_bar\_read

Function Header

**uint32\_t p\_espi\_io\_bar\_read( ESPI\_IO\_LOGICAL\_DEVICES device\_type )**

Description

reads the I/O BAR of the specified logical device.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****device\_type**** | New configuration value (Refer to the header file for corresponding macros) |

Outputs

Current contents of the BAR.

### p\_espi\_io\_bar\_write

Function Header

**void p\_espi\_io\_bar\_write( ESPI\_IO\_LOGICAL\_DEVICES device\_type, uint32\_t new\_val )**

Description

writes to the I/O BAR of the specified logical device.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****device\_type**** | New configuration value (Refer to the header file for corresponding macros) |
| ****new\_val**** | New configuration value (Refer to the header file for corresponding macros) |

Outputs

None

### p\_espi\_ltr\_status\_reg\_get\_clr

Function Header

**uint32\_t p\_espi\_ltr\_status\_reg\_get\_clr( void )**

Description

reads the LTR peripheral status register and clears any of the status bits, if set.

Inputs

None

Outputs

Current contents of the register

### p\_espi\_ltr\_peripheral\_enable\_reg\_write

Function Header

**void p\_espi\_ltr\_peripheral\_enable\_reg\_write( uint32\_t new\_val )**

Description

writes to LTR peripheral enable register.

Note –

1. Only bit[0] is currently valid for MEC2016.
2. By default, 'new\_val' should be set to ESPI\_LTR\_ALL\_INT\_DISABLED.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New configuration value (Refer to the header file for corresponding macros) |

Outputs

None

### p\_espi\_ltr\_control\_reg\_write

Function Header

**void p\_espi\_ltr\_control\_reg\_write( uint32\_t new\_val )**

Description

writes to the LTR Peripheral Control register.

Note –

1. This function does not write to the LTR\_START bit.
2. Do not use this function if TRANSMIT\_BUSY == 1.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New configuration value (Refer to the header file for corresponding macros) |

Outputs

None

### p\_espi\_ltr\_control\_reg\_write

Function Header

**void p\_espi\_ltr\_tx\_start( void )**

Description

writes a '1b' to the LTR\_START bit in LTR Peripheral Control register.

Note – Do not use this function if TRANSMIT\_BUSY == 1.

Inputs

None

Outputs

None

### p\_espi\_ltr\_peripheral\_msg\_reg\_write

Function Header

**p\_espi\_ltr\_peripheral\_msg\_reg\_write( uint16\_t ltr\_value, uint16\_t ltr\_scale, uint16\_t ltr\_latency )**

Description

writes to LTR peripheral message register.

Note – Do not use this function if TRANSMIT\_BUSY == 1.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****ltr\_value**** | 10 bit LTR value |
| ****ltr\_scale**** | 3 bit LTR scale value |
| ****ltr\_latency**** | LTR latency tolerance mode selection - 0(Infinite latency), 1(latency value decided by scale & length) |

Outputs

None

### p\_espi\_mem\_bar\_read

Function Header

**uint32\_t p\_espi\_mem\_bar\_read( ESPI\_MEM\_LOGICAL\_DEVICES device\_type )**

Description

reads the memory BAR of the specified logical device.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****device\_type**** | New configuration value (Refer to the header file for corresponding macros) |

Outputs

Current contents of the BAR.

### p\_espi\_mem\_bar\_write

Function Header

**void p\_espi\_mem\_bar\_write( ESPI\_MEM\_LOGICAL\_DEVICES device\_type, uint32\_t new\_val )**

Description

write to the memory BAR of the specified logical device.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****device\_type**** | New configuration value (Refer to the header file for corresponding macros) |
| ****new\_val**** | New configuration value (Refer to the header file for corresponding macros) |

Outputs

None

### p\_espi\_sram\_bar\_write

Function Header

**void p\_espi\_sram\_bar\_write( ESPI\_SRAM\_BARS bar\_selection, uint8\_t bar\_valid,**

**uint8\_t ram\_access\_type, uint8\_t ram\_size, uint32\_t ram\_addr )**

Description

configures the specified SRAM BAR.

