# User manual

## DA14580 Peripheral drivers

### UM-B-004

**Abstract**

*This document describes the Peripheral Drivers API of the Software Development Kit of the DA14580. Instructions for the use of the drivers and the complete API reference are provided.*

# Contents

 

# 1 Terms and definitions

| | |
|---|---|
| ADC | Analog to Digital Converter |
| BLE | Bluetooth Low Energy |
| CS | Chip Select |
| EEPROM | Electrically Erasable Programmable Memory |
| GPIO | General Purpose Input Output |
| PWM | Pulse Width Modulation |
| SDK | Software Development Kit |
| SPI | Serial Peripheral Interface |
| UART | Universal Asynchronous Receiver/Transceiver |

# 2 References

1. DA14580, Datasheet, Dialog Semiconductor

# 3 Introduction

The DA14580 supports several peripherals on different interfaces. The DA14580 Software Development Kit provides the following drivers architecture.

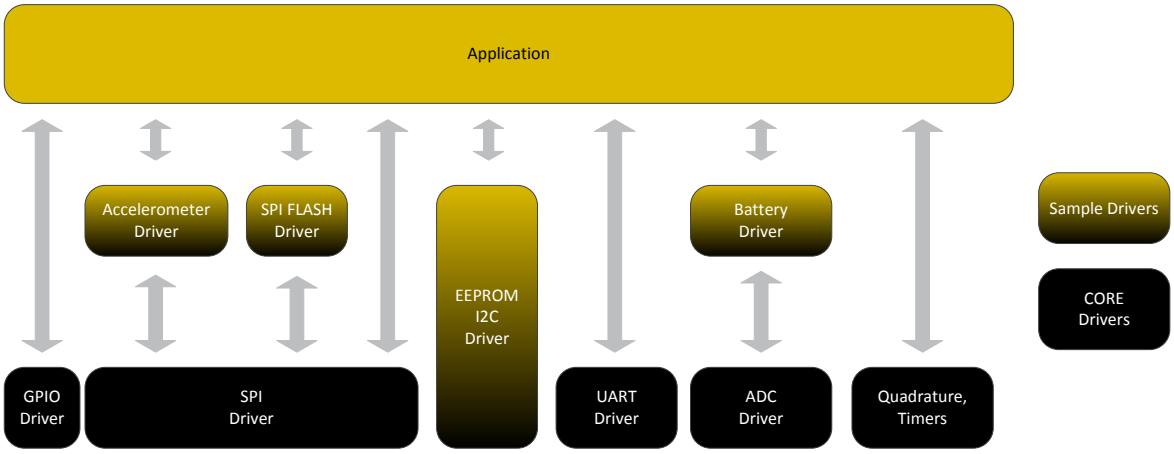


**Figure 1: Peripheral drivers**

The DA14580 SDK comes with a core driver provided for each interface (GPIO, SPI, UART, ADC, Quadrature, Timers) together with several peripheral driver examples (Accelerometer, SPI Flash, EEPROM I2C, battery). Note that even though the source code for all drivers is provided in the SDK as an aid to the debugging, modifying any of the CORE drivers must be avoided.

**Note 1** Upon system wakeup from either extended or deep sleep mode, the device initialization and configuration functions have to be called again. The dedicated location to implement these calls is the periph_init() function in periph_setup.c.

# 4 UART driver

## 4.1 API description

The following section lists the various functions of the UART driver library. This driver is used to provide the necessary abstraction to the applications when access to the UART is required.

This driver is used in the provided BLE applications. You can use the API as is, or compile additional layers to wrap the provided functionality.

**Note 2** The source code for this driver is located in: ble_sw\dk_apps\src\plf\refip\src\driver\uart

### 4.1.1 How to use this driver

- Enable the UART peripheral clock, setting the appropriate bit in CLK_PER_REG[1]
- Initialize the UART, using uart_init()
- Set the RTS signal to Active(low), using uart_flow_on()
- Set the RTS signal to Inactive(high), using uart_flow_off()
- Wait until all transfers are finished, using uart_finish_transfers()
- Read from the UART, using uart_read()
- Write to the UART, using uart_write()

### 4.1.2 Initialization and configuration

- uart_init()
- uart_flow_on()
- uart_flow_off()

- uart_finish_transfers()
- uart_read()
- uart_write()

### 4.1.1 Function reference

#### 4.1.1.1 uart_init

Initializes the UART to default values

| | |
|---|---|
| Function Name | void **uart_init**( uint8_t baud_rate, uint8_t mode) |
| Function Description | Initializes the UART to default values |
| Parameters | baud_rate<br>UART_BAUDRATE_115K2<br>mode<br>e.g. if this parameter is set to 3:<br>{no parity, 1 stop bit, 8 bits data length} settings are applied.<br>Bit7: Set it always to zero<br>Bit6: UART Break Control Bit. Setting this bit to 1, causes a break condition to be transmitted to the receiving device: the serial output is forced to the spacing (logic 0) state.<br>Bit5: Reserved 0x0<br>Bit4: Even Parity Select. This is used to select between even and odd parity, when parity is enabled (PEN set to one). If set to one, an even number of logic 1s is transmitted or checked. If set to zero, an odd number of logic 1s is transmitted or checked.<br>Bit3: Parity Enable. This bit is used to enable and disable parity generation and detection in transmitted and received serial character respectively:<br>0 = parity disabled,<br>1 = parity enabled<br>Bit2: Number of stop bits. This is used to select the number of stop bits per character that the peripheral transmits and receives.<br>0 = 1 stop bit<br>1 = 1.5 stop bits when DLS (LCR[1:0]) is zero, else 2 stop bit<br>Bits1:0 Data Length Select. This is used to select the number of data bits per character that the peripheral transmits and receives:<br>00 = 5 bits<br>01 = 6 bits<br>10 = 7 bits<br>11 = 8 bits |
| Return values | None |
| Notes | |

#### 4.1.1.2 uart_flow_on

Enables the UART RTS signal (active low)

| | |
|---|---|
| Function Name | void **uart_flow_on**(void) |
| Function Description | Enables the UART RTS signal (active low) |
| Parameters | None |
| Return values | None |
| Notes | The RTS pad, if configured, is set to active (low). Please, note that with Auto Flow Control Enabled, the RTS signal is also gated with the receiver FIFO threshold trigger (RTS is inactive-high when above the threshold). |

#### 4.1.1.3 uart_flow_off

Disables the UART RTS signal (active low)

| | |
|---|---|
| Function Name | void **uart_flow_off**(void) |
| Function Description | Disables the UART RTS signal (active low) |
| Parameters | None |
| Return values | None |
| Notes | The RTS pad, if configured, will be driven low (active) |

#### 4.1.1.4 uart_finish_transfers

Waits until all UART transfers have finished

| | |
|---|---|
| Function Name | void **uart_finish_transfers**(void); |
| Function Description | Waits until all UART transfers have finished |
| Parameters | None |
| Return values | None |
| Notes | Waits while any of the Transmitter Empty bit and Transmit Holding Register Empty bit of UART_LSR_REG register is set. |

#### 4.1.1.5 uart_write

Writes one or more bytes of data to the UART

| | |
|---|---|
| Function Name | void **uart_write**(uint8_t *bufptr, uint32_t size, void (*callback) (uint8_t)) |
| Function Description | Writes one or more bytes of data to the UART |
| Parameters | bufptr<br>pointer to the buffer<br>size<br>count of bytes to send<br>callback<br>set to NULL<br>e.g. uart_write( buffer1[], 12, NULL);<br>will send 12 bytes from buffer1 to the UART |
| Return values | None |
| Notes | |

#### 4.1.1.6 uart_read

Reads one or more bytes of data from the UART

| | |
|---|---|
| Function Name | void **uart_read**(uint8_t *bufptr, uint32_t size, void (*callback) (uint8_t)) |
| Function Description | Reads one or more bytes of data from the UART |
| Parameters | bufptr<br>pointer to the buffer<br>size<br>count of bytes to read<br>callback<br>set to NULL<br>e.g. uart_write( buffer1[], 12, NULL);<br>will read 12 bytes from the UART to buffer1 |
| Return values | None |
| Notes | |

## 4.2 Definitions

```
#define UART_BAUDRATE_115K2         9
/// Baudrate used on the UART
#ifndef CFG_ROM
#define UART_BAUDRATE  UART_BAUDRATE_115K2
#else //CFG_ROM
#define UART_BAUDRATE  UART_BAUDRATE_460K8
#endif //CFG_ROM

#if (UART_BAUDRATE == UART_BAUDRATE_921K6)
#define UART_CHAR_DURATION        11
#else
#define UART_CHAR_DURATION       (UART_BAUDRATE * 22)
#endif // (UART_BAUDRATE == UART_BAUDRATE_921K6)

/// Generic enable/disable enum for UART driver
enum
{
    /// uart disable
    UART_DISABLE = 0,
    /// uart enable
    UART_ENABLE  = 1
};

/// Character format
enum
{
    /// char format 5
    UART_CHARFORMAT_5 = 0,
    /// char format 6
    UART_CHARFORMAT_6 = 1,
    /// char format 7
    UART_CHARFORMAT_7 = 2,
    /// char format 8
    UART_CHARFORMAT_8 = 3
};

/// Stop bit
enum
{
```

```
    /// stop bit 1
    UART_STOPBITS_1 = 0,
    /* Note: The number of stop bits is 1.5 if a
     * character format with 5 bit is chosen */
    /// stop bit 2
    UART_STOPBITS_2 = 1
};

/// Parity bit
enum
{
    /// even parity
    UART_PARITYBIT_EVEN  = 0,
    /// odd parity
    UART_PARITYBIT_ODD   = 1,
    /// space parity
    UART_PARITYBIT_SPACE = 2, // The parity bit is always 0.
    /// mark parity
    UART_PARITYBIT_MARK  = 3  // The parity bit is always 1.
};

/* Error detection */
enum
{
    /// error detection disabled
    UART_ERROR_DETECT_DISABLED = 0,
    /// error detection enabled
    UART_ERROR_DETECT_ENABLED  = 1
};

/// status values
enum
{
    /// status ok
    UART_STATUS_OK,
    /// status not ok
    UART_STATUS_ERROR
};
```

# 5 GPIO driver

## 5.1 API description

The following section lists the various functions of the GPIO driver library. This driver is used to provide the necessary abstraction to the applications when access to the GPIOs is required. Furthermore, it guarantees that each GPIO is used only from one module (or place) at a time.

For the monitoring of the GPIO assignment, the assumption is made that this functionality is required only during Development phase. Thus, any variables used for this purpose are not required in the final version that will be burned in the OTP and no valuable memory space will be consumed.

Based on the above, a 64-bit variable is used for the monitoring of the GPIO assignment. The first 16-bits of this variable are assigned to port 0, the next 16-bits to port 1 and so on. Each bit represents one GPIO pin of a port. This variable is placed at the retention memory and preserved during deep sleep.

Each module that needs to use a GPIO pin must first reserve it. The reservations are made inside the source file periph_setup.h (function GPIO_reservations()) using the macro RESERVE_GPIO(). This macro is defined in gpio.h as:

```
#define RESERVE_GPIO( name, port, pin, func )    /
```

```
{ GPIO[##port##][##pin##] = (GPIO[##port##][##pin##] != 0) ? (-1) : 1; };
```

The parameters ‘name’ and ‘func’ are used only to provide a readable declaration. This macro will set a member of the GPIO[] array (that corresponds to this GPIO pin) to 1, if free, or -1, if it has been already reserved.

Upon initialization, the function GPIO_init() (gpio.c) is called. This function will first check for multiple reservations of the same GPIO pin (halting to a breakpoint if one is found) and then set the 64-bit GPIO_status variable, according to the reservations that have been made in the gpio_pindefs.h.

This variable will then be checked at the entry of any of the API functions to find out if the GPIO that the function is being called for has been previously reserved. If it was not reserved then a breakpoint will be asserted.

The functionality described so far is available only when the DEVELOPMENT__NO_OTP flag is set. Since breakpoints result to HardFault interrupts when no debugger is attached, the code snippet MUST be left out in the final version by setting DEVELOPMENT__NO_OTP to 0.

Of course, direct access to GPIOs without using this API **must be avoided**. There is no way to prevent such coding approach but one should have in mind that this way any visibility offered by this driver will be lost and there will be no guarantee that the same GPIO is not used in more than one places and, possibly, for not the same purpose.

**Note 3** The source code for this driver is located in: ble_sw\dk_apps\src\plf\refip\src\driver\gpio

### 5.1.1 How to use this driver

Typical use

- Populate GPIO_reservations() function in periph_setup.h: Add a RESERVE_GPIO() macro instruction with the proper arguments, for each i/o pin you wish to use.
- Populate set_pad_functions() function in periph_setup.h: Add a call to GPIO_ConfigurePin() function with the proper arguments, for each i/o pin you wish to use. After verifying that the pin has been previously reserved, the desired functionality and direction/electrical configuration is setup.
- Set the logical state of a properly configured pin to high, using GPIO_SetActive()
- Set the logical state of a properly configured pin to low, using GPIO_SetInactive()
- Get the logical state of a properly configured pin, using GPIO_GetPinStatus()

Other functionality

- Initialize the GPIO driver, using GPIO_init() (This is configured to be called inherently, upon initialization).
- Set the desired direction of a pin (input/output), its electrical configuration (pullup/pulldown/high-z) and its functionality (GPIO/various peripherals’ pin), using GPIO_SetPinFunction() (this is configured to be called inherently from within GPIO_ConfigurePin().