Note - The permissible value for the 'ram\_size' parameter is 0 to 15 only.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****bar\_selection**** | SRAM BAR selection  SRAM\_BAR\_0  SRAM\_BAR\_1 |
| ****bar\_valid**** | SRAM BAR valid selection  ESPI\_SRAM\_BAR\_VALID |
| ****ram\_access\_type**** | RAM access permission selection for the host  ESPI\_HOST\_HAS\_NO\_ACCESS  ESPI\_HOST\_HAS\_RO\_ACCESS  ESPI\_HOST\_HAS\_WO\_ACCESS  ESPI\_HOST\_HAS\_RW\_ACCESS |
| ****ram\_size**** | Size of the RAM region that is accessible by the selected SRAM BAR |
| ****ram\_addr**** | Base address of the RAM region |

Outputs

None

### p\_espi\_bus\_mstr\_status\_get\_clr

Function Header

**uint16\_t p\_espi\_bus\_mstr\_status\_get\_clr( ESPI\_BUS\_MASTERS bus\_master\_sel )**

Description

reads the status bits for the specified bus master and clears all the necessary bits, if set.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****bus\_master\_sel**** | Bus master selection  BUS\_MASTER\_1  BUS\_MASTER\_2 |

Outputs

Current state of the status bits for the specified bus master.

### p\_espi\_bus\_mstr\_int\_en\_reg\_write

Function Header

**void p\_espi\_bus\_mstr\_int\_en\_reg\_write( ESPI\_BUS\_MASTERS bus\_master\_sel,**

**uint16\_t new\_val )**

Description

writes to the bus master interrupt enable register bits for the specified bus master.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****bus\_master\_sel**** | Bus master selection  BUS\_MASTER\_1  BUS\_MASTER\_2  ALL\_BUS\_MASTERS |
| ****new\_val**** | New configuration value (Refer to the header file for corresponding macros) |

Outputs

None

### p\_espi\_bus\_mstr\_config\_reg\_write

Function Header

**void p\_espi\_bus\_mstr\_config\_reg\_write( ESPI\_BUS\_MASTERS bus\_master\_sel,**

**uint16\_t new\_val )**

Description

writes to the bus master configuration register bits for the specified bus master.

Note - By default, tag should be set to '0b' for bus master 1 and '1b' for bus master 2. Setting the

same tags for both bus masters is illegal.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****bus\_master\_sel**** | Bus master selection  BUS\_MASTER\_1  BUS\_MASTER\_2 |
| ****new\_val**** | 4 bit tag value for the espi bus master traffic |

Outputs

None

### p\_espi\_bus\_mstr\_ctrl\_reg\_write

Function Header

**void p\_espi\_bus\_mstr\_ctrl\_reg\_write( ESPI\_BUS\_MASTERS bus\_master\_sel, uint32\_t new\_val )**

Description

writes to the bus master control register of the specified bus master.

Note –

1. This function does not write to the BM\_START & BM\_ABORT bits.
2. Setting the BM\_LENGTH field to zero or greater than 0x1000 is illegal.
3. Do not use this function if BM\_BUSY == 1.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****bus\_master\_sel**** | Bus master selection  BUS\_MASTER\_1  BUS\_MASTER\_2  ALL\_BUS\_MASTERS |
| ****new\_val**** | New configuration value (Refer to the header file for corresponding macros) |

Outputs

None

### p\_espi\_bus\_mstr\_tx\_start

Function Header

**void p\_espi\_bus\_mstr\_tx\_start( ESPI\_BUS\_MASTERS bus\_master\_sel )**

Description

starts the bus master transfer for the specifiedbus master on its respective channel.

Note – Do not use this function if BM\_BUSY == 1.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****bus\_master\_sel**** | Bus master selection  BUS\_MASTER\_1  BUS\_MASTER\_2 |

Outputs

None

### p\_espi\_bus\_mstr\_tx\_start

Function Header

**void p\_espi\_bus\_mstr\_tx\_abort( ESPI\_BUS\_MASTERS bus\_master\_sel )**

Description

aborts an ongoing bus master transfer for the specified bus master..

Note – Use this function ONLY when BM\_BUSY == 1.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****bus\_master\_sel**** | Bus master selection  BUS\_MASTER\_1  BUS\_MASTER\_2 |

Outputs

None

### p\_espi\_bus\_mstr\_host\_addr\_reg\_write

Function Header

**void p\_espi\_bus\_mstr\_host\_addr\_reg\_write( ESPI\_BUS\_MASTERS bus\_master\_sel,**

**uint32\_t \*host\_addr\_ptr )**

Description

configures the host address used for transfer for the specified bus master.