### 5.1.2 Initialization and configuration

- GPIO_init()
- GPIO_SetPinFunction()
- GPIO_ConfigurePin()
- GPIO_SetActive()
- GPIO_SetInactive()
- GPIO_GetPinStatus()

### 5.1.3 Functions reference

#### 5.1.3.1 GPIO_init

Checks for multiple reservations of the same GPIO pin. Initializes the GPIO_status variable. Called at system startup.

| | |
|---|---|
| Function Name | void **GPIO_init**(void) |
| Function Description | Checks for multiple reservations of the same GPIO pin. Initializes the GPIO_status variable. Called at system startup. |
| Parameters | None |
| Return values | None – breakpoint is triggered in case of duplicate assignment of a pin. |
| Notes | Active only during development (DEVELOPMENT__NO_OTP = 1). Deactivate for release, to preserve memory space. |

#### 5.1.3.2 GPIO_SetPinFunction

Sets the pin type (input, input pull-up or pull-down, output) and the pin function (GPIO, UART1_RX, etc.).

| | |
|---|---|
| Function Name | void **GPIO_SetPinFunction**( int port, int pin, GPIO_PUPD mode, GPIO_FUNCTION function ) |
| Function Description | Sets the pin type (input, input pull-up or pull-down, output) and the pin function (GPIO, UART1_RX, etc.). |
| Parameters | port : the GPIO port (GPIO_PORT_n)<br>pin: the GPIO pin (GPIO_PIN_n)<br>mode: the GPIO pin direction/electrical configuration:<br>INPUT : input – high-z<br>INPUT_PULLUP : input with pull-up resistor enabled<br>INPUT_PULLDOWN : input with pull-down resistor enabled<br>OUTPUT : output<br>function: The function of the pin (assignment to internal peripherals):<br>PID_GPIO, PID_UART1_RX, PID_UART1_TX, PID_UART2_RX, PID_UART2_TX, PID_SPI_DI, PID_SPI_DO, PID_SPI_CLK, PID_SPI_EN, PID_I2C_SCL, PID_I2C_SDA, PID_UART1_IRDA_RX, PID_UART1_IRDA_TX, PID_UART2_IRDA_RX, PID_UART2_IRDA_TX, PID_ADC, PID_PWM0, PID_PWM1, PID_BLE_DIAG, PID_UART1_CTSN, PID_UART1_RTSN, PID_UART2_CTSN, PID_UART2_RTSN, PID_PWM2, PID_PWM3, PID_PWM4 |
| Return values | None |
| Notes | PID_ADC is available only on P0_0.. P0_3 |

#### 5.1.3.3 GPIO_ConfigurePin

Combined function to set the pin type and function (like GPIO_SetPinFunction does) and the state of the pin. Can be used for output pins to set them to the desired state first and then configure them as outputs.

| | |
|---|---|
| Function Name | void **GPIO_ConfigurePin**( int port, int pin, GPIO_PUPD mode, GPIO_FUNCTION function, const bool high ); |
| Function Description | Combined function to set the pin type and function (like GPIO_SetPinFunction does) and the state of the pin. Can be used for |

| | |
|---|---|
| | output pins to set them to the desired state first and then configure them as outputs. |
| Parameters | port : the GPIO port (GPIO_PORT_n)<br>pin: the GPIO pin (GPIO_PIN_n)<br>mode: the GPIO pin direction/electrical configuration:<br>INPUT : input – high-z<br>INPUT_PULLUP : input with pull-up resistor enabled<br>INPUT_PULLDOWN : input with pull-down resistor enabled<br>OUTPUT : output<br>function: The function of the pin (assignment to internal peripherals):<br>PID_GPIO, PID_UART1_RX, PID_UART1_TX, PID_UART2_RX, PID_UART2_TX, PID_SPI_DI, PID_SPI_DO, PID_SPI_CLK, PID_SPI_EN, PID_I2C_SCL, PID_I2C_SDA, PID_UART1_IRDA_RX, PID_UART1_IRDA_TX, PID_UART2_IRDA_RX, PID_UART2_IRDA_TX, PID_ADC, PID_PWM0, PID_PWM1, PID_BLE_DIAG, PID_UART1_CTSN, PID_UART1_RTSN, PID_UART2_CTSN, PID_UART2_RTSN, PID_PWM2, PID_PWM3, PID_PWM4<br>high: the desired logical level of the pin. |
| Return values | None |
| Notes | None |

#### 5.1.3.4 GPIO_SetActive

Sets the GPIO as high. The GPIO must have been previously configured as output. No check of the configuration of the pin is made in this function.

| | |
|---|---|
| Function Name | void **GPIO_SetActive**( int port, int pin ) |
| Function Description | Sets the GPIO as high. |
| Parameters | port : the GPIO port (GPIO_PORT_n)<br>pin: the GPIO pin (GPIO_PIN_n) |
| Return values | None |
| Notes | The GPIO must have been previously configured as output. No check of the configuration of the pin is done in this function. |

#### 5.1.3.5 GPIO_SetInactive

Sets the GPIO as low. The GPIO must have been previously configured as output. No check of the configuration of the pin is made in this function.

| | |
|---|---|
| Function Name | void **GPIO_SetInactive**( int port, int pin ) |
| Function Description | Sets the GPIO as low. |
| Parameters | port : the GPIO port (GPIO_PORT_n)<br>pin: the GPIO pin (GPIO_PIN_n) |
| Return values | None |
| Notes | The GPIO must have been previously configured as output. No check of the configuration of the pin is made in this function. |

#### 5.1.3.6 GPIO_GetPinStatus

Gets the status of this GPIO pin. The GPIO must have been previously configured as input. No check of the configuration of the pin is made in this function. The return value is true if the pin is high, else false.

| | |
|---|---|
| Function Name | bool **GPIO_GetPinStatus**( int port, int pin ) |
| Function Description | Gets the status of this GPIO pin.. |
| Parameters | port : the GPIO port (GPIO_PORT_n)<br>pin: the GPIO pin (GPIO_PIN_n) |
| Return values | True if the pin is high, else false. |
| Notes | The GPIO must have been previously configured as input. No check of the configuration of the pin is made in this function. |

## 5.2 Definitions

```
typedef enum {
    INPUT = 0,
    INPUT_PULLUP = 0x100,
    INPUT_PULLDOWN = 0x200,
    OUTPUT = 0x300,
} GPIO_PUPD;

typedef enum {
    GPIO_PORT_0 = 0,
    GPIO_PORT_1 = 1,
    GPIO_PORT_2 = 2,
    GPIO_PORT_3 = 3,
    GPIO_PORT_3_REMAP = 4,
} GPIO_PORT;

typedef enum {
    GPIO_PIN_0 = 0,
    GPIO_PIN_1 = 1,
    GPIO_PIN_2 = 2,
    GPIO_PIN_3 = 3,
    GPIO_PIN_4 = 4,
    GPIO_PIN_5 = 5,
    GPIO_PIN_6 = 6,
    GPIO_PIN_7 = 7,
    GPIO_PIN_8 = 8,
    GPIO_PIN_9 = 9,
} GPIO_PIN;

typedef enum {
    PID_GPIO = 0,
    PID_UART1_RX,
    PID_UART1_TX,
    PID_UART2_RX,
    PI_UART2_TX,
    PID_SPI_DI,
    PID_SPI_DO,
    PID_SPI_CLK,
    PID_SPI_EN,
    PID_I2C_SCL,
    PID_I2C_SDA,
    PID_UART1_IRDA_RX,
    PID_UART1_IRDA_TX,
    PID_UART2_IRDA_RX,
```

```
    PID_UART2_IRDA_TX,
    PID_ADC,
    PID_PWM0,
    PID_PWM1,
    PID_BLE_DIAG,
    PID_UART1_CTSN,
    PID_UART1_RTSN,
    PID_UART2_CTSN,
    PID_UART2_RTSN,
    PID_PWM2,
    PID_PWM3,
    PID_PWM4,
} GPIO_FUNCTION;

//
// Macro for pin definition structure
//      name: usage and/or module using it
//      func: GPIO, UART1_RX, UART1_TX, etc.
//
#if DEVELOPMENT__NO_OTP
#define RESERVE_GPIO( name, port, pin, func )   { GPIO[##port##][##pin##] =
(GPIO[##port##][##pin##] != 0) ? (-1) : 1;GPIO_status |=
((uint64_t)GPIO[##port##][##pin##] << ##pin##) << (##port## * 16);}
#else
#define RESERVE_GPIO( name, port, pin, func )   {}
#endif
```

# 6 Analog-to-Digital Converter (ADC) driver

## 6.1 API description

The following section lists the various functions of the ADC driver library.

**Note 4** The source code for this driver is located in: ble_sw\dk_apps\src\plf\refip\src\driver\adc

### 6.1.1 How to use this driver

- Enable the ADC module and configure it, using adc_init()
- Enable the desired ADC channel, using adc_enable_channel()
- Get a sample from the ADC, using adc_get_sample()
- Upon completion, if desired, disable the ADC, using adc_disable()

### 6.1.2 Initialization and configuration

- adc_init()
- adc_enable_channel()
- adc_get_sample()
- adc_disable()

### 6.1.3 Initialization and configuration functions

#### 6.1.3.1 adc_init

| | |
|---|---|
| Function Name | void **adc_init** (uint16_t mode, uint16_t sign) |
| Function Description | Initializes the ADC peripheral according to the parameters |
| Parameters | mode :<br>0 = Differential mode |

| | |
|---|---|
| | GP_ADC_SE(0x800) = Single ended mode<br>sign :<br>0 = Default, GP_ADC_SIGN(0x400) = Conversion with opposite sign at input and output to cancel out the internal offset of the ADC and low-frequency |
| Return values | None |
| Notes | None |

#### 6.1.3.2 adc_enable_channel

| | |
|---|---|
| Function Name | void **adc_enable_channel** (uint16_t input_selection) |
| Function Description | Enables the ADC Channel specified in the parameter |
| Parameters | input_selection: Input channel. Must pass one of the definitions starting with ADC_CHANNEL_ in adc.h |
| Return values | None |
| Notes | The device must have been initialized, using adc_init() |

#### 6.1.3.3 adc_disable

| | |
|---|---|
| Function Name | void **adc_disable**(void) |
| Function Description | Disables the ADC module |
| Parameters | None |
| Return values | None |
| Notes | None |

### 6.1.4 ADC sampling functions

#### 6.1.4.1 adc_get_sample

The function is used to get a sample from ADC module. The ADC must be initialized and a valid channel must be set.

| | |
|---|---|
| Function Name | int **adc_get_sample**(void) |
| Function Description | Reads an ADC sample |
| Parameters | None |
| Return values | None |
| Notes | None |

### 6.1.5 Definitions

#### ADC_channels

```
#define ADC_CHANNEL_P00          0
#define ADC_CHANNEL_P01          1
#define ADC_CHANNEL_P02          2
#define ADC_CHANNEL_P03          3
#define ADC_CHANNEL_AVS          4
#define ADC_CHANNEL_VDD_REF      5
#define ADC_CHANNEL_VDD_RTT      6
#define ADC_CHANNEL_VBAT3V       7
#define ADC_CHANNEL_VDCDC        8
#define ADC_CHANNEL_VBAT1V       9
```

# 7 Battery level driver

## 7.1 API description

The following section lists the various functions of the BATTERY driver library. These functions support the measurement and translation of the battery level.

**Note 5** The source code for this driver is located in: ble_sw\dk_apps\src\plf\refip\src\driver\battery

### 7.1.1 How to use this driver

- Measure the battery level, using battery_get_lvl().

### 7.1.2 Function reference

#### 7.1.2.1 battery_get_lvl

| | |
|---|---|
| Function Name | uint8_t **battery_get_lvl**(uint8_t batt_type) |
| Function Description | |
| Parameters | batt_type: The code of the battery. It is used for the correct translation of the battery level measurement, based on the battery type's characteristics. |
| Return values | The battery level that is measured. |
| Notes | Stores a copy of the battery level measurement to the retention memory. |

## 7.2 Definitions

```
// Battery types definitions
#define BATT_CR2032                    1          //CR2032 coin cell battery
```

# 8 Serial Peripheral Interface (SPI) driver

## 8.1 API description

The following section lists the various functions of the SPI driver library that handle the initialization, configuration and release of the SPI module, the control of the Chip Select (CS) line and the data transfer over the SPI.

**Note 6** The source code for this driver is located in: ble_sw\dk_apps\src\plf\refip\src\driver\spi

### 8.1.1 How to use this driver

- Enable the SPI block and configure its parameters, using spi_init().
- Activate the chip select line, using spi_set_cs_low().
- Make a sequence of SPI transfers (send and receive data), using spi_access() for each transfer.
- Select the desired SPI bitmode using setSpiBitmode()
- De-activate the chip select line, using spi_set_cs_high().
- For a simple spi transaction (1 read-write cycle of the selected Bitmode) you can use spi_transaction(). The difference to the spi_access() is that the chip select line is also driven inside the function, to form a simple SPI transaction.
- Disable the SPI module, using spi_release().

### 8.1.2 Initialization and configuration

- spi_init ()
- spi_release ()
- setSpiBitmode()
- spi_set_cs_low ()

- spi_set_cs_high ()

#### 8.1.2.1 spi_init

The function to be used for the initialization of the SPI module is spi_init(). It is called with a set of parameters that define the SPI module's operation. When the SPI module is to be configured, it is first disabled, then the status register is updated with the selected parameters, and then the module is enabled again. When the SPI block is disabled, the RX/TX buffers are reset.

| | |
|---|---|
| Function Name | void **spi_init**(SPI_Pad_t *cs_pad_param, SPI_Word_Mode_t bitmode, SPI_Role_t role, SPI_Polarity_Mode_t clk_pol, SPI_PHA_Mode_t pha_mode, SPI_MINT_Mode_t irq, SPI_XTAL_Freq_t freq) |
| Function Description | Initializes the SPI block and configures the driver according to the parameters |
| Parameters | cs_pad_param:<br>port and pin of the Chip Select (CS) pad for the target SPI slave<br>bitmode :<br>SPI_MODE_8BIT = 8-bit mode<br>SPI_MODE_16BIT = 16-bit mode<br>SPI_MODE_32BIT = 32-bit mode<br>SPI_MODE_9BIT = 9-bit mode<br>role:<br>SPI_ROLE_MASTER = Master mode<br>SPI_ROLE_SLAVE = Slave mode<br>clk_pol:<br>SPI_CLK_IDLE_POL_LOW = SPI_CLK is initially low.<br>SPI_CLK_IDLE_POL_HIGH = SPI_CLK is initially high.<br>pha_mode:<br>SPI_PHA_MODE_0<br>SPI_PHA_MODE_1<br>**If phase = polarity (pha_mode = clk_pol), then data is captured on the clock's rising edge, else it is captured on the clock's falling edge, as illustrated in**<br>Table 1: SPI .<br>irq:<br>SPI_MINT_DISABLE = Disable SPI interrupt (SPI_INT_BIT) to ICU.<br>SPI_MINT_ENABLE = Enable SPI interrupt (SPI_INT_BIT) to ICU.<br>Note that the SPI_INT interrupt is shared with AD_INT interrupt<br>freq:<br>Select SPI_CLK clock frequency in master mode:<br>SPI_XTAL_DIV_8 = (XTAL)/ (CLK_PER_REG * 8)<br>SPI_XTAL_DIV_4 = (XTAL) / (CLK_PER_REG * 4)<br>SPI_XTAL_DIV_2 = (XTAL) / (CLK_PER_REG * 2)<br>SPI_XTAL_DIV_14 = (XTAL) / (CLK_PER_REG * 14) |
| Return values | None |
| Notes | None |

**Table 1: SPI modes**

| SPI MODE | CLOCK POLARITY (clk_pol) | CLOCK_PHASE (pha_mode) | Clock edge on which data is sampled |
|---|---|---|---|
| 0 | SPI_CLK_IDLE_POL_LOW | SPI_PHA_MODE_0 | rising edge |
| 1 | SPI_CLK_IDLE_POL_LOW | SPI_PHA_MODE_1 | falling edge |
| 2 | SPI_CLK_IDLE_POL_HIGH | SPI_PHA_MODE_0 | falling edge |
| 3 | SPI_CLK_IDLE_POL_HIGH | SPI_PHA_MODE_1 | rising edge |

#### 8.1.2.2 setSpiBitmode

The setSpiBitmode function is used in order to select the bitmode in which the SPI will operate. The SPI module is first disabled, then the SPI control register (**SPI_CTRL_REG) SPI_WORD** is updated to set the selected bitmode and then the module is enabled again. When the SPI block is disabled, the RX/TX buffers are reset.

| | |
|---|---|
| Function Name | void **setSpiBitmode** (SPI_Word_Mode_t spiBitMode) |
| Function Description | selects the SPI bitmode |
| Parameters | spiBitMode<br>SPI_MODE_8BIT = 8-bit mode<br>SPI_MODE_16BIT = 16-bit mode<br>SPI_MODE_32BIT = 32-bit mode<br>SPI_MODE_9BIT = 9-bit mode |
| Return values | None |
| Notes | None |

#### 8.1.2.3 spi_release

The spi_release function is used in order to disable the SPI module. It resets the SPI_ON bit of the SPI Control Register (SPI_CTRL_REG0) and resets the SPI_ENABLE bit of the Peripheral divider register (CLK_PER_REG).

| | |
|---|---|
| Function Name | void **spi_release**(void) |
| Function Description | Disables the SPI block |
| Parameters | None |
| Return values | None |
| Notes | None |

### 8.1.3 Sending and receiving

#### 8.1.3.1 spi_access

The spi_access function performs a data transfer over the SPI interface. The state of the CS line is not altered by this function, so it can be called multiple times in conjuction with spi_cs_low and spi_cs_high to form a complex spi transaction. For a complete simple SPI transaction, see 8.1.3.2 below.

Prior to a transfer, the SPI module has to be initialized using spi_init.

The spi_access function starts by extracting the selected word mode for the current SPI configuration. Then, it writes the SPI Rx/Tx register(s) (SPI_RX_TX_REG0 in any case and SPI_RX_TX_REG1 in the cases of 32-bit and 9-bit word modes). Consequently, the function polls the SPI Control Register, waiting for the transfer completion. Upon the completion of the transfer (SPI Control Register's interrupt bit, SPI_INT_BIT, becomes 1), the function reads the received data from the SPI Rx/Tx register(s), clears the interrupt bit, and returns the received data.

| | |
|---|---|
| Function Name | uint32_t **spi_access**(uint32_t dataToSend) |
| Function Description | Writes dataToSend to SPI and reads the received value |
| Parameters | dataToSend: data to be written |
| Return values | The received data |
| Notes | The function reads the value of the status register to determine the word mode (8/16/32/9-bit) of the configuration |

#### 8.1.3.2 spi_transaction

The spi_transaction function performs a complete SPI transaction over the SPI interface (data write and read, driving the CS line to signal the start and the end of the transaction). Prior to a transaction, the SPI module has to be initialized using spi_init. For a simple SPI data transfer see 8.1.3.18.1.3.1 above.

The spi_transaction function first sets CS line low, calls spi_access to perform the data transfer and finally sets cs high.

| | |
|---|---|
| Function Name | uint32_t **spi_transaction**(uint32_t dataToSend) |
| Function Description | Writes dataToSend to SPI in a simple full transaction and reads received value |
| Parameters | dataToSend: the data to be written |
| Return values | Received data |
| Notes | See also: spi_access (called by spi_transaction) |

#### 8.1.3.3 spi_cs_low

The spi_cs_low function is used to activate the SPI Chip Select (/CS – active low) line.

Prior to using this function, the SPI module has to be initialized using spi_init.

| | |
|---|---|
| Function Name | inline void **spi_cs_low**(void) |
| Function Description | Sets the chip select line (CS) to low. Used to signal the beginning of a SPI transaction. |
| Parameters | None |
| Return values | None |
| Notes | uses the spi_driver_cs_pad structure which is initialized with the cs port and pin numbers in the spi_init function. |

#### 8.1.3.4 spi_cs_high

The spi_cs_high function is used to deactivate the SPI Chip Select (/CS – active low) line.

Prior to using this function, the SPI module has to be initialized using spi_init.

| | |
|---|---|
| Function Name | inline void **spi_cs_high**(void) |
| Function Description | Sets the chip select line (CS) to high. Used to signal the end of a SPI transaction. |
| Parameters | None |
| Return values | None |
| Notes | Uses spi_driver_cs_pad structure which is initialized with the cs port and the pin |

| | |
|---|---|
| | numbers in the spi_init function. |

### 8.1.4 Definitions

#### SPI block configuration

```
typedef enum SPI_WORD_MODES{
       SPI_MODE_8BIT,
       SPI_MODE_16BIT,
       SPI_MODE_32BIT,
       SPI_MODE_9BIT,
}SPI_Word_Mode_t;

typedef enum SPI_ROLES{
       SPI_ROLE_MASTER,
       SPI_ROLE_SLAVE,
}SPI_Role_t;

typedef enum SPI_POL_MODES{
       SPI_CLK_IDLE_POL_LOW,
       SPI_CLK_IDLE_POL_HIGH,
}SPI_Polarity_Mode_t;

typedef enum SPI_PHA_MODES{
       SPI_PHA_MODE_0,
       SPI_PHA_MODE_1,
}SPI_PHA_Mode_t;

typedef enum SPI_MINT_MODES{
       SPI_MINT_DISABLE,
       SPI_MINT_ENABLE,
}SPI_MINT_Mode_t;

typedef enum SPI_FREQ_MODES{
  SPI_XTAL_DIV_8,
  SPI_XTAL_DIV_4,
       SPI_XTAL_DIV_2,
       SPI_XTAL_DIV_14,
}SPI_XTAL_Freq_t;

typedef struct
{
       GPIO_PORT port;
       GPIO_PIN pin;
}SPI_Pad_t;
```

# 9 SPI flash driver

## 9.1 API description

The following section lists the various functions of the SPI flash driver library. These functions implement the various instructions of an SPI Flash (e.g. Winbond W25x10). This driver uses the basic SPI access functions included in the SPI driver libraries (spi.c).

**Note 7** The source code for this driver is located in: ble_sw\dk_apps\src\plf\refip\src\driver\spi_flash

### 9.1.1 How to use this driver

- Initialize the FLASH memory using spi_flash_init().

Ensure that the SPI driver has been also initialized and that the application has configured the corresponding pads.

- Controlling the write access:

  Set the Write Enable Latch (WEL) bit of the status register, using spi_flash_set_write_enable ().

  Enable the write operation for the volatile bits of the status register, using spi_flash_write_enable_volatile ().

  Reset the Write Enable Latch (WEL) bit of the status register, using spi_flash_write_disable().

- Status Register

  Read the SPI Flash status register, using spi_flash_read_status_reg().

  Write the SPI Flash status register, using spi_flash_write_status_reg().

- Read/write data

  Read data from the SPI Flash, using spi_flash_read_data().

  Program a page of the SPI Flash, using spi_flash_page_program().

  Erase either a sector(4 KB), a 32KB block or a 64KB block of the SPI Flash, using spi_flash_erase().

  Erase all data in the SPI Flash, using spi_flash_chip_erase().

  Write any amount of data to the SPI Flash, using spi_flash_write_data().

  Read the SPI Flash's Manufacturer/Device ID, using spi_read_flash_memory_man_and_dev_id().

  Read the SPI Flash's Unique ID Number, using spi_read_flash_unique_id().

  Read the SPI Flash's JEDEC ID, using spi_read_flash_jedec_id().

### 9.1.2 Initialization and configuration

- spi_flash_set_write_enable ()
- spi_flash_write_enable_volatile ()
- spi_flash_write_disable ()
- spi_flash_read_status_reg ()
- spi_flash_write_status_reg ()

#### 9.1.2.1 Read from the flash

- spi_flash_read_data ()

#### 9.1.2.2 Write to the flash

- spi_flash_page_program()
- spi_flash_write_data()
- spi_flash_page_fill()
- spi_flash_fill()

#### 9.1.2.3 Erase the flash

- spi_flash_block_erase()
- spi_flash_chip_erase()
- spi_flash_chip_erase_forced()

#### 9.1.2.4 Power management

- spi_flash_power_down()
- spi_flash_release_from_power_down()

#### 9.1.2.5 Data protection

- spi_flash_configure_memory_protection()

#### 9.1.2.6 Miscellaneous

- spi_read_flash_memory_man_and_dev_id ()
- spi_read_flash_unique_id ()
- spi_read_flash_jedec_id ()

### 9.1.3 Initialization and configuration functions

#### 9.1.3.1 spi_flash_set_write_enable

The Write Enable instruction sets the Write Enable bit (WEL) in the SPI Flash Status Register. The WEL bit must be set prior to every write or erase instruction. This function drives /CS low, writes the Write Enable instruction code and then drives the /CS high. Before returning, the function polls the SPI Flash Status Register to ensure that the WEL bit has been set.

| | |
|---|---|
| Function Name | int8_t **spi_flash_set_write_enable**(void) |
| Function Description | Sets the Write Enable bit (WEL) in the Status Register |
| Parameters | None |
| Return values | ERR_OK , ERR_TIMEOUT |
| Notes | The WEL bit must be set prior to every Page Program, Sector Erase, Block Erase, Chip Erase and Write Status Register instruction. This provision is embedded in the user functions provided by the driver. Before any write operations take place, the corresponding functions call spi_flash_set_write_enable(). In case write access cannot be achieved, timeout occurs. |

#### 9.1.3.2 spi_flash_write_enable_volatile

This instruction is used prior to a Write Status Register instruction in order for the non-volatile Status Register bits to be written as volatile bits. This instruction does not set the WEL bit, it is only valid for the Write Status Register instruction to change the volatile Status Register bit values.

| | |
|---|---|
| Function Name | void **spi_flash_write_enable_volatile**(void) |
| Function Description | Issued prior to a Write Status Register instruction in order to write the non-volatile Status Register bits. |
| Parameters | None |
| Return values | None |
| Notes | Will not set the Write Enable Latch (WEL) bit. |

#### 9.1.3.3 spi_flash_write_disable

The Write Disable instruction resets the Write Enable bit (WEL) in the SPI Flash Status Register. This function drives /CS low, writes the Write Disable instruction code and then drives the /CS high. Before returning, the function polls the SPI Flash Status Register to ensure that the WEL bit has been reset. The WEL bit is automatically reset after power-up and upon completion of the Write Status Register, Page Program, Sector Erase, Block Erase and Chip Erase instructions. The Write Disable instruction can be used to invalidate the Write Enable for Volatile Status Register instruction.

| | |
|---|---|
| Function Name | void **spi_flash_write_disable**(void) |
| Function Description | Resets the Write Enable bit (WEL) in Status Register |
| Parameters | None |
| Return values | None |

| | |
|---|---|
| Notes | None |

#### 9.1.3.4 spi_flash_read_status_reg

The Read Status Register instruction is used to read the value of the SPI Flash Status Register. This function drives /CS low, writes the Read Status Register instruction code, makes another SPI access to get the Status Register's value and then drives the /CS high. This instruction may be used while a Program, Erase or Write Status Register cycle is in progress. This allows the BUSY status bit to be checked to determine when a cycle is complete and if the device can accept another instruction. The Status Register can be read continuously.