Note – Do not use this function if BM\_BUSY == 1.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****bus\_master\_sel**** | Bus master selection  BUS\_MASTER\_1  BUS\_MASTER\_2 |
| ****host\_addr\_ptr**** | pointer to the buffer containing the 64 bit host address |

Outputs

None

### p\_espi\_bus\_mstr\_intrnl\_addr\_reg\_write

Function Header

**void p\_espi\_bus\_mstr\_intrnl\_addr\_reg\_write( ESPI\_BUS\_MASTERS bus\_master\_sel,**

**uint32\_t int\_addr )**

Description

configures the internal address used for transfer for the specified bus master.

Note – Do not use this function if BM\_BUSY == 1.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****bus\_master\_sel**** | Bus master selection  BUS\_MASTER\_1  BUS\_MASTER\_2 |
| ****int\_addr**** | DWORD aligned address value |

Outputs

None

eSPI

instances

eSPI 0

eSPI OOB Channel API’s

espi\_oob\_ch\_is\_enabled

espi\_oob\_rx\_is\_done

espi\_oob\_tx\_is\_done

eSPI OOB Channel Peripheral Functions

p\_espi\_oob\_rx\_addr\_reg\_write

p\_espi\_oob\_tx\_addr\_reg\_write

p\_espi\_oob\_rx\_length\_reg\_write

p\_espi\_oob\_rx\_length\_reg\_read

p\_espi\_oob\_tx\_length\_reg\_write

p\_espi\_oob\_rx\_start

p\_espi\_oob\_rx\_control\_reg\_read

p\_espi\_oob\_rx\_interrupt\_enable\_reg\_write

p\_espi\_oob\_rx\_status\_reg\_get\_clr

p\_espi\_oob\_tx\_control\_reg\_write

p\_espi\_oob\_tx\_start

p\_espi\_oob\_tx\_control\_reg\_read

p\_espi\_oob\_tx\_interrupt\_enable\_reg\_write

p\_espi\_oob\_tx\_status\_reg\_get\_clr

eSPI OOB Channel registers

OOB Channel Receive Address Register

OOB Channel Transmit Address Register

OOB Channel Receive Length Register

OOB Channel Transmit Length Register

OOB Channel Receive Control Register

OOB Channel Receive Interrupt Enable Register

OOB Channel Receive Status Register

OOB Channel Transmit Control Register

OOB Channel Transmit Interrupt Enable Register

OOB Channel Transmit Status Register

Interrupt Peripheral Functions

PCR Peripheral Functions

## eSPI OOB Channel API Functions

The list eSPI OOB Channel API functions are -

* espi\_oob\_ch\_is\_enabled
* espi\_oob\_rx\_is\_done
* espi\_oob\_tx\_is\_done

### espi\_ oob\_ch\_is\_enabled

Function Header

**uint8\_t espi\_oob\_ch\_is\_enabled( void )**

Description

checks if the OOB channel has been enabled by the host, and clears all the neccessary hardware bits.

Inputs

None

Outputs

0(OOB Channel is disabled), 1(OOB Channel is enabled)

### espi\_ oob\_rx\_is\_done

Function Header

**uint32\_t espi\_oob\_rx\_is\_done( void )**

Description

This function does the following –

* checks the status of the RECEIVE\_ENABLE bit in the OOB Receive Status register and returns
  + ‘0b’, if RECEIVE\_ENABLE == 1
  + Current contents of the status bits, if RECEIVE\_ENABLE == 0
* Clears all the necessary status bits in the hardware

Inputs

None

Outputs

0(OOB Rx transfer ongoing), non-zero(OOB Rx transfer complete)

### espi\_ oob\_tx\_is\_done

Function Header

**uint32\_t espi\_oob\_tx\_is\_done( void )**

Description

This function does the following –

* checks the status of the TRANSMIT\_BUSY bit in the OOB Transmit Status register and returns
  + ‘0b’, if TRANSMIT\_BUSY == 1
  + Current contents of the status bits, if TRANSMIT\_BUSY == 0
* Clears all the necessary status bits in the hardware

Inputs

None

Outputs

0(OOB Tx transfer ongoing), non-zero(OOB Tx transfer complete)

## eSPI OOB Channel Peripheral Functions

The list eSPI OOB Channel Peripheral functions are -

* p\_espi\_oob\_rx\_addr\_reg\_write
* p\_espi\_oob\_tx\_addr\_reg\_write
* p\_espi\_oob\_rx\_length\_reg\_write
* p\_espi\_oob\_rx\_length\_reg\_read
* p\_espi\_oob\_tx\_length\_reg\_write
* p\_espi\_oob\_rx\_start
* p\_espi\_oob\_rx\_control\_reg\_read
* p\_espi\_oob\_rx\_interrupt\_enable\_reg\_write
* p\_espi\_oob\_rx\_status\_reg\_get\_clr
* p\_espi\_oob\_tx\_control\_reg\_write
* p\_espi\_oob\_tx\_start
* p\_espi\_oob\_tx\_control\_reg\_read
* p\_espi\_oob\_tx\_interrupt\_enable\_reg\_write
* p\_espi\_oob\_tx\_status\_reg\_get\_clr

### p\_espi\_oob\_rx\_addr\_reg\_write

Function Header

**void p\_espi\_oob\_rx\_addr\_reg\_write( uint32\_t rx\_buff\_addr )**

Description

configures the base address of the OOB buffer in SRAM to receive the next packet.

Note – Do not use this function if RECEIVE\_ENABLE == 1 and RECEIVE\_DONE\_STATUS == 0.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****rx\_buff\_addr**** | DWORD aligned address value |

Outputs

None

### p\_espi\_oob\_tx\_addr\_reg\_write

Function Header

**void p\_espi\_oob\_tx\_addr\_reg\_write( uint32\_t tx\_buff\_addr )**

Description

configures the base address of the OOB transmit buffer in SRAM that contains the next packet.

Note – Do not use this function if TRANSMIT\_BUSY == 1.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****tx\_buff\_addr**** | DWORD aligned address value |

Outputs

None

### p\_espi\_oob\_rx\_length\_reg\_write

Function Header

**void p\_espi\_oob\_rx\_length\_reg\_write( uint32\_t rx\_buff\_len )**

Description

writes to the OOB Receive Length register.

Note – Do not use this function if RECEIVE\_ENABLE == 1 and RECEIVE\_DONE\_STATUS == 0.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****rx\_buff\_len**** | 13 bit value for the buffer length |

Outputs

None

### p\_espi\_oob\_rx\_length\_reg\_read

Function Header

**uint32\_t p\_espi\_oob\_rx\_length\_reg\_read( void )**

Description

reads the oob receive length register.

Note – Do not use this function if RECEIVE\_ENABLE == 1 and RECEIVE\_DONE\_STATUS == 0.

Inputs

None

Outputs

Current contents of the register.

### p\_espi\_oob\_tx\_length\_reg\_write

Function Header

**void p\_espi\_oob\_tx\_length\_reg\_write( uint16\_t tx\_buff\_len )**

Description

configures the length for the oob transmit buffer.

Note – Do not use this function if TRANSMIT\_BUSY == 1.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****tx\_buff\_len**** | 13 bit value for the buffer length |

Outputs

None

### p\_espi\_oob\_rx\_start

Function Header

**void p\_espi\_oob\_rx\_start( void )**

Description

writes to the SET\_RECEIVE\_AVAILABLE bit of the OOB Receive Control register.

Note – OOB Channel Rx Address register and OOB Channel Rx Length register must be

initialized before using this function.

Inputs

None

Outputs

None

### p\_espi\_oob\_rx\_control\_reg\_read

Function Header

**uint32\_t p\_espi\_oob\_rx\_control\_reg\_read( void )**

Description

reads the OOB Receive Control register.

Inputs

None

Outputs

Current contents of the register.

### p\_espi\_oob\_rx\_interrupt\_enable\_reg\_write

Function Header

**void p\_espi\_oob\_rx\_interrupt\_enable\_reg\_write( uint32\_t new\_val )**

Description

writes to the OOB Receive Interrupt Enable register.

Note – By default, 'new\_val' should be set to ESPI\_OOB\_RX\_INT\_DISABLED.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New configuration value (Refer to the header file for corresponding macros) |

Outputs

None

### p\_espi\_oob\_rx\_status\_reg\_get\_clr

Function Header

**uint32\_t p\_espi\_oob\_rx\_status\_reg\_get\_clr( void )**

Description

reads the OOB Receive Status register and clears any of the status bits, if set.

Inputs

None

Outputs

Current contents of the register.

### p\_espi\_oob\_tx\_control\_reg\_write

Function Header

**void p\_espi\_oob\_tx\_control\_reg\_write( uint32\_t new\_val )**

Description

writes to the OOB Transmit Control register.