| | |
|---|---|
| Function Name | uint8_t **spi_flash_read_status_reg**(void) |
| Function Description | Reads the value of the SPI flash status register. |
| Parameters | None |
| Return values | The SPI flash status register value |
| Notes | The Read Status Register instruction may be used at any time, even while a Program, Erase or Write Status Register cycle is in progress. This allows the BUSY status bit to be checked to determine when the cycle is complete and if the device can accept another instruction. The Status Register can be read continuously. The instruction is completed by driving /CS high. |

#### 9.1.3.5 spi_flash_write_status_reg

The Write Status Register instruction can be used to write the non-volatile bits of the Status Register. A Write Enable instruction must have been previously executed for the device to accept the Write Status Register instruction. This function issues a Write Enable instruction, then writes the Write Status Register instruction code and finally inputs the value to be written to the Status Register.

| | |
|---|---|
| Function Name | void **spi_flash_write_status_reg**(uint8_t status) |
| Function Description | Writes a 8-bit value to the SPI flash status register. |
| Parameters | Status: 8-bit value to be written to the status register. |
| Return values | None |
| Notes | A Write Enable instruction must have been previously executed for the device to accept the Write Status Register Instruction. Only non-volatile Status Register bits (7, 5, 3 and 2) can be written to. |

### 9.1.4 Read flash instructions

#### 9.1.4.1 spi_flash_read_data

The Read Data instruction is used to read a data block of the SPI Flash. The function drives /CS low, writes the Read Data instruction code, then writes the starting address of the data block (24-bit) and then keeps the /CS low until the last element of the data block is read. If the given size of the data block to be read exceeds the available memory size after the given address, the Read Data function will only read the available data.

| | |
|---|---|
| Function Name | uint32_t **spi_flash_read_data**(uint8_t *rd_data_ptr, uint32_t address, uint32_t size) |
| Function Description | Reads data from a starting address of the SPI flash, equal to ‹size› bytes. |
| Parameters | *rd_data_ptr:<br>Pointer to the memory position where the read data will be stored.<br>address:<br>Address of the first element to be read.<br>size:<br>Size of the data block to be read. |

| | |
|---|---|
| Return values | Bytes actually read or ERR_TIMEOUT in cased of failure. |
| Notes | If the size passed as a parameter exceeds the flash's size after the given address, Read Data will read only this available size. |

### 9.1.5 Write flash instructions

#### 9.1.5.1 spi_flash_page_program

The Page Program instruction allows data to be programmed at previously erased (FFh) memory locations in a single memory page. If an entire page has to be programmed, the address should be a multiple of page size. If it is not, the addressing will wrap to the beginning of the page and the respective data will be overwritten. A partial page (fewer bytes than the size of the page) can be programmed without having any effect on other bytes within the same page.

For the Program Page instruction to be completed, the respective function sends a Write Enable instruction and then the Page Program instruction. After the transmission of all data to be written is completed, the function polls the SPI Flash Status Register to determine the completion of the Page Program instruction.

| | |
|---|---|
| Function Name | int32_t **spi_flash_page_program**(uint8_t *wr_data_ptr, uint32_t address, uint16_t size) |
| Function Description | Writes a specified amount of data to a page of the SPI flash (from 1 up to SPI_FLASH_PAGE bytes). |
| Parameters | *wr_data_ptr:<br>Pointer to the memory position where the data to be written reside.<br>address:<br>Address where the first element will be written.<br>size:<br>Size of the data block to be written (1 to 'spi_flash_page_size' (in bytes) as set during initialization). |
| Return values | Bytes actually written or ERR_TIMEOUT in case of failure. |
| Notes | If the size passed as a parameter exceeds the flash page's size, Program Page will write only this available size. |

#### 9.1.5.2 spi_flash_write_data

The Write Data function uses the aforementioned Page Program function to write data of size larger than an SPI Flash page. Like Read Data, if the size passed as a parameter exceeds the flash's size it will write only the available size after the given address.

| | |
|---|---|
| Function Name | int32 **spi_flash_write_data**(uint8_t *wr_data_ptr, uint32_t address, uint32_t size) |
| Function Description | Writes a specified amount of data to the SPI flash. |
| Parameters | *wr_data_ptr:<br>Pointer to the memory position where the data to be written reside.<br>address:<br>Address where the first element will be written.<br>size:<br>Size of the data block to be written. |
| Return values | Bytes actually written. |
| Notes | Size can be any amount of data up to the available size of the SPI flash after the starting address. If the size passed as a parameter exceeds the flash's size, Write Data will only write the available size. |

#### 9.1.5.3 spi_flash_page_fill

The Page Fill instruction allows a 1-bit value to be used to fill previously erased (FFh) memory locations in a single memory page. If an entire page has to be programmed, the address should be a multiple of page size. If it is not, the addressing will wrap to the beginning of the page and the respective data will be overwritten. A partial page (fewer bytes than the size of the page) can be programmed without having any effect on other bytes within the same page.

For the Program Page instruction to be completed, the respective function sends a Write Enable instruction and then the Page Program instruction. After the transmission of all data to be written is completed, the function polls the SPI Flash Status Register to determine the completion of the Page Program instruction.

| | |
|---|---|
| Function Name | int32_t **spi_flash_page_fill**(uint8_t value, uint32_t address, uint16_t size) |
| Function Description | Fills a range within a page of the SPI flash (from 1 up to SPI_FLASH_PAGE bytes) with a 1-byte value. |
| Parameters | value:<br>The 1-byte value to which the memory range will be filled.<br>address:<br>Starting address of the range to fill<br>size:<br>Size of the range to be filled (1 to 'spi_flash_page_size' (in bytes) as set during initialization). |
| Return values | Bytes actually written or ERR_TIMEOUT in case of failure. |
| Notes | If the size passed as a parameter exceeds the flash page's size, Program Page will write only this available size. |

#### 9.1.5.4 spi_flash_fill

The Fill function uses the aforementioned Page Fill function to fill a memory range of size larger than an SPI Flash page. Like Read Data, if the size passed as a parameter exceeds the flash's size it will write only the available size after the given address.

| | |
|---|---|
| Function Name | int32 **spi_flash_fill**(uint8_t value, uint32_t address, uint32_t size) |
| Function Description | Fills a range of the SPI flash with a 1-byte value. |
| Parameters | value:<br>The 1-byte value to which the memory range will be filled.<br>address:<br>Starting address of the range to fill<br>size:<br>Size of the range to be filled |
| Return values | Bytes actually written. |
| Notes | Size can be any amount of data up to the available size of the SPI flash after the starting address. If the size passed as a parameter exceeds the flash's size, Write Data will only write the available size. |

### 9.1.6 Erase flash instructions

#### 9.1.6.1 spi_flash_block_erase

The Sector Erase function is used to erase the sector(4KB), the 32KB block or the 64KB block that the given address belongs to. The function first sends a Write Enable instruction and then the address that belongs to the sector(4KB), the 32KB block or the 64KB block to be erased.

| | |
|---|---|
| Function Name | int8_t **spi_flash_block_erase**(uint32_t address, SPI_erase_module_t spiEraseModule) |

| | |
|---|---|
| Function Description | Erases the sector(4KB), the 32KB block or the 64KB block to be erased that the specified address belongs to. |
| Parameters | address: Address belonging to the sector to be erased.<br>spiEraseModule:<br>SECTOR_ERASE : erase the (4KB) sector {address} belongs to<br>BLOCK_ERASE_32 : erase the 32KB block {address} belongs to<br>BLOCK_ERASE_64 : erase the 64KB block {address} belongs to |
| Return values | ERR_OK, ERR_TIMEOUT |
| Notes | No verification of erasure is performed in this function. |

#### 9.1.6.2 spi_flash_chip_erase

Operates like the Sector Erase function. It is not called with an address since the entire memory is erased.

| | |
|---|---|
| Function Name | int8_t **spi_flash_chip_erase** (void) |
| Function Description | Erases the whole SPI flash memory. |
| Parameters | None |
| Return values | ERR_OK, ERR_TIMEOUT |
| Notes | None |

#### 9.1.6.3 spi_flash_chip_erase_forced

Operates like the Chip Erase function. Additionally, it removes all protection schemes (complete/partial) which may have been previously configured.

| | |
|---|---|
| Function Name | int8_t **spi_flash_chip_erase_forced** (void) |
| Function Description | Erases the whole SPI flash memory, after disabling all protection schemes. |
| Parameters | None |
| Return values | ERR_OK, ERR_TIMEOUT, ERR_UNKNOWN_FLASH_TYPE |
| Notes | Only for the supported types of flash memory modules. |

### 9.1.7 Power management

#### 9.1.7.1 spi_flash_power_down

The Power Down function is used to force the flash memory device to enter power-down mode, to further reduce the power consumption. In power-down mode, all instructions except spi_flash_release_from_power_down() are ignored.

| | |
|---|---|
| Function Name | int32_t **spi_flash_power_down**(void) |
| Function Description | Sends the Power-Down instruction |
| Parameters | None |
| Return values | ERR_OK, ERR_TIMEOUT |
| Notes | Only for the supported types of flash memory modules. |

#### 9.1.7.2 spi_flash_release_from_power_down

The Release from Power Down function is used to allow the flash memory device exit from power-down mode and resume normal operation.

| | |
|---|---|
| Function Name | int32_t **spi_flash_release_from_power_down**(void) |

| | |
|---|---|
| Function Description | Sends the Release from Power-Down instruction |
| Parameters | None |
| Return values | ERR_OK, ERR_TIMEOUT |
| Notes | Only for the supported types of flash memory modules. |

### 9.1.8 Data protection

#### 9.1.8.1 spi_flash_configure_memory_protection

The Configure Memory Protection function is used to select the protection scheme applied to the memory device, in order to make some parts or the whole flash memory read-only or disable mamory protection.

| | |
|---|---|
| Function Name | int32_t **spi_flash_configure_memory_protection**(uint8_t spi_flash_memory_protection_setting) |
| Function Description | Configures the memory protection scheme applied. |
| Parameters | spi_flash_memory_protection_setting: The desired memory protection scheme:<br><br>for the W25X10 memory device:<br>W25x10_MEM_PROT_NONE: memory protection deactivated.<br>W25x10_MEM_PROT_UPPER_HALF : the upper half of the memory is protected.<br>W25x10_MEM_PROT_LOWER_HALF: the lower half of the memory is protected.<br>W25x10_MEM_PROT_ALL: the whole memory range is protected.<br><br>for the W25X20 memory device:<br>W25x20_MEM_PROT_NONE: memory protection deactivated.<br>W25x20_MEM_PROT_UPPER_QUARTER : the upper quarter of the memory is protected.<br>W25x20_MEM_PROT_UPPER_HALF : the upper half of the memory is protected.<br>W25x20_MEM_PROT_LOWER_QUARTER: the lower quarter of the memory is protected.<br>W25x20_MEM_PROT_LOWER_HALF: the lower half of the memory is protected.<br>W25x20_MEM_PROT_ALL: the whole memory range is protected. |
| Return values | ERR_OK, ERR_TIMEOUT |
| Notes | Only for the supported types of flash memory modules. |

### 9.1.9 Miscellaneous instructions

#### 9.1.9.1 spi_read_flash_memory_man_and_dev_id

Function used to read the SPI Flash's Manufacturer/Device ID.

| | |
|---|---|
| Function Name | int16_t **spi_read_flash_memory_man_and_dev_id**(void) |
| Function Description | Reads the Manufacturer/Device ID. |
| Parameters | None |
| Return values | The Manufacturer/Device ID. |
| Notes | None |

#### 9.1.9.2 spi_read_flash_unique_id

Function used to read the SPI Flash's Unique ID Number.

| | |
|---|---|
| Function Name | uint64_t **spi_read_flash_unique_id**(void) |
| Function Description | Reads the Unique ID Number. |
| Parameters | None |
| Return values | The Unique ID Number. |
| Notes | None |

#### 9.1.9.3 spi_read_flash_jedec_id

Function used to read the SPI Flash's JEDEC ID.

| | |
|---|---|
| Function Name | int32_t **spi_read_flash_jedec_id**(void) |
| Function Description | Reads the JEDEC ID. |
| Parameters | None |
| Return values | The JEDEC ID. |
| Notes | None |

### 9.1.10 Definitions

#### SPI Flash characteristics

```
typedef enum SPI_ERASE_MODULE
{
    BLOCK_ERASE_64   = 0xd8,
    BLOCK_ERASE_32   = 0x52,
    SECTOR_ERASE     = 0x20,
} SPI_erase_module_t;

#define                                 MAX_READY_WAIT_COUNT 10000
#define                                 MAX_COMMAND_SEND_COUNT 50

/* Status Register Bits */
#define STATUS_BUSY                     0x01
#define STATUS_WEL                      0x02
#define STATUS_BP0                      0x04
#define STATUS_BP1                      0x08
#define STATUS_BP2                      0x10
#define STATUS_WP                       0x80

#define ERR_OK                           0
#define ERR_TIMEOUT                     -1
#define ERR_NOT_ERASED                  -2
#define ERR_PROTECTED                   -3
#define ERR_INVAL                       -4
#define ERR_ALIGN                       -5
#define ERR_UNKNOWN_FLASH_VENDOR        -6
#define ERR_UNKNOWN_FLASH_TYPE          -7
#define ERR_PROG_ERROR                  -8

/* commands */
#define WRITE_ENABLE                    0x06
#define WRITE_ENABLE_VOL                0x50
#define WRITE_DISABLE                   0x04
#define READ_STATUS_REG                 0x05
```

```
#define WRITE_STATUS_REG                0x01
#define PAGE_PROGRAM                    0x02
#define QUAD_PAGE_PROGRAM               0x32
#define CHIP_ERASE                      0xC7
#define ERASE_SUSPEND                   0x75
#define ERASE_RESUME                    0x7a
#define POWER_DOWN                      0xb9
#define HIGH_PERF_MODE                  0xa3
#define MODE_BIT_RESET                  0xff
#define REL_POWER_DOWN                  0xab
#define MAN_DEV_ID                      0x90
#define READ_UNIQUE_ID                  0x4b
#define JEDEC_ID                        0x9f
#define READ_DATA                       0x03
#define FAST_READ                       0x0b
```

# 10 I2C EEPROM driver

## 10.1 API description

The following section lists the various functions of the I2C EEPROM driver library. These functions implement the various operations of an I2C EEPROM (e.g. Microchip 24AA02, ST M24M01-R), like read and write, plus the initialization, configuration and release of the I2C controller.

**Note 8** The source code for this driver is located in: ble_sw\dk_apps\src\plf\refip\src\driver\i2c_eeprom

### 10.1.1 How to use this driver

- Enable the I2C module and configure it, using i2c_eeprom_init()
- Read a random byte from the I2C EEPROM, using i2c_eeprom_read_byte().
- Read a desired amount of data from the I2C EEPROM, using i2c_eeprom_read_data().
- Write a random byte to the I2C EEPROM, using i2c_eeprom_write_byte().
- Write a page to the I2C EEPROM, using i2c_eeprom_write_page().
- Write a specified amount of data to the I2C EEPROM, using i2c_eeprom_write_data().
- Upon completion, if desired, disable the I2C, using i2c_eeprom_release().

### 10.1.2 Initialization and configuration

- i2c_eeprom_init ()
- i2c_eeprom_release ()

### 10.1.3 I2C EEPROM read functions

- i2c_eeprom_read_byte()
- i2c_eeprom_read_data()

### 10.1.4 I2C EEPROM write functions

- i2c_eeprom_write_byte()
- i2c_eeprom_write_page()
- i2c_eeprom_write_data()

### 10.1.5 Initialization and configuration functions

The I2C EEPROM driver provides one function for the initialization and configuration of the I2C module, i2c_eeprom_init, and one function to disable the I2C module, i2c_eeprom_release.

#### 10.1.5.1 i2c_eeprom_init

The function to be used for the initialization of the I2C module is i2c_eeprom_init(). It is called with a set of parameters that define the I2C module's operation. When the I2C module is to be configured, it is first disabled, then the control register is updated with the selected parameters, and then the module is enabled again.

| | |
|---|---|
| Function Name | void **i2c_eeprom_init**(uint16_t dev_address, uint8_t speed, uint8_t address_mode, uint8_t address_size) |
| Function Description | Initializes the I2C EEPROM according to the parameters |
| Parameters | dev_address: Slave device address (device-specific).<br>speed:<br>I2C_STANDARD = Standard (100Kb/s)<br>I2C_FAST = Fast (400Kb/s)<br>address_mode:<br>I2C_7BIT_ADDR = 7-bit addressing<br>I2C_10BIT_ADDR = 10-bit addressing<br>address_size:<br>I2C_1BYTE_ADDR = 1-byte address<br>I2C_2BYTES_ADDR = 2-bytes address<br>I2C_3BYTES_ADDR = 2-bytes address |
| Return values | None |
| Notes | The I2C module is configured as master, with restart conditions send enabled, by default. |

#### 10.1.5.2 i2c_eeprom_release

The i2c_eeprom_release function is used in order to disable the I2C module. It resets the I2C_ENABLE bit of the I2C Control Register (I2C_CON_REG) and resets the I2C_ENABLE bit of the Peripheral divider register (CLK_PER_REG).

| | |
|---|---|
| Function Name | void **i2c_eeprom_release**(void) |
| Function Description | Disables the I2C module |
| Parameters | None |
| Return values | None |
| Notes | None |

### 10.1.6 I2C EEPROM Read functions

#### 10.1.6.1 i2c_eeprom_read_byte

This function is used to read a byte from the I2C EEPROM. The function first repeatedly makes a dummy access and polls the I2C Transmit Abort Source Register, waiting for an acknowledgement that would indicate that the I2C EEPROM is not busy executing with another operation. As soon as there is an ACK of the dummy access, the function goes on to write the address to the I2C Rx/Tx Data Buffer, followed by a read command. Then, the function polls the I2C Receive FIFO Level Register, waiting for the read byte. As soon as the level on the I2C Receive FIFO Level Register is greater than 0, the function reads the byte that resides at the I2C Rx/Tx Data Buffer.

| | |
|---|---|
| Function Name | uint8_t **spi_flash_read_byte**(uint16_t address) |
| Function Description | Reads the byte that is stored at a specific address. |
| Parameters | address: The memory location of the byte to be read. |

| | |
|---|---|
| Return values | The byte that is stored at the given address. |
| Notes | None |

#### 10.1.6.2 i2c_eeprom_read_data

The Read Data instruction is used to read a data block of the SPI Flash. The function first repeatedly makes a dummy access and polls the I2C Transmit Abort Source Register, waiting for an acknowledgement that would indicate that the I2C EEPROM is not busy executing with another operation. As soon as there is an ACK of the dummy access, the function goes on to write the starting address to the I2C Rx/Tx Data Buffer, followed by as many read command as the given size. Then, the function polls the I2C Receive FIFO Level Register, waiting for the read bytes. As soon as the level on the I2C Receive FIFO Level Register is greater than 0, the function reads the data received from the I2C Rx/Tx Data Buffer. If the size is greater than 64, the above process is done in chunks of 64 bytes because of the receive FIFO limitation.

| | |
|---|---|
| Function Name | uint16_t **spi_flash_read_data**(uint8_t *rd_data_ptr,uint16_t address, uint16_t size) |
| Function Description | Reads data from a starting address of the I2C EEPROM, equal to ‹size› bytes. |
| Parameters | *rd_data_ptr:<br>Pointer to the memory position where the read data will be stored.<br>address:<br>Address of the first element to be read.<br>size:<br>Size of the data block to be read. |
| Return values | Count of the bytes actually read. |
| Notes | If the size passed as a parameter exceeds the EEPROM’s size after the given address, the function will read only the available size. The read process is done in chunks of 64 bytes due to the Rx FIFO limitation. |

### 10.1.7 I2C EEPROM Write functions

#### 10.1.7.1 i2c_eeprom_write_byte

This function is used to write a single byte to a specified I2C EEPROM location. The function first repeatedly makes a dummy access and polls the I2C Transmit Abort Source Register, waiting for an acknowledgement that would indicate that the I2C EEPROM is not busy executing with another operation. As soon as there is an ACK of the dummy access, the function goes on to write the specified address to the I2C Rx/Tx Data Buffer, followed by the byte to be written.

| | |
|---|---|
| Function Name | void **i2c_eeprom_write_byte**(uint16_t address, uint8_t wr_data) |
| Function Description | Writes a byte to a specific EEPROM’s address. |
| Parameters | address:<br>Address where the element will be written.<br>wr_data:<br>Byte to be written. |
| Return values | None |
| Notes | None |

#### 10.1.7.2 i2c_eeprom_write_page

This function is used to write a page to the I2C EEPROM. This function operates exactly like the i2c_eeprom_write_byte function, but instead of writing one byte to the I2C Rx/Tx Data Buffer it goes on to write an amount of bytes equal to ‹size›. If the size passed as a parameter exceeds the EEPROM page’s size, the function will write only the available size till the end of the page.

| | |
|---|---|
| Function Name | uint16_t **i2c_eeprom_write_page**(uint8_t* wr_data_ptr , uint32_t address , uint16_t size) |
| Function Description | Writes a specified amount of data to the I2C EEPROM. |
| Parameters | *wr_data_ptr:<br>Pointer to the memory position where the data to be written reside.<br>address:<br>Address where the first element will be written.<br>size:<br>Size of the data block to be written (1 to I2C_EEPROM_PAGE). |
| Return values | Count of the bytes actually written. |
| Notes | The function will validate the address against the EEPROM's size and will write only until the end of the page where the starting address resides, thus avoiding rollback(writing the excessive bytes to the beginning of the page). |

#### 10.1.7.3 i2c_eeprom_write_data

This function is used to write an amount of data, possibly greater than a page, to the I2C EEPROM without the limitation of the starting address to be a multiple of the EEPROM's page size. This function uses the i2c_eeprom_write_page function and at first checks whether the ‹size› given exceeds the available size from the given address to the end of the EEPROM. Then, it calculates the amount of the bytes to be written from the given address to the end of the corresponding EEPROM page. It then calls the write page function for this amount, and continues by executing as many write page functions as needed until the desired size has been written.

| | |
|---|---|
| Function Name | uint32_t **i2c_eeprom_write_data**(uint8_t *wr_data_ptr,uint32_t address, uint16_t size) |
| Function Description | Writes a specified amount of data to the I2C EEPROM. |
| Parameters | *wr_data_ptr:<br>Pointer to the memory position where the data to be written reside.<br>address:<br>Address where the first element will be written.<br>size:<br>Size of the data block to be written. |
| Return values | Count of bytes actually written. |
| Notes | Starting address does not need to be a multiple of I2C_EEPROM_PAGE. Size can be any amount of data up to the available size of the I2C EEPROM after the starting address. If the size passed as a parameter exceeds the EEPROM's size, the function will only write the available size. |

## 10.2 Definitions

```
enum I2C_SPEED_MODES{
  I2C_STANDARD,
  I2C_FAST,
};
enum I2C_ADDRESS_MODES{
  I2C_7BIT_ADDR,
  I2C_10BIT_ADDR,
};
enum I2C_ADRESS_BYTES_COUNT{
  I2C_1BYTE_ADDR,
  I2C_2BYTES_ADDR,
  I2C_3BYTES_ADDR,
};
```

### 10.3 Defines in the application necessary to the I2C EEPROM driver

The following pre-processor directives must be defined to their corresponding values in the application, in order for the driver to handle the various requests.

- I2C_EEPROM_SIZE : the size of the eeprom in bytes
- I2C_EEPROM_PAGE : the eeprom page size in bytes
- I2C_SPEED_MODE : I2C_STANDARD or I2C_FAST
- I2C_ADDRESS_MODE: I2C_7BIT_ADDR or I2C_10BIT_ADDR
- I2C_ADDRESS_SIZE : I2C_1BYTE_ADDR, I2C_2BYTES_ADDR or I2C_3BYTES_ADDR

## 11 PWM TIMERS driver

### 11.1 API description

The following section lists the various functions of the PWM TIMERS driver library. These functions implement the various operations to support the configuration and operation of the TIMER0 and TIMER2 drivers:

TIMER0

- Controls the PWM signals PWM0 and PWM1 which is always the complementary of PWM0.
- If needed, the interrupt SWTIM_IRQn can be configured to be triggered in intervals configured separately.

TIMER2

- Controls the PWM signals PWM2, PWM3 and PWM4 which all use the same frequency with individually configured duty cycles.
- If needed, the interrupt SWTIM_IRQn can be enable to be triggered, in intervals that are separately configurable.

**Note 9** The source code for this driver is located in: ble_sw\dk_apps\src\plf\refip\src\driver\pwm

### 11.2 How to use this driver

Below, you can find instructions on the initialization, use and release of the PWM TIMERS library.

Important notes:

a. The user application is responsible for the correct configuration of any pads that are to be driven by the PWM0,PWM1 (TIMER0) and PWM2,PWM3,PWM4(TIMER2) signals.

   For example, a line of the format:

   ```
   GPIO_ConfigurePin(GPIO_PORT_0, GPIO_PIN_1, OUTPUT, PID_PWM0, true);
   ```

   Will configure pin P0_1 to be driven by the PWM0 signal.

b. You should enable the TIMER0/TIMER2 peripheral clock for both TIMER0 and TIMER2, using set_tmr_enable().

TIMER0

- Enable the TIMER0/TIMER2 peripheral clock, using set_tmr_enable().
- If you intend to use the 16MHz clock source, you can select the TIMER0/TIMER2 clock division factor, using set_tmr_div(). This setting does not apply in the case of the 32kHz clock source.
- Initialize PWM with the desired pwm mode, TIMER0 “on” time division(please note: it affects only the “on” time) option and clock source selection settings, using timer0_init().
- Set the TIMER0 “on”, “high” and “low“ times, using timer0_set().
- If you wish to use interrupts:
  - In your application, define a (callback) function of the form:

`void pwm_user_callback_function(void)`
IMPORTANT NOTE: You should always keep the code size in this function to the bare minimum, in order to keep your application responsive.

- Register this callback function (will be called by the interrupt handler of SWTIM_IRQn), using timer0_register_callback().
- Enable SWTIM_IRQn, using timer0_enable_irq().

- Start TIMER0, using timer0_start().
- Stop TIMER0, using timer0_stop().
- If you wish, disable the TIMER0/TIMER2 peripheral clock, using set_tmr_enable(). Note: Be cautious as disabling the common peripheral clock, TIMER2 will also cease to run.
- If you wish, disable SWTIM_IRQn, using timer0_disable_irq().