Note –

1. Do not use this function if TRANSMIT\_BUSY == 1.
2. This function does not write to the TRANSMIT\_START bit.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New configuration value (Refer to the header file for corresponding macros) |

Outputs

None

### p\_espi\_oob\_tx\_start

Function Header

**void p\_espi\_oob\_tx\_start( void )**

Description

writes to the TRANSMIT\_START bit of the OOB Transmit Control register.

Note – OOB Channel Tx Address register and OOB Channel Tx Length register must be

initialized before using this function.

Inputs

None

Outputs

None

### p\_espi\_oob\_tx\_control\_reg\_read

Function Header

**uint32\_t p\_espi\_oob\_tx\_control\_reg\_read( void )**

Description

reads the OOB Transmit Control register.

Inputs

None

Outputs

Current contents of the register.

### p\_espi\_oob\_tx\_interrupt\_enable\_reg\_write

Function Header

**void p\_espi\_oob\_tx\_interrupt\_enable\_reg\_write( uint32\_t new\_val )**

Description

writes to the OOB Transmit Interrupt Enable register.

Note – By default, 'new\_val' should be set to ESPI\_OOB\_RX\_INT\_DISABLED.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New configuration value (Refer to the header file for corresponding macros) |

Outputs

None

### p\_espi\_oob\_tx\_status\_reg\_get\_clr

Function Header

**uint32\_t p\_espi\_oob\_tx\_status\_reg\_get\_clr( void )**

Description

reads the OOB Transmit Status register and clears any of the status bits, if set.

Inputs

None

Outputs

Current contents of the register.

eSPI

instances

eSPI 0

eSPI Flash Channel API’s

espi\_flash\_ch\_is\_enabled

espi\_flash\_ch\_tx\_is\_done

eSPI Flash Channel Peripheral Functions

p\_espi\_flash\_ch\_addr\_reg\_write

p\_espi\_flash\_ch\_buffer\_reg\_write

p\_espi\_flash\_ch\_transfer\_len\_reg\_write

p\_espi\_flash\_ch\_ctrl\_reg\_write

p\_espi\_flash\_ch\_ctrl\_reg\_read

p\_espi\_flash\_ch\_tx\_start

p\_espi\_flash\_ch\_tx\_abort

p\_espi\_flash\_ch\_int\_en\_reg\_write

p\_espi\_flash\_ch\_config\_reg\_read

p\_espi\_flash\_ch\_status\_reg\_get\_clr

eSPI Flash Channel registers

Flash Channel Flash Access Register

Flash Channel Buffer Address Register

Flash Channel Transfer Length Register

Flash Channel Control Register

Flash Channel Interrupt Enable Register

Flash Channel Configuration Register

Flash Channel Status Register

Interrupt Peripheral Functions

PCR Peripheral Functions

## eSPI Flash Channel API Functions

The list eSPI Flash Channel API functions are -

* espi\_flash\_ch\_is\_enabled
* espi\_flash\_ch\_tx\_is\_done

### espi\_ flash\_ch\_is\_enabled

Function Header

**uint8\_t espi\_flash\_ch\_is\_enabled( void )**

Description

checks if the Flash channel has been enabled by the host, and clears all the neccessary hardware bits.

Inputs

None

Outputs

0(Flash Channel is disabled), 1(Flash Channel is enabled).

### espi\_ flash\_ch\_tx\_is\_done

Function Header

**uint32\_t espi\_flash\_ch\_tx\_is\_done( void )**

Description

This function does the following –

* checks the status of the BUSY bit in the Flash Configuration register and returns
  + ‘0b’, if BUSY == 1
  + current contents of the status register, if BUSY == 0
* Clears all the necessary status bits in the hardware

Inputs

None

Outputs

0(Flash channel transfer ongoing), non-zero(Flash channel transfer complete).

## eSPI Flash Channel Peripheral Functions

The list eSPI Flash Channel Peripheral functions are -

* p\_espi\_flash\_ch\_addr\_reg\_write
* p\_espi\_flash\_ch\_buffer\_reg\_write
* p\_espi\_flash\_ch\_transfer\_len\_reg\_write
* p\_espi\_flash\_ch\_ctrl\_reg\_write
* p\_espi\_flash\_ch\_ctrl\_reg\_read
* p\_espi\_flash\_ch\_tx\_start
* p\_espi\_flash\_ch\_tx\_abort
* p\_espi\_flash\_ch\_int\_en\_reg\_write
* p\_espi\_flash\_ch\_config\_reg\_read
* p\_espi\_flash\_ch\_status\_reg\_get\_clr

### p \_espi\_flash\_ch\_addr\_reg\_write

Function Header

**void p\_espi\_flash\_ch\_addr\_reg\_write( uint32\_t new\_val )**

Description

configures the address value to be accessed in external flash by eSPI.