TIMER2

- Enable the TIMER0/TIMER2 peripheral clock, using set_tmr_enable().
- Set the TIMER0/TIMER2 clock division factor, using set_tmr_div(). Please, keep in mind that this setting is common to both TIMER0 and TIMER2.
- Initialize PWM with the desired hw_pause behaviour, sw_pause setting, using timer2_init().
- Set the duty cycle for the PWM signal(s) you wish, using timer2_set_pwm2_duty_cycle(), timer2_set_pwm3_duty_cycle, timer2_set_pwm4_duty_cycle.
- If you have initialized with the sw_pause enabled, release the sw_pause, using timer2_set_sw_pause(). In any case, the timer starts.
- Stop the timer, enabling sw_pause again whenever you wish, using the same function.
- Stop and disable the timer, using timer2_stop().
- If you wish, disable the TIMER0/TIMER2 peripheral clock, using set_tmr_enable(). Note: Be cautious as TIMER0 will also cease to run.

### 11.2.1 List of available functions

The PWM TIMERS driver provides the following functions:

TIMER0 (PWM0, PWM1)

- set_tmr_enable()
- set_tmr_div()
- timer0_init()
- timer0_start()
- timer0_stop()
- timer0_release()
- timer0_set_pwm_on_counter()
- timer0_set_pwm_high_counter()
- timer0_set_pwm_low_counter()
- timer0_set()
- timer0_enable_irq()
- timer0_disable_irq()
- timer0_register_callback()

TIMER2 (PWM2, PWM3, PWM4)

- timer2_enable()
- timer2_set_hw_pause()
- timer2_set_sw_pause()
- timer2_set_pwm_frequency()

- timer2_init()
- timer2_stop()
- timer2_set_pwm2_duty_cycle()
- timer2_set_pwm3_duty_cycle()
- timer2_set_pwm4_duty_cycle()

### 11.2.2 Summary of available functions

BOTH TIMERS (TIMER0, TIMER2)

- set_tmr_enable(): enables the peripheral clock to both TIMER0 and TIMER2, set_tmr_enable().
- set_tmr_div(): sets the division factor for the peripheral clock (not applicable for 32k clock source),

TIMER0

- timer0_init(): initializes TIMER0. The pwm mode of operation, the TIMER0 “on” time clock division option and the clock source are selected here.
- timer0_start(): starts TIMER0, if it has been initialized ( timer0_init() ).
- timer0_stop(): stops TIMER0.
- timer0_release(): same as timer0_stop().
- timer0_set_pwm_on_counter(): sets the value of TIMER0 “on(ON)” counter. This is the counter that controls the intervals between SWTIM_IRQn interrupts.
- timer0_set_pwm_high_counter(): sets the value of TIMER0 “high(M)” counter.
- timer0_set_pwm_low_counter(): sets the value of TIMER0 “low(N)” counter.
- timer0_set: Sets the values of the “on(ON)”, “high(M)” and “low(N)” counters, in a single function call.
- timer0_enable_irq(): Enables the SWTIM_IRQn irq.
- timer0_disable_irq(): Disables the SWTIM_IRQn irq.
- timer0_register_callback(): Register a user defined callback function that is called from the SWTIM_IRQn interrupt handler of the driver. IMPORTANT NOTE: You should always keep the code size in this function to the bare minimum, in order to keep your application responsive.

TIMER2

- timer2_enable(): enables/disables TIMER2.
- timer2_set_hw_pause(): enables/disables the  hw pause feature of TIMER2.
- timer2_set_sw_pause(): enables/disables the  sw pause feature of TIMER2.
- timer2_set_pwm_frequency(): sets the pwm frequency of TIMER2.
- timer2_init(): enables/disables the  hw pause and the sw features of TIMER2 and sets the pwm frequency of TIMER2, in a single function call.
- timer2_stop(): stops TIMER2.
- timer2_set_pwm2_duty_cycle(): Sets the duty cycle of PWM2.
- timer2_set_pwm3_duty_cycle(): Sets the duty cycle of PWM3.
- timer2_set_pwm4_duty_cycle(): Sets the duty cycle of PWM4.

### 11.2.1 Functions Reference

#### 11.2.1.1 et_tmr_enable

Enables the peripheral clock to both TIMER0 and TIMER2.

| Function Name | void **set_tmr_enable**(CLK_PER_REG_TMR_ENABLE_t clk_per_reg_tmr_enable) |
|---|---|

| | |
|---|---|
| Function Description | Enables the peripheral clock to both TIMER0 and TIMER2 |
| Parameters | clk_per_reg_tmr_enable:<br>CLK_PER_REG_TMR_DISABLED:<br>disables the peripheral clock to both TIMER0 and TIMER2<br>CLK_PER_REG_TMR_ENABLED:<br>enables the peripheral clock to both TIMER0 and TIMER2 |
| Return values | None |
| Notes | None |

#### 11.2.1.2 set_tmr_div

Sets the division factor for the peripheral clock (not applicable for 32k clock source). Affects TIMER0, when clocked from 16MHz clock and TIMER2 always.

| | |
|---|---|
| Function Name | void **set_tmr_div**(CLK_PER_REG_TMR_DIV_t per_tmr_div) |
| Function Description | Sets the division factor for the peripheral clock (not applicable for 32k clock source), Affects TIMER0, when clocked from 16MHz clock and TIMER2 always. |
| Parameters | per_tmr_div:<br>CLK_PER_REG_TMR_DIV_1 : The clock peripheral division factor is 1<br>CLK_PER_REG_TMR_DIV_2 :: The clock peripheral division factor is 2<br>CLK_PER_REG_TMR_DIV_4 :: The clock peripheral division factor is 4<br>CLK_PER_REG_TMR_DIV_8 :: The clock peripheral division factor is 8 |
| Return values | None |
| Notes | Not applicable for 32k clock source. Affects TIMER0, when clocked from 16MHz clock and TIMER2 always. |

#### 11.2.1.3 timer0_init

Initializes TIMER0.

| | |
|---|---|
| Function Name | void **timer0_init**(TIM0_CLK_SEL_t tim0_clk_sel , PWM_MODE_t pwm_mode, TIM0_CLK_DIV_t tim0_clk_div) |
| Function Description | Initializes TIMER0. |
| Parameters | pwm_mode:<br>PWM_MODE_ONE: The PWM signal will be always HIGH duting the “high time”.<br>PWM_MODE_CLOCK_DIV_BY_TWO: The PWM signals are not HIGH during the “high time” but output a clock in that stage. The frequency is based on the 16MHz peripheral clock (also when 32 kHz clock is used as the timer clock source), divided by the value set with timer0_init, but divided by two to get a 50 % duty cycle. So, for example, if a factor of 8 has been selected, the clock that will be observed during the “high times” of PWM0 and PWM1, will have a frequency of: (16MHz/8)/2 = 1MHz, irrespectively of the selected clock source for TIMER0.<br>tim0_clk_div: TIMER0 “on” time (on duration) division factor. Please, note that this parameter affects only the PWM “on” time. It can be either TIM0_CLK_NO_DIV, where the internal “on” counter is clocked by the same clock as TIMER0, or TIM0_CLK_DIV_BY_10, where the internal “on” counter is clocked by 1/10 of the clock used for TIMER0. This parameter affects the intervals between SWTIM_IRQn interrupts. |

| | |
|---|---|
| | tim0_clk_sel: Selects the clock source user for TIMER0.<br>TIM0_CLK_32K: the 32kHz clock source is used<br>TIM0_CLK_FAST: The 16MHz clock source is used |
| Return values | None |
| Notes | None |

#### 11.2.1.4 timer0_start

Starts TIMER0, if it has been initialized ( timer0_init() ).

| | |
|---|---|
| Function Name | void **timer0_start**(void) |
| Function Description | Starts TIMER0, provided it has been previously initialized ( timer0_init() ). |
| Parameters | None |
| Return values | None |
| Notes | The timer has to be initialized ( timer0_init() ) |

#### 11.2.1.5 timer0_stop

Stops TIMER0

| | |
|---|---|
| Function Name | void **timer0_stop**(void) |
| Function Description | Stops TIMER0. |
| Parameters | None |
| Return values | None |
| Notes | None |

#### 11.2.1.6 timer0_release

Releases TIMER0. Same as timer0_stop.

| | |
|---|---|
| Function Name | void **timer0_release**(void) |
| Function Description | Releases TIMER0. Same as timer0_stop. |
| Parameters | None |
| Return values | None |
| Notes | Exists for compliance to the driver architecture nomenclature |

#### 11.2.1.7 timer0_set_pwm_on_counter

Sets the PWM “ON” counter value.

| | |
|---|---|
| Function Name | void **timer0_set_pwm_on_counter**(uint16_t pwm_on) |
| Function Description | Sets the PWM “ON” counter value. |
| Parameters | pwm_on: The PWM “ON” counter value to set. |
| Return values | None |
| Notes | Is directly associated with the value of the tim0_clk_div parameter in timer0_init. |

#### 11.2.1.8 timer0_set_pwm_high_counter

Sets the PWM “HIGH” counter value.

| | |
|---|---|
| Function Name | void **timer0_set_pwm_high_counter**(uint16_t pwm_high) |
| Function Description | Sets the PWM “HIGH” counter value. |
| Parameters | pwm_high: The PWM “HIGH” counter value to set. |
| Return values | None |
| Notes | None |

#### 11.2.1.9 timer0_set_pwm_low_counter

Sets the PWM “LOW” counter value.

| | |
|---|---|
| Function Name | void **timer0_set_pwm_low_counter**(uint16_t pwm_low) |
| Function Description | Sets the PWM “LOW” counter value. |
| Parameters | pwm_low: The PWM “LOW” counter value to set. |
| Return values | None |
| Notes | None |

#### 11.2.1.10 timer0_set

Sets the PWM “ON”, “HIGH” and “LOW” counter values in a single function call.

| | |
|---|---|
| Function Name | void **timer0_set**(uint16_t pwm_on, uint16_t pwm_high, uint16_t pwm_low) |
| Function Description | Sets the PWM “ON”, “HIGH” and “LOW” counter values in a single function call. |
| Parameters | pwm_on: The PWM “ON” counter value to set.<br>pwm_high: The PWM “HIGH” counter value to set.<br>pwm_low: The PWM “LOW” counter value to set. |
| Return values | None |
| Notes | Please, refer to the paragraphs for timer0_set_pwm_on_counter, timer0_set_pwm_high_counter, timer0_set_pwm_low_counte |

#### 11.2.1.11 timer0_enable_irq

Enables the SWTIM_IRQn irq.

| | |
|---|---|
| Function Name | void **timer0_enable_irq**(void) |
| Function Description | Enables the SWTIM_IRQn irq. |
| Parameters | None |
| Return values | None |
| Notes | None |

#### 11.2.1.12 timer0_disable_irq

Disables the SWTIM_IRQn irq.

| | |
|---|---|
| Function Name | void **timer0_disable_irq**(void) |
| Function Description | Disables the SWTIM_IRQn irq. |
| Parameters | None |

| | |
|---|---|
| Return values | None |
| Notes | None |

#### 11.2.1.13 timer0_register_callback

Registers a callback function that is called from the body of the SWTIM_IRQn irq handler in the driver

| | |
|---|---|
| Function Name | void **timer0_register_callback**(timer0_handler_function_t* callback) |
| Function Description | Registers a callback function that is called from the body of the SWTIM_IRQn irq handler in the driver. |
| Parameters | callback: the user callback function |
| Return values | None |
| Notes | The user callback function has to be of type timer0_handler_function_t |

#### 11.2.1.14 timer2_enable

Enables/disables TIMER2

| | |
|---|---|
| Function Name | void **timer2_enable**(TRIPLE_PWM_ENABLE_t triple_pwm_enable); |
| Function Description | Enables/disables TIMER2. |
| Parameters | triple_pwm_enable:<br>TRIPLE_PWM_DISABLED : TIMER2 disabled<br>TRIPLE_PWM_ENABLED : TIMER2 enabled |
| Return values | None |
| Notes | On disable, it does not disable the TIM clock, as it is shared with TIMER0. |

#### 11.2.1.15 timer2_set_hw_pause

Enables/disables the “pause_by_hw” TIMER2 feature.

| | |
|---|---|
| Function Name | void **timer2_set_hw_pause**(TRIPLE_PWM_HW_PAUSE_EN_t hw_pause_en); |
| Function Description | Enables/disables the “pause_by_hw” TIMER2 feature, that allows for the h/w to disable the TIMER2 pwm, during e.g. radio transmission and reception, to reduce interference. (HW_PAUSE_EN bit of TRIPLE_PWM_REGISTER) |
| Parameters | hw_pause_en:<br>HW_CAN_NOT_PAUSE_PWM_2_3_4 : TIMER2 cannot be paused by hardware<br>HW_CAN_PAUSE_PWM_2_3_4 : TIMER2 can be paused by hardware |
| Return values | None |
| Notes | None |

#### 11.2.1.16 timer2_set_sw_pause

Pauses/resumes TIMER2 operation (SW_PAUSE_EN)

| | |
|---|---|
| Function Name | void **timer2_set_sw_pause**(TRIPLE_PWM_SW_PAUSE_EN_t sw_pause_en); |
| Function Description | Pauses/resumes TIMER2 operation. (SW_PAUSE_EN bit of TRIPLE_PWM_REGISTER) |
| Parameters | sw_pause_en: |

| | |
|---|---|
| | PWM_2_3_4_SW_PAUSE_DISABLED : TIMER2 gets paused by software.<br>PWM_2_3_4_SW_PAUSE_ENABLED : TIMER2 operation resumes. |
| Return values | None |
| Notes | None |

#### 11.2.1.17 timer2_set_pwm_frequency

Sets the TIMER2 TRIPLE_PWM_FREQUENCY value, that is the reload value used by the internal counter that determines the TIMER2 frequency.

| | |
|---|---|
| Function Name | void **timer2_set_pwm_frequency**(uint16_t triple_pwm_frequency) |
| Function Description | Sets the TIMER2 TRIPLE_PWM_FREQUENCY value, that is the reload value used by the internal counter that determines the TIMER2 frequency. |
| Parameters | pwm_frequency: The value that will be automatically reloaded to the counter that determines the TIMER2 frequency. So, if this value is e.g. $500_d$ and the per_tmr_div parameter in set_tmr_div() is CLK_PER_REG_TMR_DIV_8, the TIMER2 frequency (PWM2,PWM3,PWM4) will be: (16MHz/8)/$500_d$ = 4kHz. |
| Return values | None |
| Notes | None |

#### 11.2.1.18 timer2_init

Initializes TIMER2 parameters in a single function call.

| | |
|---|---|
| Function Name | void **timer2_init**(TRIPLE_PWM_HW_PAUSE_EN_t hw_pause_en, TRIPLE_PWM_SW_PAUSE_EN_t sw_pause_en, uint16_t triple_pwm_frequency) |
| Function Description | Initializes TIMER2 parameters in a single function call. |
| Parameters | hw_pause_en:<br>HW_CAN_NOT_PAUSE_PWM_2_3_4 : TIMER2 cannot be paused by hardware<br>HW_CAN_PAUSE_PWM_2_3_4 : TIMER2 can be paused by hardware<br><br>sw_pause_en:<br>PWM_2_3_4_SW_PAUSE_DISABLED : TIMER2 gets paused by software.<br>PWM_2_3_4_SW_PAUSE_ENABLED : TIMER2 operation resumes.<br><br>triple_pwm_frequency:<br>The value that will be automatically reloaded to the counter that determines the TIMER2 frequency. So, if this value is e.g. $500_d$ and the per_tmr_div parameter in set_tmr_div() is CLK_PER_REG_TMR_DIV_8, the TIMER2 frequency (PWM2,PWM3,PWM4) will be: (16MHz/8)/$500_d$ = 4kHz. |
| Return values | None |
| Notes | You can also set the parameters individually, using the other dedicated driver functions. |

#### 11.2.1.19 timer2_stop

Stops timer2. Same as timer2_enable(TRIPLE_PWM_DISABLED)

| | |
|---|---|
| Function Name | void **timer2_stop**(void) |

| | |
|---|---|
| Function Description | Stops timer2. Same as timer2_enable. |
| Parameters | None |
| Return values | None |
| Notes | See note at timer2_enable function |

#### 11.2.1.20 timer2_set_pwm2_duty_cycle

Sets the PWM2_DUTY_CYCLE value, that is the reload value used by the internal counter that determines the TIMER2 PWM2 duty cycle.

| | |
|---|---|
| Function Name | void **timer2_set_pwm2_duty_cycle**(uint16_t pwm2_duty_cycle) |
| Function Description | Sets the PWM2_DUTY_CYCLE value, that is the reload value used by the internal counter that determines the TIMER2 PWM2 duty cycle. |
| Parameters | pwm2_duty_cycle: the reload value used by the internal counter that determines the TIMER2 PWM2 duty cycle. So, if e.g. the pwm_frequency parameter of timer2_set_pwm_frequency() function is $500_d$, and pwm2_duty_cycle parameter is 100, the duty cycle will be 100/500 = 20%. |
| Return values | None |
| Notes | None |

#### 11.2.1.21 timer2_set_pwm3_duty_cycle

Sets the PWM3_DUTY_CYCLE value, that is the reload value used by the internal counter that determines the TIMER2 PWM3 duty cycle.

| | |
|---|---|
| Function Name | void **timer2_set_pwm3_duty_cycle**(uint16_t pwm3_duty_cycle) |
| Function Description | Sets the PWM3_DUTY_CYCLE value, that is the reload value used by the internal counter that determines the TIMER2 PWM3 duty cycle. |
| Parameters | Pwm3_duty_cycle: the reload value used by the internal counter that determines the TIMER2 PWM3 duty cycle. |
| Return values | None |
| Notes | Associated with pwm_frequency parameter. See example in timer2_set_pwm2_duty_cycle |

#### 11.2.1.22 timer2_set_pwm4_duty_cycle

Sets the PWM4_DUTY_CYCLE value, which is the reload value used by the internal counter that determines the TIMER2 PWM4 duty cycle.

| | |
|---|---|
| Function Name | void **timer2_set_pwm4_duty_cycle**(uint16_t pwm4_duty_cycle) |
| Function Description | Sets the PWM4_DUTY_CYCLE value, that is the reload value used by the internal counter that determines the TIMER2 PWM4 duty cycle. |
| Parameters | Pwm4_duty_cycle: the reload value used by the internal counter that determines the TIMER2 PWM4 duty cycle. |
| Return values | None |
| Notes | Associated with pwm_frequency parameter. See example in timer2_set_pwm2_duty_cycle |

## 11.3 Definitions

```
typedef enum
{
    PWM_MODE_ONE,
    PWM_MODE_CLOCK_DIV_BY_TWO
} PWM_MODE_t;

typedef enum
{
    TIM0_CLK_DIV_BY_10,
    TIM0_CLK_NO_DIV
} TIM0_CLK_DIV_t;

typedef enum
{
    TIM0_CLK_32K,
    TIM0_CLK_FAST
} TIM0_CLK_SEL_t;

typedef enum
{
    TIM0_CTRL_OFF_RESET,
    TIM0_CTRL_RUNNING
} TIM0_CTRL_t;

typedef enum
{
    CLK_PER_REG_TMR_DISABLED,
    CLK_PER_REG_TMR_ENABLED,
} CLK_PER_REG_TMR_ENABLE_t;

typedef enum
{
    CLK_PER_REG_TMR_DIV_1,
    CLK_PER_REG_TMR_DIV_2,
    CLK_PER_REG_TMR_DIV_4,
    CLK_PER_REG_TMR_DIV_8
} CLK_PER_REG_TMR_DIV_t;

typedef enum
{
    HW_CAN_NOT_PAUSE_PWM_2_3_4,
    HW_CAN_PAUSE_PWM_2_3_4
} TRIPLE_PWM_HW_PAUSE_EN_t;

typedef enum
{
    PWM_2_3_4_SW_PAUSE_DISABLED,
    PWM_2_3_4_SW_PAUSE_ENABLED
} TRIPLE_PWM_SW_PAUSE_EN_t;

typedef enum
{
    TRIPLE_PWM_DISABLED,
    TRIPLE_PWM_ENABLED
} TRIPLE_PWM_ENABLE_t;

typedef void (timer0_handler_function_t)(void);
```

# 12 QUADRATURE DECODER driver

## 12.1 QUADRATURE DECODER driver API description

The following section lists the various functions of the QUADRATURE DECODER driver library. These functions implement the various operations to support the interfacing to a rotary encoder of up to three axes (X,Y,Z) plus the initialization, configuration and release of the driver interface.

**Note 10** The source code for this driver is located in: ble_sw\dk_apps\src\plf\refip\src\driver\wkupct_quadec

### 12.1.1 How to use this driver

Important note 1:

When the wakeup timer, the quadrature controller or both are used in your application, the pre-processor directives: WKUP_ENABLED and/or QUADEC_ENABLED respectively must be defined in your application, to allow for the inclusion of essential parts of the code.

Important note 2:

If, upon reception of interrupt from the wakeup timer or the quadrature decoder, the system resumes from sleep mode and you wish to resume peripherals functionality, it is necessary to include in your wakeup handler function(s) - the one(s) you register using wkupct_register_callback() and/or quad_decoder_register_callback() - the following lines:

```
// Init System Power Domain blocks: GPIO, WD Timer, Sys Timer, etc.
// Power up and init Peripheral Power Domain blocks,
// and finally release the pad latches.
  if(GetBits16(SYS_STAT_REG, PER_IS_DOWN))
     periph_init();
```

With polling:

- Enable and initialize the quadrature block using quad_decoder_init().
- Poll quadrature decoder counter values using quad_decoder_get_x_counter(),quad_decoder_get_y_counter(), quad_decoder_get_z_counter().
- Release the quadrature decoder driver, using quad_decoder_release().

With interrupts:

- Register a callback function to be called from within WKUP_QUADEC_Handler, when interrupt is sourced from quadrature decoder, using quad_decoder_register_callback().
- In the callback function, placed in your application code, the quadec counter values for x,y,z are passed as parameters for further processing.
- Setup and enable the interrupts for quadec, using quad_decoder_enable_irq().
- After you have finished, if desired, disable the quadec irq (caution: the irq will be disabled only if it has not been anabled also by Wakeup Timer), using disable_quadec_irq().
- Release the quadrature decoder driver, using quad_decoder_release(). (The function disable_quadec_irq() is also called).

### 12.1.2 Initialization and configuration

- void quad_decoder_init()
- quad_decoder_register_callback()
- quad_decoder_release()
- quad_decoder_enable_irq()
- quad_decoder_disable_irq()

### 12.1.3 Reading quadrature decoder counters

- quad_decoder_get_x_counter()
- quad_decoder_get_y_counter()

- quad_decoder_get_z_counter()

### 12.1.4 Initialization and configuration functions

- The QUADRATURE DECODER driver provides one function for the initialization and configuration of the quadec block, quad_decoder_init() and one function to disable the quadec, quad_decoder_release.
- For working with interrupts, driver provides a function to register a feedback function, quad_decoder_register_callback(), a function to enable irq, quad_decoder_enable_irq() and a function to disable irq, quad_decoder_disable_irq().

#### 12.1.4.1 quad_decoder_init

The function to be used for the initialization of the quadec module is quad_decoder_init(). It is called with a structure of parameters that define the quadrarure decoder's operation.

| | |
|---|---|
| Function Name | void **quad_decoder_init**(QUAD_DEC_INIT_PARAMS_t *quad_dec_init_params); |
| Function Description | Initializes the quadec module according to the parameters |
| Parameters | quad_dec_init_params:<br>chx_port_sel :Selection of port X pads (see below for CHX_PORT_SEL_t struct)<br>chy_port_sel :Selection of port Y pads (see below for CHY_PORT_SEL_t struct)<br>chz_port_sel :Selection of port Z pads (see below for CHZ_PORT_SEL_t struct)<br>qdec_clockdiv: The quadrature decoder operates on the system clock. This parameter defines the number of clock cycles every which the decoding logic samples the data input on the channels lines. |
| Return values | None |
| Notes | |

#### 12.1.4.2 quad_decoder_release

The quad_decoder_release function is used in order to disable the quadec module. It resets the pin assignment of the quadec to QUAD_DEC_CHXA_NONE_AND_CHXB_NONE, QUAD_DEC_CHYA_NONE_AND_CHYB_NONE and QUAD_DEC_CHZA_NONE_AND_CHZB_NONE. Finally, it sets the QUAD_ENABLE bit of CLK_PER_REG to 0, to disable the quadec. Also, calls the function disable_quadec_irq().

| | |
|---|---|
| Function Name | void **quad_decoder_release**(void) |
| Function Description | Disables the quadec module |
| Parameters | None |
| Return values | None |
| Notes | None |
| | |

#### 12.1.4.3 quad_decoder_register_callback

The quad_decoder_register_callback function is used in order to assign a callback function to be called by the handler of the WKUP_QUADEC_IRQn, when the interrupt source is detected to be the quadec.

| | |
|---|---|
| Function Name | void **quad_decoder_register_callback**(uint32_t* callback) |
| Function Description | Assigns a callback function to be called by the handler of the WKUP_QUADEC_IRQn, when the interrupt source is detected to be the quadec. |
| Parameters | Callback : pointer to the callback function |
| Return values | None |

| | |
|---|---|
| Notes | The callback function has to be of the type:<br>void my_quad_decoder_user_callback_function(int16_t qdec_xcnt_reg, int16_t qdec_ycnt_reg, int16_t qdec_zcnt_reg) |

#### 12.1.4.4 quad_decoder_enable_irq

The quad_decoder_enable_irq function is used to setup and enable the interrupts for the quadec. Any pending WKUP_QUADEC_IRQn interrupt is cleared, the count of quadec events to trigger an interrupt is set, the QD_IRQ_MASK is reset and the WKUP_QUADEC_IRQn is enabled.

| | |
|---|---|
| Function Name | void **quad_decoder_enable_irq**(uint8_t event_count) |
| Function Description | Setup and enable the interrupts for quadec. |
| Parameters | event_count: the count of quadec events to trigger an interrupt |
| Return values | None |
| Notes | |

#### 12.1.4.5 quad_decoder_disable_irq

The quad_decoder_disable_irq is used to unregister the quadrature controller from the use of the WKUP_QUADEC_IRQn and call wkupct_quad_disable_IRQ() function to disable the interrupts for quadec and wakeup timer, only if no registration from wakeup timer is active.

| | |
|---|---|
| Function Name | wkupct_quadec_error_t **quad_decoder_disable_irq**(void) |
| Function Description | Unregisters the quadrature controller from the use of the WKUP_QUADEC_IRQn and calls wkupct_quad_disable_IRQ() function to disable the interrupts for the quadec and the wakeup timer, only if no registration from the wakeup timer is active. |
| Parameters | None |
| Return values | WKUPCT_QUADEC_ERR_OK: ok<br><br>WKUPCT_QUADEC_ERR_OK: The wakeup timer has previously registered for WKUP_QUADEC_IRQn. |
| Notes | |

### 12.1.5 Quadrature decoder counter read functions

#### 12.1.5.1 quad_decoder_get_x_counter

The quad_decoder_get_x_counter function is used to retrieve the current value of the QDEC_XCNT_REG register, which holds the counter for the X channel of the quadrature decoder.

| | |
|---|---|
| Function Name | inline int16_t **quad_decoder_get_x_counter**(void) |
| Function Description | Retrieves the current value of the QDEC_XCNT_REG register. |
| Parameters | None |
| Return values | The current value of the QDEC_XCNT_REG register. |

#### 12.1.5.2 quad_decoder_get_y_counter

The quad_decoder_get_y_counter function is used to retrieve the current value of the QDEC_YCNT_REG register, which holds the counter for the Y channel of the quadrature decoder.

| | |
|---|---|
| Function Name | inline int16_t **quad_decoder_get_y_counter**(void) |
| Function Description | Retrieves the current value of the QDEC_YCNT_REG register. |

| | |
|---|---|
| Parameters | None |
| Return values | The current value of the QDEC_YCNT_REG register. |

#### 12.1.5.3 quad_decoder_get_z_counter

The quad_decoder_get_z_counter function is used to retrieve the current value of the QDEC_ZCNT_REG register, which holds the counter for the Z channel of the quadrature decoder.

| | |
|---|---|
| Function Name | inline int16_t **quad_decoder_get_z_counter**(void) |
| Function Description | Retrieves the current value of the QDEC_ZCNT_REG register. |
| Parameters | None |
| Return values | The current value of the QDEC_ZCNT_REG register. |

## 12.2 Definitons

```
typedef void (*quad_encoder_handler_function_t)(int16_t qdec_xcnt_reg, int16_t
qdec_ycnt_reg, int16_t qdec_zcnt_reg);
typedef enum
{
    QUAD_DEC_CHXA_NONE_AND_CHXB_NONE = 0,
    QUAD_DEC_CHXA_P00_AND_CHXB_P01 = 1,
    QUAD_DEC_CHXA_P02_AND_CHXB_P03 = 2,
    QUAD_DEC_CHXA_P04_AND_CHXB_P05 = 3,
    QUAD_DEC_CHXA_P06_AND_CHXB_P07 = 4,
    QUAD_DEC_CHXA_P10_AND_CHXB_P11 = 5,
    QUAD_DEC_CHXA_P12_AND_CHXB_P13 = 6,
    QUAD_DEC_CHXA_P23_AND_CHXB_P24 = 7,
    QUAD_DEC_CHXA_P25_AND_CHXB_P26 = 8,
    QUAD_DEC_CHXA_P27_AND_CHXB_P28 = 9,
    QUAD_DEC_CHXA_P29_AND_CHXB_P20 = 10
}CHX_PORT_SEL_t;

typedef enum
{
    QUAD_DEC_CHYA_NONE_AND_CHYB_NONE = 0<<4,
    QUAD_DEC_CHYA_P00_AND_CHYB_P01 = 1<<4,
    QUAD_DEC_CHYA_P02_AND_CHYB_P03 = 2<<4,
    QUAD_DEC_CHYA_P04_AND_CHYB_P05 = 3<<4,
    QUAD_DEC_CHYA_P06_AND_CHYB_P07 = 4<<4,
    QUAD_DEC_CHYA_P10_AND_CHYB_P11 = 5<<4,
    QUAD_DEC_CHYA_P12_AND_CHYB_P13 = 6<<4,
    QUAD_DEC_CHYA_P23_AND_CHYB_P24 = 7<<4,
    QUAD_DEC_CHYA_P25_AND_CHYB_P26 = 8<<4,
    QUAD_DEC_CHYA_P27_AND_CHYB_P28 = 9<<4,
    QUAD_DEC_CHYA_P29_AND_CHYB_P20 = 10<<4
}CHY_PORT_SEL_t;

typedef enum
{
    QUAD_DEC_CHZA_NONE_AND_CHZB_NONE = 0<<8,
    QUAD_DEC_CHZA_P00_AND_CHZB_P01 = 1<<8,
    QUAD_DEC_CHZA_P02_AND_CHZB_P03 = 2<<8,
    QUAD_DEC_CHZA_P04_AND_CHZB_P05 = 3<<8,
    QUAD_DEC_CHZA_P06_AND_CHZB_P07 = 4<<8,
    QUAD_DEC_CHZA_P10_AND_CHZB_P11 = 5<<8,
    QUAD_DEC_CHZA_P12_AND_CHZB_P13 = 6<<8,
    QUAD_DEC_CHZA_P23_AND_CHZB_P24 = 7<<8,
    QUAD_DEC_CHZA_P25_AND_CHZB_P26 = 8<<8,
```

```
    QUAD_DEC_CHZA_P27_AND_CHZB_P28 = 9<<8,
    QUAD_DEC_CHZA_P29_AND_CHZB_P20 = 10<<8
}CHZ_PORT_SEL_t;

typedef struct
{
    CHX_PORT_SEL_t chx_port_sel;
    CHY_PORT_SEL_t chy_port_sel;
    CHZ_PORT_SEL_t chz_port_sel;
    uint16_t qdec_clockdiv;
    uint8_t qdec_events_count_to_trigger_interrupt;
}QUAD_DEC_INIT_PARAMS_t;

typedef enum
{
    WKUPCT_QUADEC_ERR_RESERVED = -1,
    WKUPCT_QUADEC_ERR_OK = 0,
} wkupct_quadec_error_t;

typedef enum
{
    RESERVATION_STATUS_FREE = 0,
    RESERVATION_STATUS_RESERVED,
} reservation_status_t;

typedef void (*quad_encoder_handler_function_t)(int16_t qdec_xcnt_reg, int16_t
qdec_ycnt_reg, int16_t qdec_zcnt_reg);
```

### 12.3 Defines in the application necessary to the QUADRATURE DECODER (QUADEC) driver

The following pre-processor directive must be defined in the application, in order to include necessary parts of the code:

QUADEC_ ENABLED

(See: 12.1.1 important note 1)

# 13 WAKEUP TIMER driver

## 13.1 API description

The following section lists the various functions of the WAKEUP TIMER driver library. These functions support the configuration of the Wakeup Interrupt Controller (WIC).

**Note 11** The source code for this driver is located in: ble_sw\dk_apps\src\plf\refip\src\driver\wkupct_quadec

### 13.1.1 How to use this driver

**Note 12** When the wakeup timer, the quadrature controller or both are used in your application, the preprocessor directives: WKUP_ENABLED and/or QUADEC_ENABLED respectively must be defined in your application, to allow for the inclusion of essential parts of the code.

**Note 13** If, upon reception of an interrupt from the wakeup timer or the quadrature decoder, the system resumes from sleep mode and you wish to resume peripherals functionality, it is necessary to include in your wakeup handler function(s) - the one(s) you register using wkupct_register_callback() and/or quad_decoder_register_callback() - the following lines:

```
// Init System Power Domain blocks: GPIO, WD Timer, Sys Timer, etc.
// Power up and init Peripheral Power Domain blocks,
// and finally release the pad latches.
  if(GetBits16(SYS_STAT_REG, PER_IS_DOWN))
     periph_init();
```

- Register a callback function that is called from the driver's WKUP_QUADEC_IRQn irq handler, using wkupct_register_callback().
- Enable the WKUP_QUADEC_IRQn irq with the wakup parameters, using wkupct_enable_irq().

### 13.1.2 List of available functions

- wkupct_register_callback()
- wkupct_enable_irq()
- wkupct_disable_irq()

### 13.1.3 Summary of available functions

- wkupct_register_callback(): registers a callback function that is called from the driver's WKUP_QUADEC_IRQn irq handler.
- wkupct_enable_irq(): Registers the wakeup timer for use of WKUP_QUADEC_IRQn irq and enables WKUP_QUADEC_IRQn with the desired wakeup parameters.
- wkupct_disable_irq(): Unregisters the wakeup timer from the use of WKUP_QUADEC_IRQn irq and calls wkupct_quad_disable_IRQ() function to disable the interrupts for the quadec and the wakeup timer, only if no registration from the quadrature decoder is active.

### 13.1.4 Functions Reference

#### 13.1.4.1 wkupct_register_callback

Registers a callback function that is called from the driver's WKUP_QUADEC_IRQn irq handler.

| | |
|---|---|
| Function Name | void **wkupct_register_callback**(uint32_t* callback) |
| Function Description | Registers a callback function that is called from the driver's WKUP_QUADEC_IRQn irq handler. |
| Parameters | callback: the user-defined callback function |
| Return values | None |
| Notes | A local pointer to this function is stored in the retention memory area. |

#### 13.1.4.2 wkupct_enable_irq

Enables and configures the wakeup timer and enables the WKUP_QUADEC_IRQn.

| | |
|---|---|
| Function Name | void **wkupct_enable_irq**(uint32_t sel_pins, uint32_t pol_pins, uint16_t events_num, uint16_t deb_time) |
| Function Description | Enables and configures the wakeup timer and enables the WKUP_QUADEC_IRQn. |
| Parameters | sel_pins:<br>Select enabled inputs: Bits 0-7 -> port 0, Bits 8-15 -> port 1, Bits -> 16-23 port 2,<br>Bits 24-31 -> port 3. 0-diasbled, 1-enabled.<br>pol_pins:<br>Inputs' polarity: Bits 0-7 -> port 0, Bits 8-15 -> port 1, Bits 16-23 -> port 2,<br>Bits 24-31 -> port 3. 0-high, 1-low.<br>events_num:<br>Number of events before wakeup interrupt. Max 255.<br>deb_time:<br>Debouncing time. Max 0x3F. |

| | |
|---|---|
| Return values | None |
| Notes | None |

#### 13.1.4.3 wkupct _disable_irq

The wkupct_disable_irq is used to unregister Wakeup Timer from the use of the WKUP_QUADEC_IRQn and call wkupct_quad_disable_IRQ() function to disable the interrupts for quadec and wakeup timer, only if no registration from Quadrature Decoder is active.

| | |
|---|---|
| Function Name | wkupct_quadec_error_t **wkupct_disable_irq**(void) |
| Function Description | Unregisters the wakeup timer from the use of the WKUP_QUADEC_IRQn and calls wkupct_quad_disable_IRQ() function to disable the interrupts for the quadec and the wakeup timer, only if no registration from the quadrature decoder is active. |
| Parameters | None |
| Return values | WKUPCT_QUADEC_ERR_OK: ok<br>WKUPCT_QUADEC_ERR_OK: the quadrature decoder has previously registered for WKUP_QUADEC_IRQn. |
| Notes | |

## 13.2 Definitions

```
enum
{
    SRC_WKUP_IRQ = 0x01,
    SRC_QUAD_IRQ,
};

typedef enum
{
    WKUPCT_QUADEC_ERR_RESERVED = -1,
    WKUPCT_QUADEC_ERR_OK = 0,
} wkupct_quadec_error_t;

typedef enum
{
    RESERVATION_STATUS_FREE = 0,
    RESERVATION_STATUS_RESERVED,
} reservation_status_t;

typedef void (*wakeup_handler_function_t)(void);
```

## 13.3 Defines in the application necessary to the WAKEUP TIMER driver

The following preprocessor directive must be defined in the application, in order to include necessary parts of the code: WKUP_ENABLED (see: 13.1.1 important note 1).

# 14 Revision history

| Revision | Date | Description |
|---|---|---|
| 1.0 | 19-Mar-2014 | Initial version. |
| 1.1 | 8-Jul-2014 | SPI FLASH libraries updated, minor changes |

## Status definitions

| Status | Definition |
|---|---|
| DRAFT | The content of this document is under review and subject to formal approval, which may result in modifications or additions. |
| APPROVED or unmarked | The content of this document has been approved for publication. |

## Disclaimer

Information in this document is believed to be accurate and reliable. However, Dialog Semiconductor does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information. Dialog Semiconductor furthermore takes no responsibility whatsoever for the content in this document if provided by any information source outside of Dialog Semiconductor.

Dialog Semiconductor reserves the right to change without notice the information published in this document, including without limitation the specification and the design of the related semiconductor products, software and applications.

Applications, software, and semiconductor products described in this document are for illustrative purposes only. Dialog Semiconductor makes no representation or warranty that such applications, software and semiconductor products will be suitable for the specified use without further testing or modification. Unless otherwise agreed in writing, such testing or modification is the sole responsibility of the customer and Dialog Semiconductor excludes all liability in this respect.

Customer notes that nothing in this document may be construed as a license for customer to use the Dialog Semiconductor products, software and applications referred to in this document. Such license must be separately sought by customer with Dialog Semiconductor.

All use of Dialog Semiconductor products, software and applications referred to in this document are subject to Dialog Semiconductor's Standard Terms and Conditions of Sale, unless otherwise stated.



## RoHS Compliance

Dialog Semiconductor complies to European Directive 2001/95/EC and from 2 January 2013 onwards to European Directive 2011/65/EU concerning Restriction of Hazardous Substances (RoHS/RoHS2).
Dialog Semiconductor's statement on RoHS can be found on the customer portal https://support.diasemi.com/. RoHS certificates from our suppliers are available on request.

## Contacting Dialog Semiconductor

Germany Headquarters
*Dialog Semiconductor GmbH*
Phone: +49 7021 805-0

United Kingdom
*Dialog Semiconductor (UK) Ltd*
Phone: +44 1793 757700

The Netherlands
*Dialog Semiconductor B.V.*
Phone: +31 73 640 8822

Email:
enquiry@diasemi.com

North America
*Dialog Semiconductor Inc.*
Phone: +1 408 845 8500

Japan
*Dialog Semiconductor K. K.*
Phone: +81 3 5425 4567

Taiwan
*Dialog Semiconductor Taiwan*
Phone: +886 281 786 222

Web site:
www.dialog-semiconductor.com

Singapore
*Dialog Semiconductor Singapore*
Phone: +65 64 849929

China
*Dialog Semiconductor China*
Phone: +86 21 5178 2561

Korea
*Dialog Semiconductor Korea*
Phone: +82 2 3469 8291