Note – Do not use this function if BUSY == 1.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | new address value |

Outputs

None

### p \_espi\_flash\_ch\_buffer\_reg\_write

Function Header

**void p\_espi\_flash\_ch\_buffer\_reg\_write( uint32\_t new\_val )**

Description

configures the base address for the flash channel data buffer in the SRAM.

Note – Do not use this function if BUSY == 1.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | new address value |

Outputs

None

### p\_espi\_flash\_ch\_transfer\_len\_reg\_write

Function Header

**void p\_espi\_flash\_ch\_transfer\_len\_reg\_write( uint32\_t new\_val )**

Description

configures the total number of bytes to be transferred in the transaction sequence.

Note – Do not use this function if BUSY == 1.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | new count value |

Outputs

None

### p\_espi\_flash\_ch\_ctrl\_reg\_write

Function Header

**void p\_espi\_flash\_ch\_ctrl\_reg\_write( uint32\_t new\_val )**

Description

writes to the Flash Channel Control register.

Note –

1. Do not use this function if BUSY == 1.
2. This function does not write to ABORT\_ACCESS and FLASH\_START bits.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New configuration value (Refer to the header file for corresponding macros) |

Outputs

None

### p\_espi\_flash\_ch\_ctrl\_reg\_read

Function Header

**uint32\_t p\_espi\_flash\_ch\_ctrl\_reg\_read( void )**

Description

reads the Flash Channel Control register.

Inputs

None

Outputs

Current contents of the register.

### p\_espi\_flash\_ch\_tx\_start

Function Header

**void p\_espi\_flash\_ch\_tx\_start( void )**

Description

writes a '1b' to the FLASH\_START bit in Flash Channel Control register.

Note – Do not use this function if BUSY == 1.

Inputs

None

Outputs

None

### p\_espi\_flash\_ch\_tx\_abort

Function Header

**void p\_espi\_flash\_ch\_tx\_abort( void )**

Description

writes a '1b' to the ABORT\_ACCESS bit in Flash Channel Control register.

Note – Use this function ONLY when BUSY == 1. Calling this function when BUSY != 1 has no

effect.

Inputs

None

Outputs

None

### p\_espi\_flash\_ch\_int\_en\_reg\_write

Function Header

**void p\_espi\_flash\_ch\_int\_en\_reg\_write( uint32\_t new\_val )**

Description

writes to the Flash Channel Interrupt Enable register.

Note – By default, new\_val should be set to ESPI\_FLASH\_ALL\_INT\_DISABLED.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****new\_val**** | New configuration value (Refer to the header file for corresponding macros) |

Outputs

None

### p\_espi\_flash\_ch\_config\_reg\_read

Function Header

**void p\_espi\_flash\_ch\_tx\_abort( void )**

Description

reads the Flash Channel Configuration register.

Inputs

None

Outputs

Current contents of the register.

### p \_espi\_flash\_ch\_status\_reg\_get\_clr

Function Header

**uint32\_t p\_espi\_flash\_ch\_status\_reg\_get\_clr( void )**

Description

reads the Flash Channel Status register and clears any of the status bits, if set.

Inputs

None

Outputs

Current contents of the register.

eSPI

instances

eSPI 0

eSPI VWire Channel registers

IRQ Selection Format Register

eSPI VWire Errors Register

VWire Status Register

MSVW Register

SMVW Register

eSPI VWire Channel API’s

None

eSPI VWire Channel Peripheral Functions

p\_espi\_vwire\_status\_reg\_read

p\_espi\_vwire\_irq\_sel\_reg\_write

p\_espi\_vwire\_err\_reg\_read

p\_espi\_vwire\_err\_reg\_write

p\_espi\_vwire\_msvw\_configure

p\_espi\_vwire\_msvw\_data\_read

p\_espi\_vwire\_downstream\_traffic\_disable

p\_espi\_vwire\_smvw\_configure

p\_espi\_vwire\_smvw\_data\_write

p\_espi\_vwire\_upstream\_tx\_is\_done

Interrupt Peripheral Functions

## 

PCR Peripheral Functions

## eSPI VWire Channel Peripheral Functions

The list eSPI VWire Channel Peripheral functions are -

* p\_espi\_vwire\_status\_reg\_read
* p\_espi\_vwire\_irq\_sel\_reg\_write
* p\_espi\_vwire\_err\_reg\_read
* p\_espi\_vwire\_err\_reg\_write
* p\_espi\_vwire\_msvw\_configure
* p\_espi\_vwire\_msvw\_data\_read
* p\_espi\_vwire\_downstream\_traffic\_disable
* p\_espi\_vwire\_smvw\_configure
* p\_espi\_vwire\_smvw\_data\_write
* p\_espi\_vwire\_upstream\_tx\_is\_done

### p\_espi\_vwire\_status\_reg\_read

Function Header

**uint32\_t p\_espi\_vwire\_status\_reg\_read( void )**

Description

reads the Virtual Wire Status register.

Inputs

None

Outputs

Current contents of the register.

### p\_espi\_flash\_ch\_int\_en\_reg\_write

Function Header

**void p\_espi\_vwire\_irq\_sel\_reg\_write( VWIRE\_IRQ\_SOURCES source\_id, uint8\_t new\_val )**

Description

writes to the IRQ Selection register of the specified VWire IRQ device source.

Note – By default, new\_val should be set to ESPI\_FLASH\_ALL\_INT\_DISABLED.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****source\_id**** | 0 based vwire IRQ source ID (Refer to the header file for corresponding macros) |
| ****new\_val**** | New configuration value |

Outputs

None

### p\_espi\_vwire\_err\_reg\_read

Function Header

**uint8\_t p\_espi\_vwire\_err\_reg\_read( void )**

Description

reads the VWire Error register.

Inputs

None

Outputs

Current contents of the register.

### p\_espi\_vwire\_err\_reg\_write

Function Header

**void p\_espi\_vwire\_err\_reg\_write( uint8\_t err\_bits\_sel )**

Description

writes to the VWire Error register.

Note – By default, new\_val should be set to ESPI\_FLASH\_ALL\_INT\_DISABLED.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****err\_bits\_sel**** | vwire error status bits to be cleared (Refer to the header file for corresponding macros) |

Outputs

None

### p\_espi\_vwire\_msvw\_configure

Function Header

**void p\_espi\_vwire\_msvw\_configure( MEC2016\_MTOS\_REGS reg\_num,**

**uint8\_t index,**

**uint16\_t reset\_src,**

**uint16\_t reset\_state,**

**uint32\_t src0\_irq\_type,**

**uint32\_t src1\_irq\_type,**

**uint32\_t src2\_irq\_type,**

**uint32\_t src3\_irq\_type )**

Description

writes to the specified Virtual Wire MSVW register.

Note –

1. Setting the index = 1 is illegal.
2. Usage example for 'reset\_state' parameter - Writing a value of 0x05 will denote SRC0 = 0, SRC1 = 1, SRC2 = 0, SRC3 = 1 after reset event.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****reg\_num**** | 0 based register ID (Refer to the header file for corresponding macros) |
| ****reset\_src**** | reset source selection for SRC[3:0] bits (Refer to the header file for corresponding macros) |
| ****reset\_state**** | state of the SRC bits after a reset event (Refer to the header file for corresponding macros) |
| ****src0\_irq\_type**** | interrupt type for SRC0 bit (Refer to the header file for corresponding macros) |
| ****src1\_irq\_type**** | interrupt type for SRC1 bit (Refer to the header file for corresponding macros) |
| ****src2\_irq\_type**** | interrupt type for SRC2 bit (Refer to the header file for corresponding macros) |
| ****src3\_irq\_type**** | interrupt type for SRC3 bit (Refer to the header file for corresponding macros) |

Outputs

None

### p\_ espi\_vwire\_msvw\_data\_read

Function Header

**uint32\_t p\_espi\_vwire\_msvw\_data\_read( MEC2016\_MTOS\_REGS reg\_num )**

Description

writes to the specified Virtual Wire MSVW register.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****reg\_num**** | 0 based register ID (Refer to the header file for corresponding macros) |

Outputs

Current contents of the upper 32 bits of the MSVW register.

### p\_espi\_vwire\_downstream\_traffic\_disable

Function Header

**void p\_espi\_vwire\_downstream\_traffic\_disable( MEC2016\_MTOS\_REGS reg\_num )**

Description

disables all master to slave transactions for the specified virtual wire index.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****reg\_num**** | 0 based register ID (Refer to the header file for corresponding macros) |

Outputs

None

### p\_espi\_vwire\_smvw\_configure

Function Header

**void p\_espi\_vwire\_smvw\_configure( MEC2016\_STOM\_REGS reg\_num,**

**uint8\_t index,**

**uint16\_t reset\_src,**

**uint16\_t reset\_state )**

Description

writes to the specified Virtual Wire SMVW register.

Note - Usage example for 'reset\_state' parameter - Writing a value of 0x05 will denote SRC0 = 0,

SRC1 = 1, SRC2 = 0, SRC3 = 1 after reset event.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****reg\_num**** | 0 based register ID (Refer to the header file for corresponding macros) |
| ****index**** | index value to be set for the specified SMVW register |
| ****reset\_src**** | reset source selection for SRC[3:0] bits (Refer to the header file for corresponding macros) |
| ****reset\_state**** | state of the SRC bits after a reset event (Refer to the header file for corresponding macros) |

Outputs

None

### p\_espi\_vwire\_smvw\_data\_write

Function Header

**void p\_espi\_vwire\_smvw\_data\_write( MEC2016\_STOM\_REGS reg\_num, uint8\_t upstream\_data )**

Description

writes to the SRC[3:0] bits of the specified Virtual Wire SMVW register.

Note - Usage example for 'upstream\_data' parameter - Writing a value of 0x05 will denote

SRC3 = 0, SRC2 = 1, SRC1 = 0, SRC0 = 1.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****reg\_num**** | 0 based register ID (Refer to the header file for corresponding macros) |
| ****upstream\_data**** | 4 bit data to be passed to the master |

Outputs

None

### p\_espi\_vwire\_upstream\_tx\_is\_done

Function Header

**uint8\_t p\_espi\_vwire\_upstream\_tx\_is\_done( MEC2016\_STOM\_REGS reg\_num )**

Description

checks if any of the CHANGE bits are high.

Note - Usage example for 'upstream\_data' parameter - Writing a value of 0x05 will denote

SRC3 = 0, SRC2 = 1, SRC1 = 0, SRC0 = 1.

Inputs

|  |  |
| --- | --- |
| Input Parameter | Description |
| ****reg\_num**** | 0 based register ID (Refer to the header file for corresponding macros) |

Outputs

0(transfer in progress), 1(transfer complete)

# Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Revision | Name | Changes |
| 12/23/2015 | 1.0 | Srinivas | Initial Draft – Integrated from CEC1302 and updated PWM block |
| 12/23/2015 | 2.0 | Swastik | Updated the UART block |
| 12/28/2015 | 3.0 | Hemal | Updated PCR block APIs |
| 1/06/2016 | 4.0 | Srinivas | Updated ADC block |
| 1/07/2016 | 5.0 | Swastik | Updated the LED block |
| 1/7/2016 | 6.0 | Srinivasa | Updated the Tach block |
| 1/11/2016 | 7.0 | Swastik | Updated the GPIO block |
| 1/12/2016 | 8.0 | Swastik | Added new API to the UART block |
| 1/14/2016 | 9.0 | Srinivasa | Did correction in ADC Block after testing. |
| 1/14/2016 | 10.0 | Hemal | Update SMBus block |
| 1/14/2016 | 11.0 | Arun | Added QMSPI section |
| 1/21/2016 | 12.0 | Jay Vasanth | Added ICCT block |
| 1/28/2016 | 13.0 | Swastik | Added PS/2 and Keyboard Scan Matrix block |
| 1/28/2016 | 14.0 | Jay Vasanth | Added week timer block |
| 1/29/2016 | 15.0 | Hemal | Added EMI block |
| 1/29/2016 | 16.0 | Vidya sagar | Added Mailbox block |
| 2/01/2016 | 17.0 | Arun K | Added BClink Module |
| 2/23/2016 | 18.0 | Swastik | Added VBAT Register bank and RPM – PWM blocks |
| 2/27/2016 | 19.0 | Swastik | Added ACPI ECI and 8042 Emulator blocks |
| 2/29/2016 | 20.0 | Swastik | Added Port 80 block |
| 3/2/2016 | 21.0 | Swastik | Added RTOS Timer block |
| 4/29/2015 | 22.0 | Swastik | Added EEPROM and eSPI blocks |
| 1/19/2017 | 23.0 | Swastik | Updated the QMSPI chapter for BootROM A1 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |