Tutorial

RTC Concept

For the DA1468x SoC

Abstract

This tutorial should be used as a reference guide to gain a deeper understanding of the 'Real-Time Clock' concept. As such, it covers a broad range of topics including an introduction to RTC measurements as well as a detailed description of the RTC mechanism – a custom implementation for the DA1468x family of devices. Furthermore, it covers a number of sections containing in depth software analysis of a complete demonstration example.

Contents

Figures

Tables

Terms and Definitions

API	Application Programming Interface
DevKit	Development Kit
HW	Hardware
lp	Low Power
OS	Operating System
RTC	Real – Time Clock
SDK	Software Development Kit

References

[1] UM-B-044, DA1468x Software Platform Reference, User Manual, Dialog Semiconductor.

1 Introduction

1.1 Before You Start

Before you start you need to:

- Install the latest SmartSnippets Studio
- Download the latest SDK for the DA1468x platforms

These can be downloaded from the Dialog Semiconductor support portal.

Additionally, for this tutorial either a Pro or Basic Development kit is required.

The key goals of this tutorial are to:

- Provide a basic understanding of RTC functionality
- Explain the different APIs and configurations of the RTC mechanism
- Give a complete sample project demonstrating the usage of the RTC mechanism

2 Real-Time Clock Concept

This section explains the key features of the Real-Time Clock (RTC). It is sometimes necessary to accurately measure time, for instance you may need to measure the current time or how much time it takes for a routine to be executed. Advanced architectures often include a hardware block dedicated to such operations but this is not the case for all architectures including the DA1468x family of devices. For these architectures, an RTC must be implemented using an alternative approach. To do this, the SDK includes some APIs that can be used for measuring time with high accuracy.

2.1 OS Timers vs RTC

Usually, Real Time Operating Systems (RTOS) include timers that can be used for various RTOS-related operations. These timers are events with the Operating System's (OS) tick granularity, thus their accuracy is not as good as the one of a hardware-implemented RTC. OS timers may also include delays caused by higher priority tasks or system interrupts. For these reasons, we recommend using the RTC-related APIs provided by the SDK if timestamping is required.

2.2 RTC Implementation

The RTC implementation measures the time, in low power (lp) clock cycles, that has elapsed since the device's power up or hardware (HW) reset. This implies that the accuracy of the RTC functionality is the accuracy of the lp clock used. Using the number of the lp clock cycles that fit within a certain time interval and the period of each lp clock cycle, time can be easily computed using the following formula:

$$Time = Period\ of\ lp\ Clock\ Cycle * Number\ of\ lp\ Clock\ Cycles$$

Figure 1: Formula for Converting lp Clocks to Time

For instance, assuming the external crystal XTAL32K is selected as the lp clock and the RTC functionality has measured a value equal to 1000 lp clock cycles, time can be computed as follows:

***Time** = (1000000 / 32768) * 1000 = 30.518 us * 1000 = 30.518 ms*

Figure 2: Formula for Converting lp Clocks to Time – Example

The APIs related to RTC functionality can be found in sdk/cpm/include/sys_rtc.h. There are only two APIs that can be used by the developer. Table 1 briefly explains the available APIs (red indicates the path under which the files are stored, and green indicates which ones are used for RTC related operations).

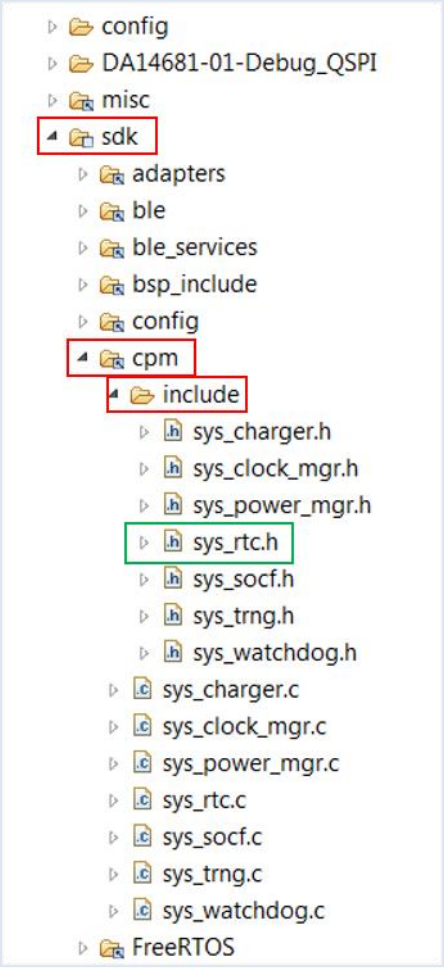

Figure 3: Headers for RTC Implementation.

Table 1: APIs for the RTC Functionality

API Name	Description
rtc_get()	This returns the current RTC time expressed in lp clock cycles, that is either XTAL32K or RCX.
rtc_get_fromISR()	This has the same functionality as rtc_get(). It should be called from within an Interrupt Service Routine (ISR).

2.3 Supported Low Power Clocks

Picture 4 illustrates the clocks that can be used as the lp clock when the device is in sleep mode. The device can also be clocked with an external digital clock in place of the external crystal. Currently, the SDK does not support the internal RC32K RC oscillator. For the DA1468x Pro DevKit, XTAL32K is the place where an external crystal of 32768 Hz has been be attached and RCX is the internal RC oscillator of the chip. At room temperature, the RC oscillator frequency is close to 10.5 kHz.

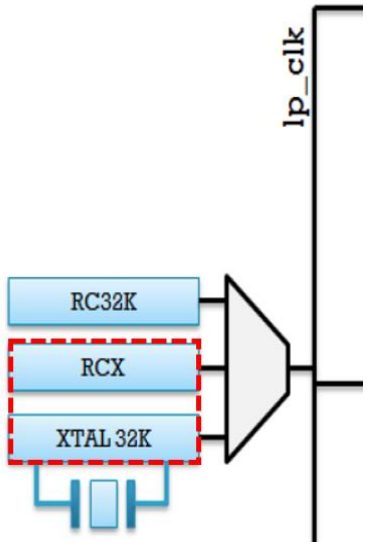

Figure 4: Available Sources for lp Clocks in Sleep Mode.

It is recommended that the customer uses the external crystal over the internal RC oscillator. The main reasons for this are stability and accuracy. In general, crystals are quite stable and feature more accuracy. Due to their nature, RC oscillators are vulnerable to both voltage and temperature variations. This implies that the RCX frequency changes over time depending on its surrounding. Thus calibrations need to be performed at regular time intervals.

Note: The RCX accuracy for the DA1468x family of devices is 500 ppm. This is enough for both BLE and non-BLE operations. To preserve the aforementioned accuracy, it is necessary for the RCX to be calibrated frequently.

When recording chip characterization information, the maximum time RCX can remain uncalibrated is 4 seconds at room temperature and the expected temperature dependent drift of the RCX is +/-50 ppm/^{o}C.

The default SDK algorithm for RCX calibration assumes that there is a regular wake/sleep cycle of the device due to BLE activity. However, if there are either long sleep or long active periods, or, even worse, if the device is always active (for example, while plugged-in to a USB port and charging) then the RCX is not calibrated sufficiently and it will drift more than expected. This non-expected drifting of the RCX significantly affects the RCX accuracy and, in extreme cases, the stability of the BLE link. For such cases, request the RCX-patch from Dialog support which implements a more sophisticated RCX calibration approach including, adaptive to temperature ramping, RCX calibration.

RTC Concept

The SDK comes with a few useful macros that should be used by the developer to compute the period of the lp clock used. These are useful for the RCX where its frequency is constantly changing. Table 2 briefly explains the most useful macros that can be used during RTC measurements.

Table 2: Useful Macros for RTC Measurements

Macro Name	Description
configSYSTICK_CLOCK_HZ	This macro returns the frequency of the selected lp clock. It should be used for all of the supported lp clocks.
rcx_clock_period	This macro returns the period of the lp clock and should only be used for the RCX. Please note that the returned value is multiplied by (1024 * 1024) to increase accuracy.

Note: The calibration process of the RCX uses the external crystal XTAL16M as a reference clock. This means that if the XTAL16M is not trimmed, then this inaccuracy is reflected in the RCX calibration. Therefore, if the RCX is the selected lp clock source then its drift is also reflected in the RTC functionality.

2.4 Best Practices for Dealing with RTCs

2.4.1 Optimizing CPU Performance Using Integer Values

For devices that do not incorporate a **Floating-Point Unit** (FPU), it is good practice for the developer to use integer numbers as much as possible. Otherwise, it takes many clock cycles for the CPU to execute non-integer calculations and this has adverse consequences on the overall system's performance. If possible, use integer numbers that are a power of two, for instance 2^10 = 1024 or (2^10 * 2^10) = (1024 * 1024). In fact, any multiplication which involves numbers that are a power of two, is interpreted by the compiler as a bit-shift operation to the left. Similarly, any division which involves numbers that are a power of two, is interpreted as a bit-shift operation to the right.

2.4.2 Optimizing CPU Performance Using Small Numbers

As mentioned, the RTC mechanism measures lp clock cycles on device power up or cold boot (HW reset). This implies that values returned by the rtc_get() function increase as time passes. Instead of measuring all the lp clock cycles every time an RTC measurement is performed, it is good practice to only use the lp clock cycles since the last RTC measurement. Picture 5 illustrates this concept using a pair of variables named old and new respectively.

RTC Concept

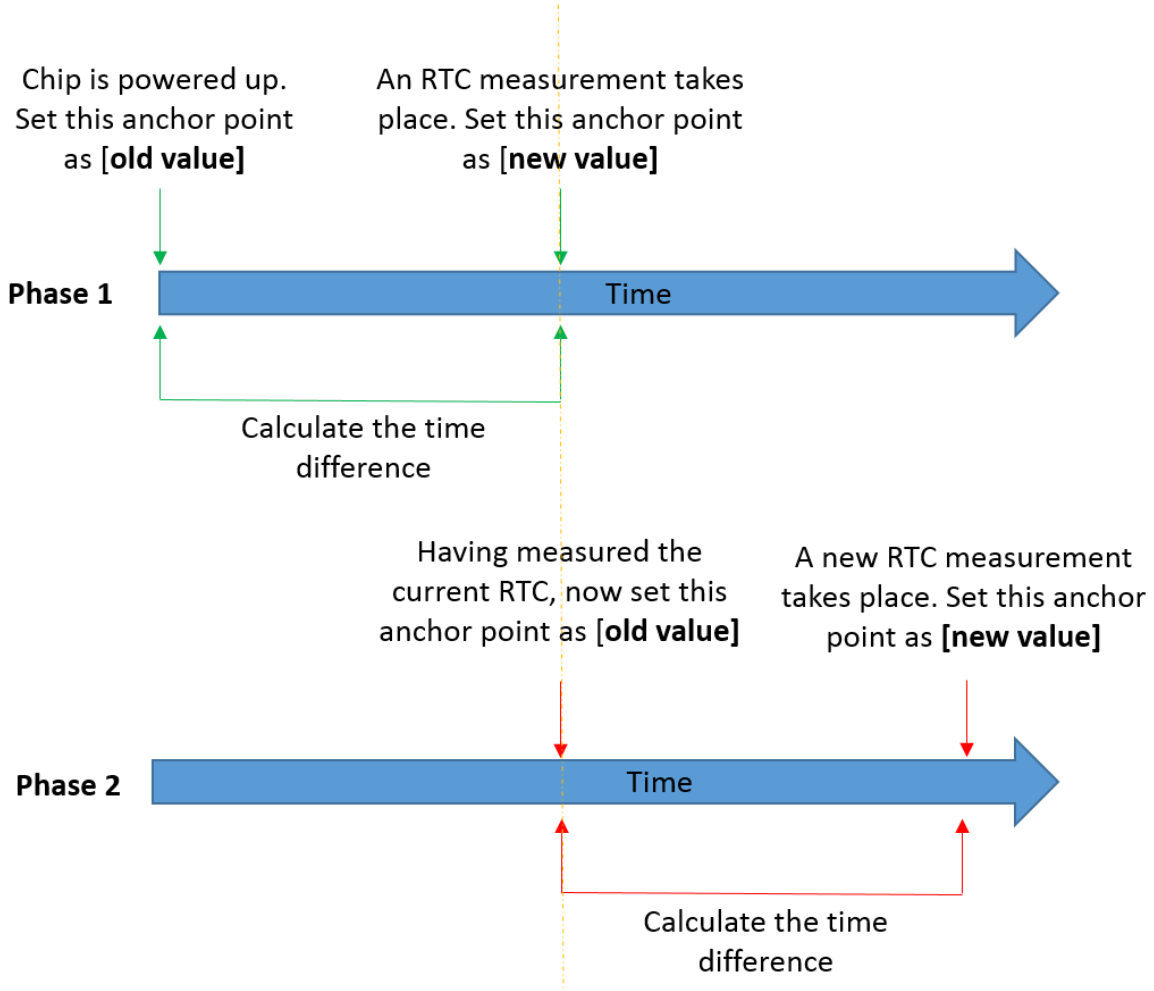

Figure 5: Consecutive RTC Measurements Using the Old-New Scheme

The following code snippet demonstrates a possible RTC measurement routine using this old-new scheme:

```
__RETAINED static uint64_t sw_rtc_new = 0;
__RETAINED static uint64_t sw_rtc_old = 0;

void old_new_scheme(void) {

        uint64_t sw_rtc_dif = 0;

        /* Get the current RTC value expressed in ticks of the lp clock used. */
        sw_rtc_new = rtc_get();

        /*
         * To reduce values used in various calculations, calculate
         * only the lp clocks since the last RTC measurement.
         * (instead of invoking all the lp clock cycles measured
         * from the very beginning of chip's operation.
         */
        sw_rtc_dif = sw_rtc_new - sw_rtc_old;


        /*
```

```
         * The current RTC value becomes the old value
         * for the next RTC measurement.
         */
        sw_rtc_old = sw_rtc_new;
}
```

2.5 Converting lp Clocks to Time

The following code snippet implements all the aforementioned concepts, tips, and insights, and illustrates a routine that could be used for converting and translating lp clock cycles to time.

```
/*
 * Convert low power (lp) clocks into time expressed in microseconds.
 */
uint64_t sw_rtc_convert_lp_to_time(uint64_t lp_clocks)
{
        uint64_t time = 0;

        /*
         * Indentify the lp clock used
         */
        if ((dg_configUSE_LP_CLK == LP_CLK_32768)
                        || (dg_configUSE_LP_CLK == LP_CLK_32000)) {

                /*
                 * Compute the period of the lp clock used
                 * and then multiply with 1024 to increase
                 * accuracy!
                 */
                const uint32_t lp_clk_period = ((1000000 * 1024)

/configSYSTICK_CLOCK_HZ);

                /*
                 * Convert lp clocks into time
                 */
                time = (lp_clocks * (uint64_t)lp_clk_period);
                time = (time >> 10); // divide by 1024

        } else if (dg_configUSE_LP_CLK == LP_CLK_RCX) {

                /*
                 * Use [rcx_clock_period] to get the current RCX
                 * period in microseconds. Please note that this
                 * value is multiplied by (1024 * 1024).
                 */
                time = (lp_clocks * (uint64_t)rcx_clock_period);
                time = (time >> 20); // divide with (1024 * 1024)

        } else {

                OS_ASSERT(0); // Invalid LP clock source
        }

        return time;
}
```

3 Demonstration Example - RTC Description

This section analyzes an application example which demonstrates using the RTC functionality. The application also demonstrates alerts based on freeRTOS Timers functionality. The example uses the **freertos_retarget** sample code, found in the SDK, as its framework. Two additional tasks have been created: one task performs RTC measurements and, based on the results, computes the current date and time in standard format; the second task performs alert operations based on freeRTOS events. The code also enables the wake-up timer for handling external events. To aid readability, using the alert functionality is described in the next section.

3.1 Application Structure of RTC Functionality

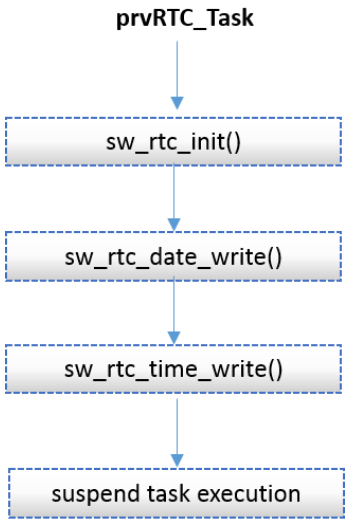

Figure 6: Application SW RTC FSM – Initialization Process

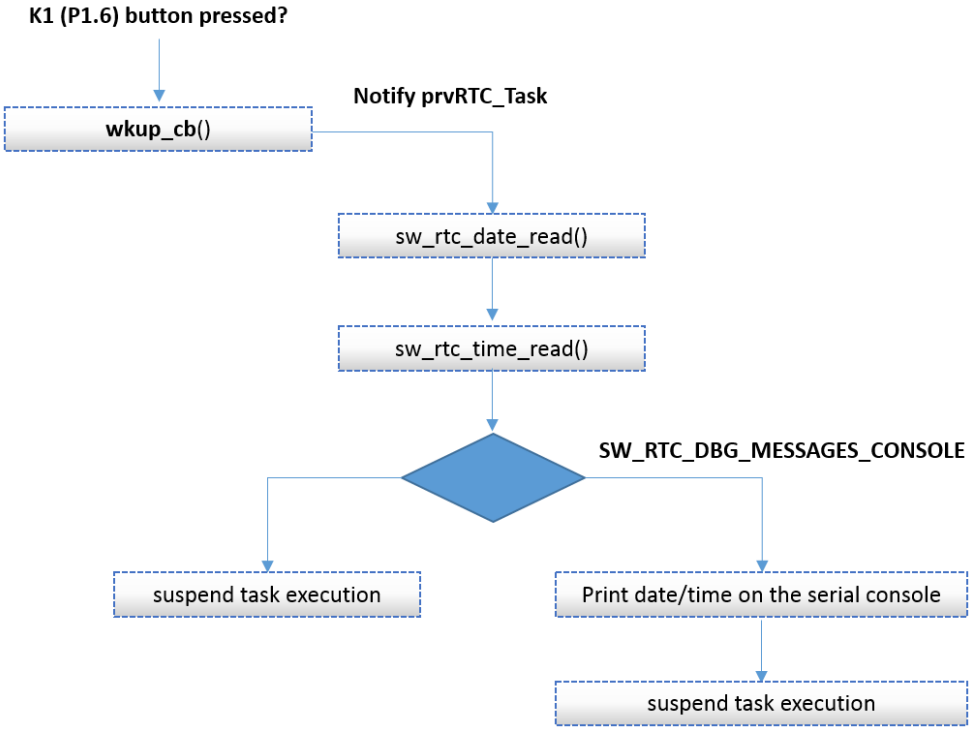

Figure 7: Application SW RTC FSM – Main Execution Path

1. The key goal of this task is for the device to calculate the current time and date following an event. For demonstration purposes button **K1** (P1.6) on the Pro DevKit has been configured as a wake-up pin. For more information on how to configure and set a pin to handle external events, please read the External Interruption tutorial. At each external event (produced at every button press - K1 on Pro Devkit) a dedicated callback function named wkup_cb() is triggered. This callback executes many operations including the activation of prvRTC_Task().

2. In this freeRTOS task, the current date and time is read through the RTC functionality. Depending on the value of **SW_RTC_DBG_MESSAGES_CONSOLE,** the current date and time is printed out on the serial console.

3.2 Time Related APIs and Macros

The sample code has a set of APIs that can be used to perform time measurements. Table 3 briefly explains the most useful APIs that should be used for time measurement operations.

Table 3: APIs for the RTC Functionality

API Name	Description
sw_rtc_init()	This function initializes all the resources required for the SW RTC mechanism. It should be the first invoked API before any other SW RTC related operation.
sw_rtc_date_time_read()	This function reads the current date and time. It should be considered as a shortcut function since it combines both sw_rtc_date_read() and sw_rtc_time_read(). **Note:** This function can be called at any time from any task without overlap issues.
sw_rtc_date_read()	This function reads the current date only. **Note:** This function can be called at any time from any task without overlap issues.
sw_rtc_time_read()	This function reads the current time only. **Note:** This function can be called at any time from any task without overlap issues.
sw_rtc_date_time_write()	This function sets the current date and time. It should be considered as a shortcut function since it combines both sw_rtc_date_write() and sw_rtc_time_write(). Sanity checks are performed to check the validity of the assigned values (values out of the allowable boundaries). **Note:** This function can be called at any time from any task without overlap issues. **Note:** The weekday is calculated automatically by the code.

sw_rtc_date_write()	This function sets the current date only. Sanity checks are performed to check the validity of the assigned values (values out of the allowable boundaries). **Note:** This function can be called at any time from any task without overlap issues. **Note:** The weekday is calculated automatically by the code.
sw_rtc_time_write()	This function sets the current time only. Sanity checks are performed to check the validity of the assigned values (values out of the allowable boundaries). **Note:** This function can be called at any time from any task without overlap issues.

Table 4 briefly explains the most useful structures for SW RTC measurements.

Table 4: Structures for the RTC Functionality

Structure Name	Description
sw_rtc_date_time_t	This structure contains parameters both for setting/reading the current date and time.
sw_rtc_date_t	This structure contains parameters for setting/reading the current date only.
sw_rtc_time_t	This structure contains parameters for setting/reading the current time only

4 Demonstration Example - Alert Description

This section demonstrates a possible implementation of an alert mechanism. In such cases, time accuracy is not as critical as in RTC operations and therefore SW timers, like the ones offered by freeRTOS, can be used for triggering alerts.

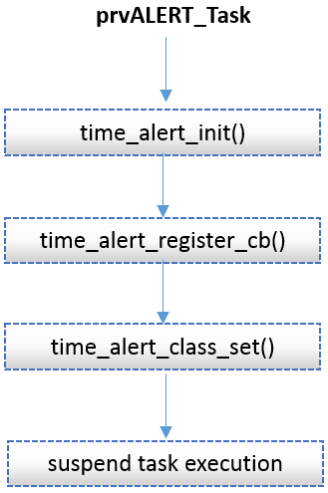

Figure 8: SW FSM of the Alert Functionality

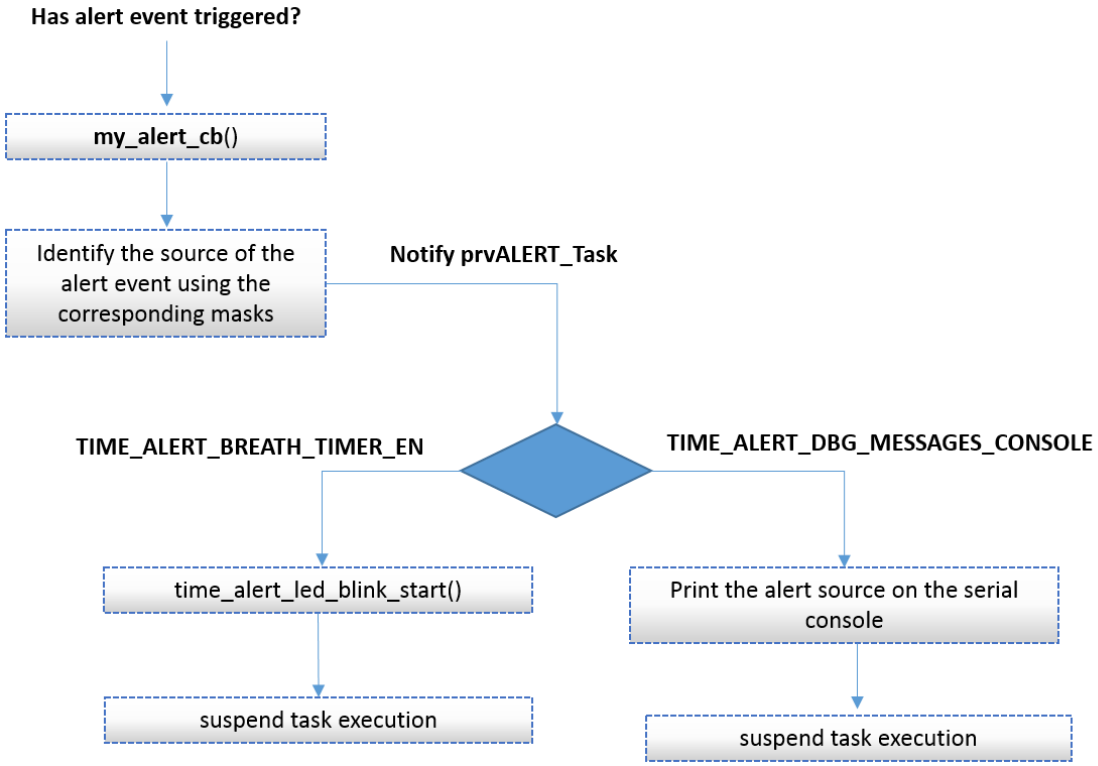

Figure 9: Application Alerts SW FSM – Main Execution Path

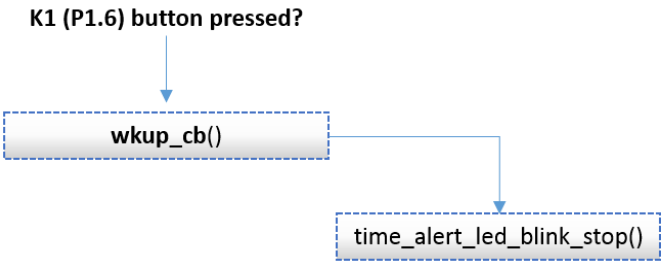

Figure 10: Application Alerts SW FSM – Main Execution Path Continued

4.1 Application Structure of Alert Functionality

1. The key goal of this task is for the device to trigger alerts following a freeRTOS event. The code features three different alert classes: **Seconds Class**, **Minutes Class** and **Hours Class**. Depending on the preferred time resolution, the developer is free to choose any of the classes or a combination or all of them at the same time.

2. So far, the provided demonstration example features two different types of alert:

   - One type of alert uses the breath timer and the white led of the chip to blink led D1 on Pro DevKit at a constant frequency. At any time, alerts can be disabled by pressing the K1 button on Pro DevKit. To enable blinking alerts, the developer should set

TIME_ALERT_BREATH_TIMER_EN to '1'. Please note that blinking functionality is not standard feature of the alert mechanism.

- The second type of alert, which is enabled by default, uses callback functions. The developer can define a function that is called automatically at each alert event. Please note that this callback function is the same regardless of the alert class that has been configured to trigger alerts. For this reason, the callback function passes a dedicated variable that should be used by the developer to determine the source of the alert event. For user's convenience, the sample code exhibits macros that should be used to determine the alert source.

Code snippet of the user-defined callback function:

```
/*
 * User-defined callback function that is called upon an alert event.
 * Note that all alert classes will call the same function. Thus, it
 * is recommended that the developer will use \p [source] to determine
 * the source of the alert
 */
void my_alert_cb(uint8_t source)
{

        /*
         * Check whether [Hours Class] triggered an alert.
         * Then notify [prvALERT_Task] task accordingly.
         */
        if (source & ALERT_SOURCE_HOURS_Msk) {

                // Do something...
        }

        /*
         * Check whether [Minutes Class] triggered an alert.
         * Then notify [prvALERT_Task] task accordingly.
         */
        if (source & ALERT_SOURCE_MINUTES_Msk) {

                // Do something...
        }

        /*
         * Check whether [Seconds Class] triggered an alert.
         * Then notify [prvALERT_Task] task accordingly.
         */
        if (source & ALERT_SOURCE_SECONDS_Msk) {

                // Do something...

        }
}
```

Note: The device must remain in active mode while the breath timer is functioning. By pressing the K1 button on Pro DevKit, LED D1 will stop blinking and the device will return to its initial state, that is extended sleep mode.

4.2 Alert Related APIs and Macros

The provided sample code exhibits a set of APIs that can be used to perform alert operations. Table 5 briefly explains the most useful APIs for alerts.

Table 5: APIs for the Alert Functionality

API Name	Description
time_alert_init()	This function initializes all the resources required for the alert mechanism. It should be the first invoked API before any other alert related operation.
time_alert_register_cb()	This function register callback function to be triggered upon an alert event. **Note:** All the three alert classes call the same callback function. It's, user's responsibility to identify the source of the alert event. Developer can use the appropriate masks (starting with **ALERT_SOURCE**_xxx_xxx).
time_alert_unregister_cb()	This function unregisters any previously registered callback function.
time_alert_class_set()	This function sets alert events for a specific alert class, that is either Hours, Minutes or Seconds
time_alert_class_set_all()	This function sets alert events for all the three alert classes at once. This function can be considered as a shortcut function.
time_alert_class_clear()	This function disables/clears alert events either for an individual alert class, or for all the three at once.
void (*alert_cb) (uint8_t class)	Function prototype for callback functions

Table 6 briefly explains the most useful macros/enums for Alert related operations.

Table 6: Macros/Enums for the Alert Functionality

Macro/Enum Name	Description
SET_ALERT_CLASS	Use this enum to specify an alert class during setting related operations
CLEAR_ALERT_CLASS	Use this enum to specify an alert class/classes during clearance related operations.
ALERT_SOURCE_HOURS_Msk	Use this mask to check whether the source of an alert event is [Hours]
ALERT_SOURCE_MINUTES_Msk	Use this mask to check whether the source of an alert event is [Minutes]

ALERT_SOURCE_SECONDS_Msk	Use this mask to check whether the source of an alert event is [Seconds]

5 Running The Demonstration Example

This section describes the steps required to prepare the Pro DevKit and other tools to successfully run the example code. A serial terminal and optionally a frequency meter or an oscilloscope are required for testing and verifying the code. For information on configuring a serial terminal, as well as a Pro DevKit, read the Starting a Project tutorial.

5.1 Measuring the lp Clock Frequency

This demonstration example gives the developer the ability to measure the selected lp clock by using either a frequency meter or an oscilloscope. The lp clock pulses can be selected from P3.0 pin on Pro DevKit, which by default is disabled. To do so, **TEST_LP_CLOCK_OUTPUT** in main.c should be set to '1'. Please note that this feature is intended only for debugging purposes. Picture 8 illustrates the frequency of the selected XTAL32K lp clock, as measured using a digital frequency meter.

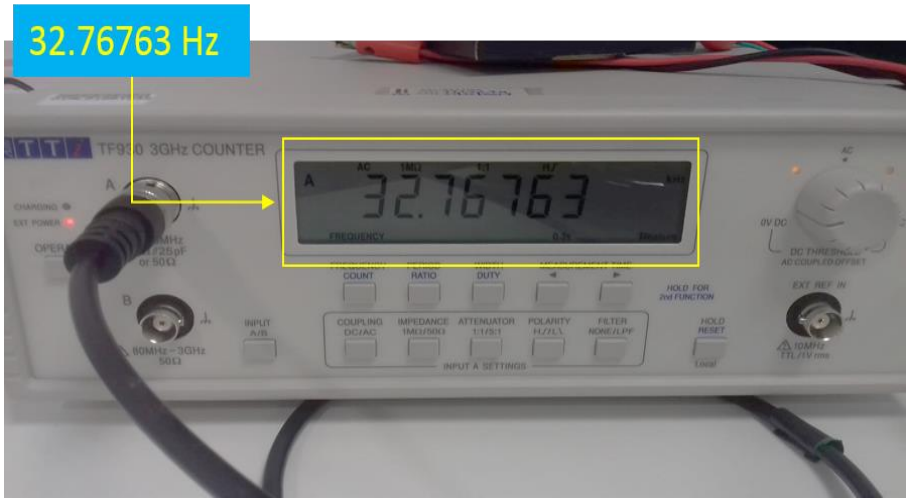

Figure 11: Measuring the lp Clock Period

Warning: In order for the RTC measurements to be as accurate as possible, the measured frequency should be as close as possible to the expected one (for the external crystal, this is 32768 Hz). It is possible to change the pin on which the lp clock is measured, however, pins P1.0, P1.5, and P1.7 might affect radio performance if toggled while there is RF activity. Therefore, it is recommended to use them at low speed and not while radio is active.

5.2 Configuring The Demo Code

1. Establish a connection between the target device and your PC through the **USB2(DBG)** port of the motherboard. This port is used both for powering and communicating with the DA1468x SoC. For this demonstration, a Pro DevKit is used.

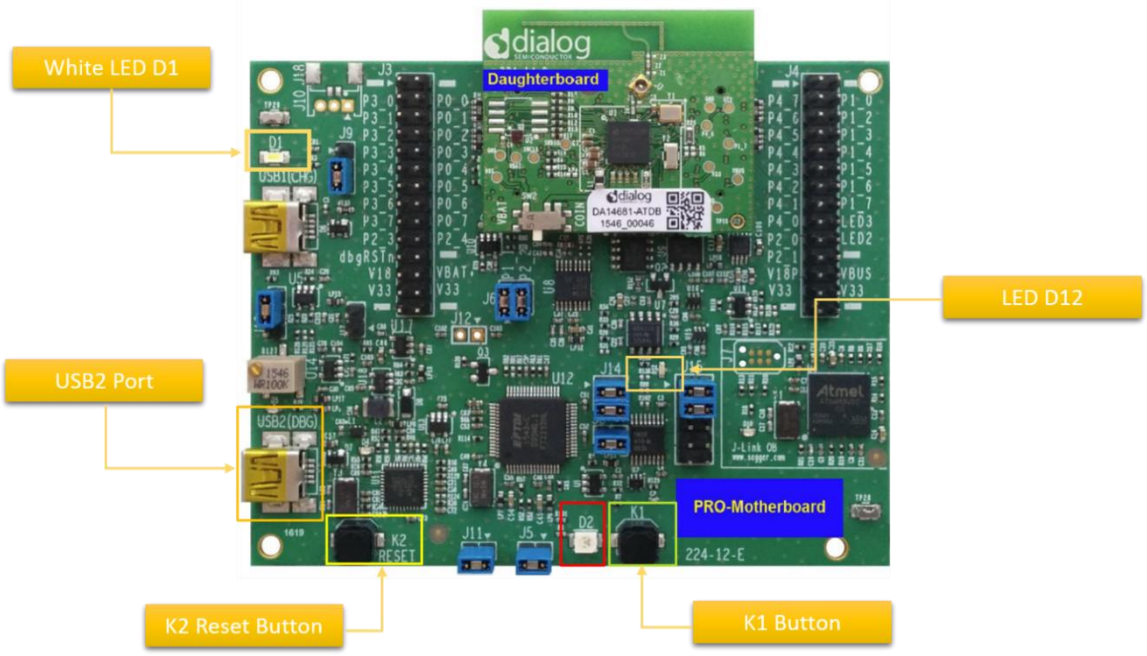

Figure 12: DA1468x Pro DevKit

2. Import and then make a copy of the **freertos_retarget** sample code found in the SDK of the DA1468x family of devices.

Note: It is essential to import the folder named scripts to perform various operations (including building, debugging, and downloading)

3. In the target application, add/modify all the required code blocks as illustrated in the Code Overview section.

Note: It is possible for the defined macros not to be taken into consideration instantly. Thus, resulting in errors during compile time. If this is the case, the easiest way to deal with the issue is to: right-click on the application folder, select Index > Rebuild and then Index > Freshen All Files.

4. In sw_rtc_task.c, declare time parameters of your choice. These values will be the starting point for all of the subsequent RTC operations.

```
/*
 * Initialize calendar with preferred values.
 * Date format: Day - Month - Year
 * Time format: Hours - Minutes - Seconds - Milliseconds
 */
__RETAINED_RW sw_rtc_calendar_t cal = {
        /* Declare all parameters for the [date] */
        .date = {
                .year       = 2018,
                .month      = 4,
                .month_date = 20,
        },
        /* Declare all parameters for the [time] */
```

```
        .time = {
                .hour   = 16,
                .min    = 35,
                .sec    = 0,
                .msec   = 0,
        }
};
```

5. Optionally, enable/disable debugging messages on the serial console (enabled by default):

```
#define SW_RTC_DBG_MESSAGES_CONSOLE  1
```

6. In alert_task.c, declare alert parameters of your choice.

```
/*
 * Declare parameters for the alert events.
 *
 * \note: There are three distinct alert classes which can be enabled-disabled-
 *        configured individually.
 *
 */
__RETAINED_RW time_alert_t my_alert = {
        .hour     = 1,          // Configure a time interval expressed in hour

        .min      = 0,          // Configure a time interval expressed in min

        .sec      = 0,          // Configure a time interval exoressed in sec

        .repeat_h = true,      // If enabled, alert for hours becomes repeatable

        .repeat_m = false,     // If enabled, alert for hours becomes repeatable

        .repeat_s = false,     // If enabled, alert for hours becomes repeatable

};
```

7. Optionally, enable debugging messages on the serial console (enabled by default):

```
#define TIME_ALERT_DBG_MESSAGES_CONSOLE   1
```

8. Users may also enable the breath timer and the white LED1 of the chip (disabled by default):

```
#define TIME_ALERT_BREATH_TIMER_EN 0
```

9. Build the project in either **Debug_QSPI** or **Release_QSPI** mode and burn the generated image to the chip (either via the serial or jtag port).

10. Press the **K2** button on Pro DevKit to reset the device. This is required in order for the chip to start executing its firmware.

5.3 Verifying The Demo Code

1. Press the **K1** button on Pro DevKit. If enabled, a debugging message is displayed on the console indicating the current date and time.

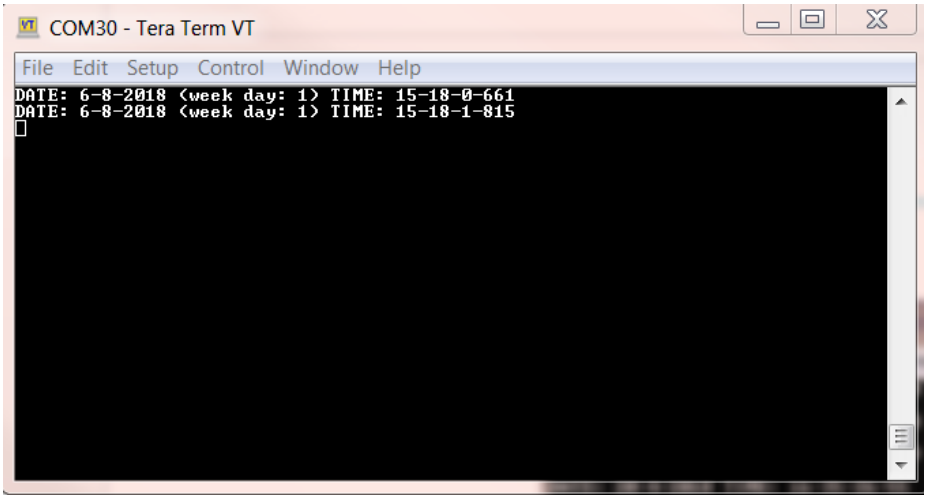

Figure 13: Debugging Messages Indicating the Current Date and Time.

2. Upon an alert event and given that debugging messages have been enabled, a message will be displayed on the serial console indicating the source of the alert. If breath timer has also been enabled, then the white LED1 on Pro DevKit will start blinking. At any time, it can be disabled by pressing the K1 button.

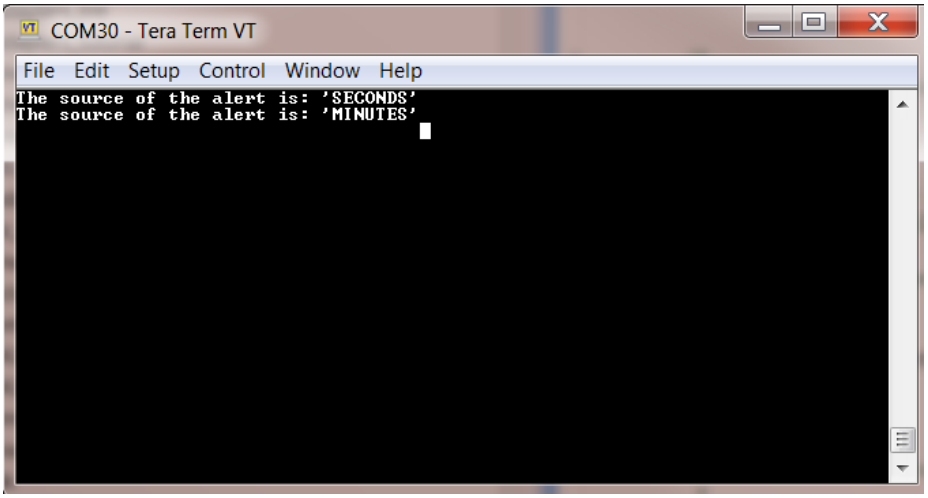

Figure 14: Debugging Messages Indicating the Alert Source

6 Code Overview

This section provides the code blocks needed to successfully execute this tutorial.

6.1 Header Files

In **main.c**, add the following header files:

```
/*
 * Required header files for the main.c source file
 */
#include "hw_wkup.h"

#include "sw_rtc_date_time.h"
#include "alert_mechanism.h"
```

6.2 System Init Code

In **main.c**, replace system_init() with the following code:

```
/* Task priorities */
#define mainRTC_TASK_PRIORITY      ( OS_TASK_PRIORITY_NORMAL )
#define mainALERT_TASK_PRIORITY  ( OS_TASK_PRIORITY_NORMAL )



/******** Notification Bitmasks *************/
#define SW_RTC_TRIG_NOTIF   (1 << 1)


/*
 * Macro for outputting the low power (lp) clock on P3.0 pin.
 * 1: configures P3.0 pin to output the lp clock.
 * 0: nothing
 *
 * \note: This macro is intended for debugging purposes only in order to measure
 *          the accuracy of the lp clock used. The device should stay active for
 *          as long as measurements on P3.0 pin are performed.
 */
#define TEST_LP_CLOCK_OUTPUT 0


/*
 * Functions prototypes
 */
void prvRTC_Task( void *pvParameters );
void prvALERT_Task( void *pvParameters );

extern void time_alert_led_blink_stop(void);


/* Task handlers */
static OS_TASK task_rtc_h   = NULL;
static OS_TASK task_alert_h = NULL;
```

```
static void system_init( void *pvParameters )
{
    OS_TASK task_h = NULL;

#if defined CONFIG_RETARGET
    extern void retarget_init(void);
#endif

    /*
     * Prepare clocks. Note: cm_cpu_clk_set() and cm_sys_clk_set()
     * can be called only from a task since they will suspend the
     * task until the  XTAL16M has settled and, maybe, the PLL is
     * locked.
     */
    cm_sys_clk_init(sysclk_XTAL16M);
    cm_apb_set_clock_divider(apb_div1);
    cm_ahb_set_clock_divider(ahb_div1);
    cm_lp_clk_init();

    /* Prepare the hardware to run this demo. */
    prvSetupHardware();

    /* init resources */
    resource_init();

#if defined CONFIG_RETARGET
    retarget_init();
#endif


#if (TEST_LP_CLOCK_OUTPUT == 1)
    /* Set the device to stay in active mode. */
    pm_set_sleep_mode(pm_mode_active);

    /*
     * Select which clock to map when PID = FUNC_CLOCK.
     *
     * Valid values are:
     * [0x00]: When XTAL32K is the selected lp clock
     * [0x02]: When RCX is the selected lp clock
     */
    REG_SETF(GPIO, GPIO_CLK_SEL, FUNC_CLOCK_SEL, 0x00);
#else
    /* Set the desired sleep mode. */
    pm_set_sleep_mode(pm_mode_extended_sleep);
#endif


  /* Start main task here */
  OS_TASK_CREATE( "Template",            /* The text name assigned to the task,
                                            for debug only; not used by the
                                            kernel. */
            prvTemplateTask,        /* The function that implements the
                                       task. */
            NULL,                /* The parameter passed to the task */
```

```
                200 * OS_STACK_WORD_SIZE,    /* The number of bytes to allocate to
                                                the stack of the task. */
                mainTEMPLATE_TASK_PRIORITY,  /* The priority assigned to the task */
                task_h );                    /* The task handle */
    OS_ASSERT(task_h);

    /* Suspend task execution */
    OS_TASK_SUSPEND(task_h);


     /*
      * Task responsible for performing RTC related
      * operations
      */
     OS_TASK_CREATE( "RTC",

                 prvRTC_Task,
                 NULL,
                 300 * OS_STACK_WORD_SIZE,

                 mainRTC_TASK_PRIORITY,
                 task_rtc_h );
     OS_ASSERT(task_rtc_h);


     /*
      * Task responsible for performing alert related
      * operations
      */
     OS_TASK_CREATE( "ALERT",

                 prvALERT_Task,
                 NULL,
                 300 * OS_STACK_WORD_SIZE,

                 mainALERT_TASK_PRIORITY,
                 task_alert_h );
     OS_ASSERT(task_alert_h);

     /* The work of the SysInit task is done */
     OS_TASK_DELETE( xHandle );

}
```

6.3 Wake-Up Timer Code

In **main.c**, add the following code for handling external events, after periph_init().

```
/*
 * Callback function to be called after an external event
 * is generated, that is after K1 button on the Pro DevKit is
```

```
 * pressed.
 */
void wkup_cb(void)
{
      /*
       * This function must be called by any user-defined
       * interrupt callback, to clear the interrupt flag.
       */
      hw_wkup_reset_interrupt();

     /*
      * Notify the [prvRTC_Task] task that time for performing
      * an RTC operation has elapsed
      */
      OS_TASK_NOTIFY_FROM_ISR(task_rtc_h, SW_RTC_TRIG_NOTIF,
                                                OS_NOTIFY_SET_BITS);

      /*
       * Disable breath timer and resume the device to its
       * initial state, that is extended sleep mode!
       */
      time_alert_led_blink_stop();

}

/*
 * Function which makes all the necessary initializations for the
 * wake-up controller
 */
static void init_wkup(void)
{
      /*
       * This function must be called first and is responsible
       * for the initialization of the hardware block
       */
      hw_wkup_init(NULL);

      /*
       * Configure the pin(s) that can trigger the device to
       * wake up while in sleep mode. The last input parameter determines
       * the triggering edge of the pulse (event)
       */
      hw_wkup_configure_pin(HW_GPIO_PORT_1, HW_GPIO_PIN_6,
                                      true, HW_WKUP_PIN_STATE_LOW);

      /*
       * This function defines a delay between the moment at which
       * a trigger event is present on a pin and the moment at which the controller
       * takes this event into consideration. Setting debounce time to 0
       * hardware debouncing mechanism is disabled. Maximum debounce time
       * is 63 ms.
       */
      hw_wkup_set_debounce_time(10);

// Check if the chip is either DA14680 or 81
```

```
#if dg_configBLACK_ORCA_IC_REV == BLACK_ORCA_IC_REV_A

    /*
     * Set threshold for event counter. Interrupt is generated after
     * the event counter reaches the configured value. This function
     * is only supported in DA14680/1 chips.
     */
    hw_wkup_set_counter_threshold(1);
#endif

    /* Register interrupt handler */
    hw_wkup_register_interrupt(wkup_cb, 1);
}
```

6.4 Hardware Initialization

In **main.c**, replace both **periph_init()** and **prvSetupHardware()** with the following code for configuring pins after power-up/wake-up:

```
/**
 * @brief Initialize the peripherals domain after power-up.
 *
 */
static void periph_init(void)
{
#   if dg_configBLACK_ORCA_MB_REV == BLACK_ORCA_MB_REV_D
#       define UART_TX_PORT    HW_GPIO_PORT_1
#       define UART_TX_PIN     HW_GPIO_PIN_3
#       define UART_RX_PORT    HW_GPIO_PORT_2
#       define UART_RX_PIN     HW_GPIO_PIN_3
#   else
#       error "Unknown value for dg_configBLACK_ORCA_MB_REV!"
#   endif


    hw_gpio_set_pin_function(UART_TX_PORT, UART_TX_PIN,
            HW_GPIO_MODE_OUTPUT, HW_GPIO_FUNC_UART_TX);
    hw_gpio_set_pin_function(UART_RX_PORT, UART_RX_PIN,
            HW_GPIO_MODE_INPUT, HW_GPIO_FUNC_UART_RX);

#if (TEST_LP_CLOCK_OUTPUT == 1)
    // Output the selected lp clock on P3.0 pin - Intended for debugging purposes only.
    hw_gpio_set_pin_function(HW_GPIO_PORT_3, HW_GPIO_PIN_0,
                    HW_GPIO_MODE_OUTPUT, HW_GPIO_FUNC_CLOCK);
#endif

}

/**
 * @brief Hardware Initialization
 */
static void prvSetupHardware( void )
```

```
{
    /* Init hardware */
    pm_system_init(periph_init);
    init_wkup();
}
```

6.5 Task Code for Alert Operations

Code snippet of **prvRTC_Task** task, responsible for performing RTC related operations. Create a new source file, for example sw_rtc_task.c and add the following code:

```
#include <string.h>
#include <stdio.h>

#include "sw_rtc_date_time.h"
#include "osal.h"


/******** Notification Bitmasks *************/
#define SW_RTC_TRIG_NOTIF       (1 << 1)


/*
 * Macro for enabling/disabling debugging messages on the serial console:
 *
 * 1: enables debugging messages
 * 0: disables debugging messages
 */
#define SW_RTC_DBG_MESSAGES_CONSOLE   1


 /*
  * Structure that holds date/time related info
  */
__RETAINED_RW sw_rtc_date_time_t date_time = {
    /*
     * Declare all the info related to date.
     *
     * \note: The weekday is calculated automatically by the app.
     *
     * \note: The app performs sanity checks to identify values out of boundaries.
     *          For any invalid value, the code gets stuck in assertions!
     *
     */
    .date = {
        .year      = 2018,
        .month     = 9,
        .month_date = 9,
    },
    /*
     * Declare all the info related to time.
```

```
         *
         * \note: The max time resolution is microseconds (us) */
        .time = {
                .hour = 0,
                .min  = 0,
                .sec  = 0,
        }
};



/*
 * Task responsible for performing RTC operations
 */
void prvRTC_Task( void *pvParameters )
{
        char calendar_str[60];

        /*
         * Initialize the RTC mechanism. This should be the first invoked
         * function before any other related SW RTC API.
         */
        sw_rtc_init();


        /* Set the current date */
        sw_rtc_date_write(&date_time.date);

        /* Set the current time */
        sw_rtc_time_write(&date_time.time);

        /*
         * Instead of declaring date/time individually, sw_rtc_date_time_write()
         * can be used to set both of them at once:
         */


        memset((uint8_t *)&date_time, 0x00, sizeof(sw_rtc_date_time_t));

        for(;;) {

                OS_BASE_TYPE ret;
                uint32_t notif;

                /*
                 * Block forever waiting for any of the notification bits,
                 * then clear them all. In this example at every K1 button
                 * press (on pro DevKit)
                 */
                ret = OS_TASK_NOTIFY_WAIT(0, OS_TASK_NOTIFY_ALL_BITS,
                                          &notif, OS_TASK_NOTIFY_FOREVER);

                /*
                 * Thus, the return value must always be OS_OK
                 */
```

```
            OS_ASSERT(ret == OS_OK);

            /*
             * Has time for updating the calendar elapsed?
             */
            if (notif & SW_RTC_TRIG_NOTIF) {

                /* Read the current date */
                sw_rtc_date_read(&date_time.date);

                /* Read the current time */
                sw_rtc_time_read(&date_time.time);


                /*
                 * Instead of reading date/time individually, sw_rtc_date_time_read()
                 * can be used to read both of them at once:
                 */


                /* If enabled, print the current date/time on the serial console */
                if (SW_RTC_DBG_MESSAGES_CONSOLE) {

                    /* Clear the buffer */
                    memset(calendar_str, '\0', sizeof(calendar_str));

                    /*
                     * Prepare the debugging message (current date/time)
                     */
                    sprintf(calendar_str,
                        "DATE: %lu-%lu-%lu (week day: %lu) TIME: %lu-%lu-%lu-%lu\n\r",
                         date_time.date.month_date,  date_time.date.month,
                         date_time.date.year,        date_time.date.week_date,
                         date_time.time.hour,        date_time.time.min,
                         date_time.time.sec,         date_time.time.msec );


                    /* Send the message on the serial console */
                    printf("%s", calendar_str);
                    fflush(stdout);
                }

            }

        } // end of main for(;;) loop
    } // end of task
```

6.6 Task Code for Alert Operations

Code snippet of **prvALERT_Task** task, responsible for performing alert related operations. Create a new source file, for example alert_task.c and add the following code:

```
#include <string.h>
#include <stdio.h>

#include "alert_mechanism.h"
#include "osal.h"

#include <stdbool.h>

#include "sys_power_mgr.h"
#include "hw_led.h"
#include "hw_breath.h"

#include "sw_rtc_date_time.h"



/* Handle of [prvALERT_Task] task */
__RETAINED_RW OS_TASK alert_task_h = NULL;

/******** Notification Bitmasks *************/
#define TIME_ALERT_TRIG_NOTIF_H  (1 << 1)
#define TIME_ALERT_TRIG_NOTIF_M  (1 << 2)
#define TIME_ALERT_TRIG_NOTIF_S  (1 << 3)


/*
 * Macro for enabling/disabling debugging messages on the serial console:

 * 1: enables debugging messages
 * 0: disables debugging messages
 */
#define TIME_ALERT_DBG_MESSAGES_CONSOLE    1

/*
 * Macro for enabling/disabling blinking functionality upon an alert event:

 * 1: enables blinking functionality
 * 0: disables blinking functionality
 */
#define TIME_ALERT_BREATH_TIMER_EN        0



/*
 * Keep track of device state (active/sleep)
 */
__RETAINED_RW static bool old_alerting = false;

/* Function prototypes */
void time_alert_set_mode(bool new_alerting);



/*
 * Turn on LED1 on Pro DevKit (blinking functionality).
```

```
 * LED1 is driven by breath timer.
 */
void time_alert_led_blink_start(breath_config *breath_cfg)
{
	/* Force the device to stay in active mode */
	time_alert_set_mode(true);

	/* Configure the breath timer */
	hw_breath_init(breath_cfg);

	/* Configure the white LED1 */
	hw_led_set_led1_src(HW_LED_SRC1_BREATH);
	hw_led_enable_led1(true);

	/* Enable the breath timer */
	hw_breath_enable();
}


/* Turn off LED1 on Pro DevKit */
void time_alert_led_blink_stop(void)
{
	/* Resume device to its initial state */
	time_alert_set_mode(false);

	/* Disable breath timer */
	hw_breath_disable();
}


/*
 * Check whether the device should be forced to stay
 * awake or it's time to resume to its initial state.
 *
 * \note Number of calls to pm_stay_alive() must match the
 *       number of calls to pm_resume_sleep()!!!!!!!
 */
void time_alert_set_mode(bool new_alerting)
{
	/* First check if a new alert flag has been assigned */
	if (old_alerting != new_alerting) {

		/* If flag is set to true then force the device to stay alive */
		if (new_alerting) {
			pm_stay_alive();

		/*
		 * Otherwise the new value is zero and thus the device should
		 * resume to its initial state.
		 */
		} else {
			pm_resume_sleep();
		}

		/* Make a copy of the new alert flag */
```

```
            old_alerting = new_alerting;
        }
}


/*
 * Structure that holds alert related parameters
 *
 * \note: There are three distinct alert classes which can be
 *          enabled-disabled-configured individually.
 *
 */
__RETAINED_RW time_alert_t my_alert = {
        .hour    = 4,   // Set alerts expressed in hours   (every x hours  ).
        .min     = 2,   // Set alerts expressed in minutes (every x minutes).
        .sec     = 5,   // Set alerts expressed in seconds (every x seconds).

        .repeat_h = false,  // If set to false, alerts are triggered once, repeatedly otherwise.
        .repeat_m = true,  // If set to false, alerts are triggered once, repeatedly otherwise.
        .repeat_s = false,  // If set to false, alerts are triggered once, repeatedly otherwise.
};


/*
 * User-defined callback function, called upon an alert event.
 * It should have that specific prototype!
 *
 * \note: All the alert classes will call the same callback. The \p source parameter
 *          should be used to identify the source of the event.
 *
 */
void my_alert_cb(uint8_t source)
{

        /*
         * Check whether Hours Alert Class triggered the alert.
         * Then notify the main task accordingly.
         */
        if (source & ALERT_SOURCE_HOURS_Msk) {

                OS_TASK_NOTIFY(alert_task_h, TIME_ALERT_TRIG_NOTIF_H,
                                        OS_NOTIFY_SET_BITS);
        }

        /*
         * Check whether Minutes Alert Class triggered the alert.
         * Then notify the main task accordingly.
         */
        if (source & ALERT_SOURCE_MINUTES_Msk) {

                OS_TASK_NOTIFY(alert_task_h, TIME_ALERT_TRIG_NOTIF_M,
                                        OS_NOTIFY_SET_BITS);
        }

        /*
```

```
        * Check whether Seconds Alert Class triggered the alert.
        * Then notify the main task accordingly.
        */
       if (source & ALERT_SOURCE_SECONDS_Msk) {

            OS_TASK_NOTIFY(alert_task_h, TIME_ALERT_TRIG_NOTIF_S,
                                        OS_NOTIFY_SET_BITS);
       }
}


/*
 * Task responsible for performing alert operations.
 */
void prvALERT_Task( void *pvParameters )
{

       /* Get the handler of the current task */
       alert_task_h = OS_GET_CURRENT_TASK();


       /*
        * Initialize the alert mechanism. This function should be invoked
        * before any other related API.
        */
       time_alert_init();


       /*
        * Register callback function that will be called upon an alert event.
        */
       time_alert_register_cb(my_alert_cb);


       /*
        * Set alerts, expressed in minutes.
        *
        * \note If repeat flag is set to true, alerts are triggered every x minutes
        *
        * \note If a new alert configuration is performed before the execution of the
        *          current alert, the old alert will be discarded and the new one will be taken
        *         into consideration.
        *
        * \note   At any time, alerts can be disabled by calling time_alert_class_clear()
        *
        */
       time_alert_class_set(my_alert.min, my_alert.repeat_m, SET_ALERT_CLASS_Min);


       /*
        * Set alerts expressed in hours. Configure an alert to be triggered after 1 hour.
        *
        * \note If repeat flag is set to true, alerts are triggered every x hours
        *
        * \note If a new alert configuration is performed before the execution of the
```

```
         *          current alert, the old alert will be discarded and the new one will be taken
         *         into consideration.
         *
         * \note  At any time, alerts can be disabled by calling time_alert_class_clear()
         *
         */
        time_alert_class_set(my_alert.hour, my_alert.repeat_h, SET_ALERT_CLASS_Hour);


        /*
         * Set alerts expressed in seconds
         *
         * \note If repeat flag is set to true, alerts are triggered every x seconds
         *
         * \note If a new alert configuration is performed before the execution of the
         *         current alert, the old alert will be discarded and the new one will be taken
         *        into consideration.
         *
         * \note At any time, alerts can be disabled by calling time_alert_class_clear()
         *
         */
        time_alert_class_set(my_alert.sec, my_alert.repeat_s, SET_ALERT_CLASS_Sec);


#if (TIME_ALERT_BREATH_TIMER_EN == 1)
        /*
         * Breath timer parameters
         */
        breath_config breath_cfg = {
            .dc_min   = 0,
            .dc_max   = 255,
            .freq_div = 255,
            .dc_step  = 96,
            .polarity = HW_BREATH_PWM_POL_POS
        };
#endif

        for (;;) {

            OS_BASE_TYPE ret;
            uint32_t notif;

            /*
             * Block forever waiting for any of the notification bits,
             * then clear them all
             */
            ret = OS_TASK_NOTIFY_WAIT(0, OS_TASK_NOTIFY_ALL_BITS,
                                      &notif, OS_TASK_NOTIFY_FOREVER);

            /*
             * Thus, the return value must always be OS_OK
             */
            OS_ASSERT(ret == OS_OK);
```

```
        /*
         * This code block is executed upon an alert event
         * produced by Hours Alert Class.
         */
        if (notif & TIME_ALERT_TRIG_NOTIF_H) {

#if (TIME_ALERT_BREATH_TIMER_EN == 1)
            /* Start blinking white LED D1 on pro DevKit */
            time_alert_led_blink_start(&breath_cfg);
#endif

            /* Print out a debugging message */
            if (TIME_ALERT_DBG_MESSAGES_CONSOLE) {
                printf("The source of the alert is: 'HOURS'\n\r");
            }
        }


        /*
         * This code block is executed upon an alert event
         * produced by Minutes Alert Class.
         */
        if (notif & TIME_ALERT_TRIG_NOTIF_M) {

#if (TIME_ALERT_BREATH_TIMER_EN == 1)
            /* Start blinking white LED D1 on pro DevKit */
            time_alert_led_blink_start(&breath_cfg);
#endif

            /* Print out a debugging message */
            if (TIME_ALERT_DBG_MESSAGES_CONSOLE) {
                printf("The source of the alert is: 'MINUTES'\n\r");
            }
        }


        /*
         * This code block is executed upon an alert event
         * produced by Seconds Alert Class.
         */
        if (notif & TIME_ALERT_TRIG_NOTIF_S) {

#if (TIME_ALERT_BREATH_TIMER_EN == 1)
            /* Start blinking white LED D1 on pro DevKit */
            time_alert_led_blink_start(&breath_cfg);
#endif


            /* Print out a debugging message */
            if (TIME_ALERT_DBG_MESSAGES_CONSOLE) {
                printf("The source of the alert is: 'SECONDS'\n\r");
            }

        }
    }
```

```
}
```

6.7 Date/Time Implementation Source File

Code snippet of date/time mechanism implementation. Create a new source file, for example sw_rtc_date_time.c and add the following code:

```
#include "sw_rtc_date_time.h"

#include <stdbool.h>
#include <stdlib.h>

#include <math.h>
#include "osal.h"

#include "sys_rtc.h"
#include "sys_power_mgr.h"

#include <string.h>
#include <stdio.h>



/* Months of a year */
typedef enum {
      JANUARY   = 1,
      FEBRUARY  = 2,
      MARCH     = 3,
      APRIL     = 4,
      MAY       = 5,
      JUNE      = 6,
      JULY      = 7,
      AUGUST    = 8,
      SEPTEMBER = 9,
      OCTOBER   = 10,
      NOVEMBER  = 11,
      DECEMBER  = 12
} SW_RTC_MONTH;


/*
 * Set this macro to perform a forced RTC measurement via an OS Timer.
 */
#define SW_RTC_FORCED_DATE_TIME_EN    1


/* Mutex handle */
__RETAINED_RW static OS_MUTEX sw_rtc_mutex = NULL;


/*
 * Time interval, expressed in milliseconds, for triggering a forced RTC operation.
```

```
 * This macro is used only when SW_RTC_FORCED_DATE_TIME_EN is set. Default value
 * is 1 day, that is:
 *
 * (24 hrs * 60 minutes/hour * 60 seconds/minute * 1000 milliseconds/second) =
 * 86,400,000
 *
 */
#define SW_RTC_FORCED_DATE_TIME_IN_MS   86400000


/*
 * Variables used to store the lp clocks, using the old/new scheme
 */
__RETAINED_RW static uint64_t sw_rtc_new = 0;
__RETAINED_RW static uint64_t sw_rtc_old = 0;


/*
 * SW RTC parameters. This structure holds the current date/time.
 */
__RETAINED_RW static sw_rtc_date_time_t *calendar;



#if (SW_RTC_FORCED_DATE_TIME_EN == 1)

/* OS timer handle */
static OS_TIMER sw_rtc_timer_h;


/* Callback function called after OS timer's expiration */
void sw_rtc_trg_cb(OS_TIMER timer)
{
      OS_BASE_TYPE ret;

      /*
       * Try to get the mutex without blocking. If the mutex was successfully acquired
       * then  no other tasks perform RTC measurements. If not, another task is already
       * performing RTC operations and it's OK to abort the operation.
       *
       * \warning Blocking within timer's callback is not permitted!!!
       *
       */
      ret = OS_MUTEX_GET(sw_rtc_mutex, OS_MUTEX_NO_WAIT);

      if (ret == OS_MUTEX_TAKEN) {
            /* Allocate memory both date/time */
            sw_rtc_date_time_t *date_time = (sw_rtc_date_time_t *)
                                          OS_MALLOC(sizeof(sw_rtc_date_time_t));

            /* Just perform a read date/time operation */
            sw_rtc_date_time_read(date_time);

            /* Release the previously allocated memory */
            OS_FREE(date_time);
```

```
            /* Release the already acquired mutex */
            OS_MUTEX_PUT(sw_rtc_mutex);

        }
}

#endif


/*
 * Initialize the SW RTC mechanism. First, memory for the SW RTC parameters
 * is allocated and then an OS timer responsible for performing forced RTC
 * operations is created.
 *
 */
void sw_rtc_init()
{
        /* Allocate memory for the calendar (date/time) structure */
        calendar = (sw_rtc_date_time_t *) OS_MALLOC(sizeof(sw_rtc_date_time_t));
        OS_ASSERT(calendar);

        /* Clear the calendar structure */
        memset((uint8_t *)calendar, 0x00, sizeof(sw_rtc_date_time_t));


#if (SW_RTC_FORCED_DATE_TIME_EN == 1)

        uint64_t timer_ticks = (((uint64_t)SW_RTC_FORCED_DATE_TIME_IN_MS *
                                                        configTICK_RATE_HZ) / 1000);

        /*
         * Create an OS timer to trigger a forced RTC operation.
         */
        sw_rtc_timer_h = OS_TIMER_CREATE("RTC_Trigger",
                        (uint32_t)timer_ticks, OS_TIMER_SUCCESS,
                                        (void *) OS_GET_CURRENT_TASK(), sw_rtc_trg_cb);
        OS_ASSERT(sw_rtc_timer_h);

        /* Start the OS timer */
        OS_TIMER_START(sw_rtc_timer_h, OS_TIMER_FOREVER);
#endif

        /* Create mutex for RTC operations */
        OS_MUTEX_CREATE(sw_rtc_mutex);
        OS_ASSERT(sw_rtc_mutex != NULL);

}


/* Reset calendar values to zero */
void sw_rtc_date_time_reset(void)
{
        /* Reset all the calendar related parameters */
        sw_rtc_new = 0;
```

```
        sw_rtc_old = 0;

        /* Reset date parameters */
        calendar->date.year      = 0;
        calendar->date.month     = 1;
        calendar->date.month_date = 1;

        /* Reset time parameters */
        calendar->time.hour = 0;
        calendar->time.min  = 0;
        calendar->time.sec  = 0;
        calendar->time.msec = 0;
        calendar->time.usec = 0;
}


/*
 * Get the number of days of a month.
 */
uint8_t sw_rtc_get_days_of_month(void)
{
        uint8_t days = 0;

        /* Given the current month, calculate its maximum days */
        switch(calendar->date.month) {

        case JANUARY:
                days = 31;
                break;
        case FEBRUARY:
                /*
                 * Check if the year is a leap year, taking into consideration
                 * the following three criteria:
                 * 1. the year can be evenly divided by 4
                 * 2. the year is divisible by 400
                 * 3. the year is not divisible by 100
                 */
                if ((!(calendar->date.year % 4)) ||
                        ((!(calendar->date.year % 400)) && (calendar->date.year % 100))) {
                        days = 29;
                } else {
                        days = 28;
                }
                break;
        case MARCH:
                days = 31;
                break;
        case APRIL:
                days = 30;
                break;
        case MAY:
                days = 31;
                break;
        case JUNE:
                days = 30;
```

```
            break;
        case JULY:
            days = 31;
            break;
        case AUGUST:
            days = 31;
            break;
        case SEPTEMBER:
            days = 30;
            break;
        case OCTOBER:
            days = 31;
            break;
        case NOVEMBER:
            days = 30;
            break;
        case DECEMBER:
            days = 31;
            break;
        default:
            OS_ASSERT(0); // Invalid value!
        }

        return days;
}



/* Check whether the given year is a leap year */
bool sw_rtc_is_leap_year(void)
{
        /*
         * Check if the year is a leap year, taking into consideration
         * the following three criteria:
         * 1. the year can be evenly divided by 4
         * 2. the year is divisible by 400
         * 3. the year is not divisible by 100
         */
        if ((!(calendar->date.year % 4)) ||
            ((!(calendar->date.year % 400)) && (calendar->date.year % 100))) {
            return true;
        } else {
            return false;
        }
}


/* Calculate the month code used in weekday computations */
uint8_t sw_rtc_compute_month_code(void)
{
        uint8_t month_code;

        switch(calendar->date.month) {
```

```
        case JANUARY:
                month_code = 0;
                break;
        case FEBRUARY:
                month_code = 3;
                break;
        case MARCH:
                month_code = 3;
                break;
        case APRIL:
                month_code = 6;
                break;
        case MAY:
                month_code = 1;
                break;
        case JUNE:
                month_code = 4;
                break;
        case JULY:
                month_code = 6;
                break;
        case AUGUST:
                month_code = 2;
                break;
        case SEPTEMBER:
                month_code = 5;
                break;
        case OCTOBER:
                month_code = 0;
                break;
        case NOVEMBER:
                month_code = 3;
                break;
        case DECEMBER:
                month_code = 5;
                break;
        default:
                OS_ASSERT(0); // Invalid value!
        }

        return month_code;
}

/* Calculate the century code used in weekday computations */
uint8_t sw_rtc_compute_century_code(void)
{
        uint8_t century_code = 0;

        if (calendar->date.year >= 2000 && calendar->date.year < 2100) {
                century_code = 6;
        } else if (calendar->date.year >= 2100 && calendar->date.year < 2200) {
                century_code = 4;
        } else if (calendar->date.year >= 2200 && calendar->date.year < 2300) {
                century_code = 2;
        } else if (calendar->date.year >= 2300 && calendar->date.year < 2400) {
```

```
            century_code = 0;
        }

        return century_code;
}

/* Function responsible for computing the current day of the week */
uint32_t sw_rtc_compute_date_of_week(void)
{
    /* Isolate the last two digits from the year e.g. 2018 -> 18 */
    uint8_t year_code = (uint8_t)(calendar->date.year % 100);

    /* Compute the year code */
    year_code = ((year_code + (year_code / 4)) % 7);

    /* Compute the month code */
    uint8_t month_code = sw_rtc_compute_month_code();

    /* Compute the century code */
    uint8_t centure_code = sw_rtc_compute_century_code();

    /* Check whether the year is a leap year */
    uint8_t leap_year = (uint8_t)sw_rtc_is_leap_year();

    /* Add all the codes */
    uint8_t day_week = (year_code + month_code + centure_code +
                                        (uint8_t)calendar->date.month_date);


    if (leap_year && ((calendar->date.month == JANUARY) ||
                                    (calendar->date.month == FEBRUARY)) ) {
        return (uint32_t)((day_week - 1) % 7);
    } else {
        return (uint32_t)(day_week % 7);
    }

}



/*
 * Function responsible for converting time, expressed in microseconds (us), into full
 * calendar (date/time).
 */
void sw_rtc_convert_time_to_calendar(uint64_t usec_time)
{
    calendar->time.usec += usec_time;

    /*
     * Check if there is an overflow in [microseconds],
     * that is greater than 999.
     */
    if (calendar->time.usec > 999) {
        /* Compute the right next greater unit, that is milliseconds */
        calendar->time.msec += (uint32_t)(calendar->time.usec / 1000);
```

```
/* Then compute the new value for [microseconds] */
calendar->time.usec %= 1000;

/*
 * Check if there is an overflow in [milliseconds],
 * that is greater than 999.
 */
if (calendar->time.msec > 999) {
    /* Compute the right next greater unit, that is [seconds] */
    calendar->time.sec += (calendar->time.msec / 1000);

    /* Then compute the new value for [milliseconds] */
    calendar->time.msec %= 1000;

    /*
     * Check if there is an overflow in [seconds],
     * that is greater than 59
     */
    if (calendar->time.sec > 59) {
        /* Compute the right next greater unit, that is [minutes] */
        calendar->time.min += (calendar->time.sec / 60);

        /* Then compute the new value for [seconds] */
        calendar->time.sec %= 60;

        /*
         * Check is there is an overflow in [minutes],
         * that is greater than 59
         */
        if (calendar->time.min > 59) {
            /* Compute the right next greater unit, that is [hours] */
            calendar->time.hour += (calendar->time.min / 60);

            /* Then compute the new value for [minutes] */
            calendar->time.min %= 60;

            /*
             * Check if there is an overflow in hours,
             * that is greater than 23
             */
            if (calendar->time.hour > 23) {
                /* Compute the right next greater unit, that is [days] */
                calendar->date.month_date += (calendar->time.hour / 24);

                /* Then compute the new value for [hours] */
                calendar->time.hour %= 24;

                /*
                 * Check if there is an overflow in days,
                 * that is greater than 28/29/30 or 31
                 * depending on the current month.
                 */
                if (calendar->date.month_date >
                        sw_rtc_get_days_of_month()) {
```

```
                                /* Calculate months and days */
                                do {
                                    calendar->date.month_date -=
                                                sw_rtc_get_days_of_month();
                                    calendar->date.month++;

                                    /*
                                     * Check if there is an overflow in [months],
                                     * that is greater than 12
                                     */
                                    if (calendar->date.month > 12) {
                                        calendar->date.year +=
                                            (calendar->date.month / 12);
                                        calendar->date.month %= 12;
                                    }

                                } while (calendar->date.month_date >
                                            sw_rtc_get_days_of_month());


                            } // end of days monitoring
                        } // end of hours monitoring
                    } // end of minutes monitoring
                } // end of seconds monitoring
            } // end of milliseconds monitoring
        } // end of microseconds monitoring

        /* Finally compute the day of the week */
        calendar->date.week_date = sw_rtc_compute_date_of_week();

} // end of function



/*
 * Convert low power (lp) clock cycles into time - expressed in microseconds.
 */
uint64_t sw_rtc_convert_lp_to_time(uint64_t lp_clocks)
{
        uint64_t time = 0;

        /*
         * Check whether the lp clock source is either the external crystal XTAL32K
         * or an external clock source.
         */
        if ((dg_configUSE_LP_CLK == LP_CLK_32768) ||
                        (dg_configUSE_LP_CLK == LP_CLK_32000)) {

                /*
                 * Compute the period of the lp clock source - expressed in microseconds (us)
                 *
                 * \note Multiply by [1024] to increase accuracy!
                 *
```

```
            */
            const uint32_t lp_clk_period = ((1000000UL * 1024)
                                    /configSYSTICK_CLOCK_HZ);

            /*
             * Compute the time that has elapsed.
             */
            time = (lp_clocks * (uint64_t)lp_clk_period);
            time = (time >> 10); // divide with 1024

        /* Check whether the lp clock source is the internal RCX */
        } else if (dg_configUSE_LP_CLK == LP_CLK_RCX) {

            /*
             * Use [rcx_clock_period] to get the current RCX period in usec.
             * Please note that this value is multiplied by [1024 * 1024]
             */
            time = (lp_clocks * (uint64_t)rcx_clock_period);
            time = (time >> 20); // divide with (1024 * 1024)

        } else {

            OS_ASSERT(0); // Invalid lp clock source
        }

        return time;
}



/*
 * \callgraph
 *
 * \brief This function performs a SW RTC measurement using APIs from the RTC
 *        mechanism (provided by the SDK), to calculate the number of low power clock
 *        cycles that fit within a given time interval.
 */
void sw_rtc_date_time_read(sw_rtc_date_time_t *val)
{

        uint64_t sw_rtc_value_us = 0;
        uint64_t sw_rtc_dif = 0;


        /* Get the current RTC value expressed in lp clock ticks. */
        sw_rtc_new = rtc_get();


        /*
         * To reduce values used in various calculations, calculate
         * only the lp clocks since the last RTC measurement.
         * (instead of invoking all the lp clock cycles measured
         * from the very beginning of chip's operation)
         */
        sw_rtc_dif = sw_rtc_new - sw_rtc_old;
```

```
#if (DBG == 1)
        printf("RTC diff: %ld\r\n", (uint32_t)sw_rtc_dif);
#endif


        /*
         * The current RTC value becomes the old  for the next RTC measurement.
         */
        sw_rtc_old = sw_rtc_new;


        /* Convert the lp clock cycles into [microseconds]  */
        sw_rtc_value_us = sw_rtc_convert_lp_to_time(sw_rtc_dif);


        /* Update the calendar using the current time in [microseconds]  */
        sw_rtc_convert_time_to_calendar(sw_rtc_value_us);


        /* Safeguard the structure that holds the current date/time */
        OS_MUTEX_GET(sw_rtc_mutex, OS_MUTEX_FOREVER);

        /* Copy the current date/time */
        memcpy((uint8_t *)val, (uint8_t *)calendar, sizeof(sw_rtc_date_time_t));

        /* Release the previously acquired mutex so that other tasks can use it e.g. the
           daemon task */
        OS_MUTEX_PUT(sw_rtc_mutex);
}


/*
 * Function that performs an SW RTC measurement and returns the current date.
 */
void sw_rtc_date_read(sw_rtc_date_t *val)
{

        /* Allocate memory to hold both date and time */
        sw_rtc_date_time_t *date_time = (sw_rtc_date_time_t *)
                                        OS_MALLOC(sizeof(sw_rtc_date_time_t));
        OS_ASSERT(date_time);

        /* Read the current date/time */
        sw_rtc_date_time_read(date_time);

        /* Safeguard the structure that holds the current date/time */
        OS_MUTEX_GET(sw_rtc_mutex, OS_MUTEX_FOREVER);

        /* Copy the current date */
        memcpy((uint8_t *)val, (uint8_t *)&date_time->date, sizeof(sw_rtc_date_t));

        /* Release the previously acquired mutex so that other tasks can use it e.g. the
           daemon task */
        OS_MUTEX_PUT(sw_rtc_mutex);
```

```
        OS_FREE(date_time);
}



/*
 * Function that performs an SW RTC measurement and returns the current time.
 */
void sw_rtc_time_read(sw_rtc_time_t *val)
{

        /* Allocate memory to hold both date and time */
        sw_rtc_date_time_t *date_time = (sw_rtc_date_time_t *)
                                        OS_MALLOC(sizeof(sw_rtc_date_time_t));
        OS_ASSERT(date_time);

        /* Read the current date/time */
        sw_rtc_date_time_read(date_time);

        /* Safeguard the structure that holds the current date/time */
        OS_MUTEX_GET(sw_rtc_mutex, OS_MUTEX_FOREVER);

        /* Copy the current time  */
        memcpy((uint8_t *)val, (uint8_t *)&date_time->time, sizeof(sw_rtc_time_t));

        /* Release the previously acquired mutex so that other tasks can use it e.g. the
           daemon task */
        OS_MUTEX_PUT(sw_rtc_mutex);


        OS_FREE(date_time);
}


/* Sanity checks for the date structure */
int sw_rtc_date_write_sanity_check(sw_rtc_date_t *val)
{
        if ((val->month_date > 31) || (val->month_date == 0) || (val->month > 12) ||
                                                (val->month == 0) || (val->week_date > 6)) {
                return -1;
        } else {
                return 0;
        }
}


/* Sanity checks for the time structure */
int sw_rtc_time_write_sanity_check(sw_rtc_time_t *val)
{
        if ((val->hour > 23) || (val->min > 59) || (val->sec > 59) || (val->msec > 999) ||
(val->usec > 999))  {
                return -1;
        } else {
```

```
            return 0;
        }
}


/*
 * Function responsible for setting the current date and time
 */
void sw_rtc_date_time_write(sw_rtc_date_time_t *val)
{
        int error_date;
        int error_time;


        /**
         * Perform sanity checks on the provided values.
         *
         * \note If the returned value is -1 then the provided values are out of the
         *       boundaries
         *
         */
        error_date = sw_rtc_date_write_sanity_check(&val->date);
        error_time = sw_rtc_time_write_sanity_check(&val->time);

        OS_ASSERT((error_date == 0) && (error_time == 0));


        /* Safeguard the structure that holds the current date/time */
        OS_MUTEX_GET(sw_rtc_mutex, OS_MUTEX_FOREVER);

        /* Update date/time parameters */
        memcpy((uint8_t *)calendar, (uint8_t *)val, sizeof(sw_rtc_date_time_t));

        /* Release the previously acquired mutex so that other tasks can use it e.g. the
           daemon task */
        OS_MUTEX_PUT(sw_rtc_mutex);
}




/*
 * Function responsible for setting the current date
 */
void sw_rtc_date_write(sw_rtc_date_t *val)
{
        int error;


        /**
         * Perform sanity checks on the provided values.
         *
         * \note If the returned value is -1 then the provided values are out of the
         *       boundaries
         *
```

```
        */
        error = sw_rtc_date_write_sanity_check(val);
        OS_ASSERT(error == 0);

        /* Safeguard the structure that holds the current date */
        OS_MUTEX_GET(sw_rtc_mutex, OS_MUTEX_FOREVER);

        /* Update date parameters */
        memcpy((uint8_t *)&calendar->date, (uint8_t *)val, sizeof(sw_rtc_date_t));

        /* Release the previously acquired mutex so that other tasks can use it e.g. the
daemon task */
        OS_MUTEX_PUT(sw_rtc_mutex);
}


/*
 * Function responsible for setting the current time
 */
void sw_rtc_time_write(sw_rtc_time_t *val)
{
        int error;


        /**
         * Perform sanity checks on the provided values.
         *
         * \note If the returned value is -1 then the provided values are out of  boundaries
         *
         */
        error = sw_rtc_time_write_sanity_check(val);
        OS_ASSERT(error == 0);


        /* Safeguard the structure that holds the current time */
        OS_MUTEX_GET(sw_rtc_mutex, OS_MUTEX_FOREVER);

        /* Update time parameters */
        memcpy((uint8_t *)&calendar->time, (uint8_t *)val, sizeof(sw_rtc_time_t));

        /* Release the previously acquired mutex so that other tasks can use it e.g. the
daemon task */
        OS_MUTEX_PUT(sw_rtc_mutex);
}
```

6.8 Date/Time Implementation Header File

Create a new header file, for example sw_rtc_date_time.h containing the following code:

```
#include <stdint.h>
#include <stdbool.h>
```

```
/* Days of a week */
typedef enum {
        WEEK_DATE_SUNDAY        = 0,
        WEEK_DATE_MONDAY        = 1,
        WEEK_DATE_TUESDAY       = 2,
        WEEK_DATE_WEDNESDAY     = 3,
        WEEK_DATE_THURSDAY      = 4,
        WEEK_DATE_FRIDAY        = 5,
        WEEK_DATE_SATURDAY      = 6,
} WEEK_DATE;


/*
 * Date related parameters
 */
typedef struct sw_rtc_date {
        uint32_t year;          // Current year: there is no limitation
        uint32_t month;         // Range: (1..12)
        uint32_t month_date; // Range: (1..31) or (1..30) or (1..29) or (1..28)
        uint32_t week_date;  // Range: (0:6) where 0 = Sunday ... 6 = Saturday
} sw_rtc_date_t;


/*
 * Time related parameters
 */
typedef struct sw_rtc_time {
        uint32_t hour;  // Range: (0..23)
        uint32_t min;   // Range: (0..59)
        uint32_t sec;   // Range: (0..59)
        uint32_t msec;  // Range: (0..999)
        uint64_t usec;  // Range: (0..999)
} sw_rtc_time_t;


/*
 * Date/time (calendar) related parameters
 */
typedef struct sw_rtc_calendar {
        sw_rtc_date_t date;
        sw_rtc_time_t time;
} sw_rtc_date_time_t;



/**
 *
 * \brief SW RTC initialization
 *
 * This function initializes all the resources required for the SW RTC mechanism.
 *
 * \note It should be the first invoked API before any other SW RTC related operations.
 *
 */
```

```
void sw_rtc_init();


/**
 *
 * \brief SW RTC date/time read operation
 *
 * This function is responsible for reading the current date and time. It should be
 * considered as a shortcut function since it combines both sw_rtc_date_read()
 * and sw_rtc_time_read().
 *
 *
 * \param [out] val  A structure in which the current date/time will be stored
 *
 * \note This function can be called by any task at any time, to read the current date.
 *      A protection mechanism against multiple accesses has been implemented.
 *
 */
void sw_rtc_date_time_read(sw_rtc_date_time_t *val);




/*
 * \brief SW RTC date read operation
 *
 * This function is responsible for reading the current date.
 *
 *
 * \param [out] val   A structure in which the current date will be stored
 *
 * \note This function can be called by any task at any time, to read the current date.
 *          A protection mechanism against multiple accesses has been implemented.
 *
 */
void sw_rtc_date_read(sw_rtc_date_t *val);


/*
 * \brief SW RTC time read operation
 *
 * This function is responsible for reading the current time.
 *
 *
 * \param [out] val  A structure in which the current time will be stored
 *
 * \note This function can be called by any task at any time, to read the current date.
 *          A protection mechanism against multiple accesses has been implemented.
 *
 */
void sw_rtc_time_read(sw_rtc_time_t *val);



/*
```

```
 * \brief SW RTC date/time write operation
 *
 * This function is responsible for setting the current date and time. It should be
 * considered as a shortcut function since it combines both sw_rtc_date_write()
 * and sw_rtc_time_write()
 *
 *
 * \param[in] val The preferred values for date/time.
 *
 * \note The weekday is calculated automatically by the app.
 *
 * \note The code performs sanity checks to identify values out of boundaries.
 *       For any invalid value, the code gets stuck in assertions!
 *
 * \note This function can be called by any task at any time, to set date/time.
 *     A protection mechanism against multiple accesses has been implemented.
 *
 */
void sw_rtc_date_time_write(sw_rtc_date_time_t *val);



/*
 * \brief SW RTC date write operation
 *
 * This function is responsible for setting the current time.
 *
 *
 * \param[in] val The preferred values for time
 *
 * \note The code performs sanity checks to identify values out of boundaries.
 *       For any invalid value, the code gets stuck in assertions!
 *
 * \note This function can be called by any task at any time, to set date/time.
 *       A protection mechanism against multiple accesses has been implemented.
 *
 */
void sw_rtc_time_write(sw_rtc_time_t *val);



/*
 * \brief SW RTC time write operation
 *
 * This function responsible for setting the current time.
 *
 *
 * \param[in] val The preferred values for date.
 *
 * \note The weekday is calculated automatically by the code.
 *
 * \note The code performs sanity checks to identify values out of boundaries.
 *       For any invalid value, the code gets stuck in assertions!
 *
 * \note This function can be called by any task at any time, to set date/time.
```

```
 *        A protection mechanism against multiple accesses has been implemented.
 *
 */
void sw_rtc_date_write(sw_rtc_date_t *val);
```

6.9 Alert Implementation Source File

Code snippet of the alert mechanism implementation. Create a new source file, for example alert_mechanism.c containing the following code:

```
/*
 * Masks for positioning alert sources (alert class)
 */
#define ALERT_SOURCE_HOURS_Pos   (0x00)
#define ALERT_SOURCE_MINUTES_Pos (0x01)
#define ALERT_SOURCE_SECONDS_Pos (0x02)


/* Handles of OS timers - One for each ALERT class */
__RETAINED_RW static OS_TIMER timer_h = NULL;
__RETAINED_RW static OS_TIMER timer_m = NULL;
__RETAINED_RW static OS_TIMER timer_s = NULL;



/*
 * Hold the current user-defined callback function
 */
__RETAINED_RW static alert_cb cb = NULL;

/*
 * Hold the current alert configurations.
 */
__RETAINED_RW static time_alert_t *alert_cfg = NULL;



/*
 * OS timer callback, dedicated for Hours Alert Class.
 * This callback is called upon timer's expiration.
 */
static void timer_h_cb(OS_TIMER timer)
{
    // Check whether the [repeat] flag is set, then restart timer.
    if (alert_cfg->repeat_h == pdTRUE) {
        OS_TIMER_START(timer, OS_TIMER_FOREVER);
    }

    // Check whether a user-defined function has been declared.
    if (cb) {
        cb((uint8_t)(1 << ALERT_SOURCE_HOURS_Pos));
    }
```

```
}


/*
 * OS timer callback, dedicated for Minutes Alert Class.
 * This callback is called upon timer's expiration.
 */
void timer_m_cb(OS_TIMER timer)
{
    // Check whether the [repeat] flag is set, then restart timer.
    if (alert_cfg->repeat_m == pdTRUE) {
        OS_TIMER_START(timer, OS_TIMER_FOREVER);
    }

    // Check whether a user-defined function has been declared.
    if (cb) {
        cb((uint8_t)(1 << ALERT_SOURCE_MINUTES_Pos));
    }

}



/*
 * OS timer callback, dedicated for Seconds Alert Class.
 * This callback is called upon timer's expiration.
 */
void timer_s_cb(OS_TIMER timer)
{
    // Check whether the [repeat] flag is set, then restart timer.
    if (alert_cfg->repeat_s == pdTRUE) {
        OS_TIMER_START(timer, OS_TIMER_FOREVER);
    }

    // Check whether a user-defined function has been declared.
    if (cb) {
        cb((uint8_t)(1 << ALERT_SOURCE_SECONDS_Pos));
    }

}




/*
 * Initialize the alert mechanism. Three OS timers are being created,
 * one for each ALERT class, without enabling them.
 */
void time_alert_init(void)
{
    /* Allocate memory for the alert parameters */
    alert_cfg = (time_alert_t *) OS_MALLOC(sizeof(time_alert_t));
    OS_ASSERT(alert_cfg);
```

```
    /* Clear the alert structure */
    memset((uint8_t *)alert_cfg, 0x00, sizeof(time_alert_t));


    /*
     * Create one-shot OS timer dedicated for Hours Alert Class.
     *
     * \note  Timer period - expressed in OS ticks -
     *       must be greater than 1, otherwise an assertion is issued!
     */
    timer_h = OS_TIMER_CREATE("HOURS", OS_MS_2_TICKS(10),
                              OS_TIMER_FAIL, (void *)0, timer_h_cb);
    OS_ASSERT(timer_h);

    /*
     * Create one-shot OS timer dedicated for Minutes Alert Class.
     *
     * \note:  Timer's period - expressed in OS ticks -
     *        must be greater than 1, otherwise an assertion is issued!
     */
    timer_m = OS_TIMER_CREATE("MINUTES", OS_MS_2_TICKS(10),
                              OS_TIMER_FAIL, (void *)0, timer_m_cb);
    OS_ASSERT(timer_m);

    /*
     * Create one-shot OS timer dedicated for Seconds Alert Class.
     *
     * \note: Timer's period - expressed in OS ticks -
     *       must be greater than 1, otherwise an assertion is issued.
     */
    timer_s = OS_TIMER_CREATE("SECONDS", OS_MS_2_TICKS(10),
                               OS_TIMER_FAIL, (void *)0, timer_s_cb);
    OS_ASSERT(timer_s);

}


/*
 * Function responsible for setting an alert
 */
void time_alert_class_set(uint16_t value, bool repeat, SET_ALERT_CLASS alert)
{
    OS_BASE_TYPE ret = 0;
    uint64_t timer_ticks = 0;

    switch(alert) {

    /* Set alerts for Hours Alert Class */
    case SET_ALERT_CLASS_Hour:
        /* Then check whether the new value assigned is non-zero */
        if (value > 0) {
            /*
             * Recalculate timer period, that is,
             * (hours * 60 minutes/hour * 60 seconds/minute * 1000
             * milliseconds/second)
```

```
                *
                * Note: Changing the period of a dormant timer will also
                * start the timer!
                */

                timer_ticks = (((uint64_t)value * 3600000 * configTICK_RATE_HZ) / 1000);

                /* Perform a sanity check for 32-bit max number */
                if (timer_ticks >= (uint64_t)4294967295UL) {
                    OS_ASSERT(0);
                }

                ret = OS_TIMER_CHANGE_PERIOD(timer_h,
                            (uint32_t)timer_ticks, OS_TIMER_FOREVER);

                OS_ASSERT(ret == OS_OK);

            /* Otherwise the new value is zero and hence timer should be halted */
            } else {
                ret = OS_TIMER_STOP(timer_h, OS_TIMER_FOREVER);
                OS_ASSERT(ret == OS_OK);
            }

            /* Store the new parameters for Hours Alert Class */
            alert_cfg->hour     = value;
            alert_cfg->repeat_h = repeat;
            break;

        /* Set alerts for Minutes Alert Class */
        case SET_ALERT_CLASS_Min:
            /* Then check whether the new value assigned is non-zero */
            if (value > 0) {
                /*
                * Recalculate timer period, that is,
                * (minutes * 60 seconds/minute * 1000 milliseconds/second)
                *
                * Note: changing the period of a dormant timer will also
                * start the timer!
                */
                timer_ticks = (((uint64_t)value * 60000 * configTICK_RATE_HZ) / 1000);

                /* Perform a sanity check for 32-bit max number */
                if (timer_ticks >= (uint64_t)4294967295UL) {
                    OS_ASSERT(0);
                }

                ret = OS_TIMER_CHANGE_PERIOD(timer_m,
                            (uint32_t)timer_ticks, OS_TIMER_FOREVER);

                OS_ASSERT(ret == OS_OK);

            /* Otherwise the new value is zero and hence timer should be halted */
            } else {
                ret = OS_TIMER_STOP(timer_m, OS_TIMER_FOREVER);
                OS_ASSERT(ret == OS_OK);
```

```
                }

            /* Store the new parameters for Minutes Alert Class */
            alert_cfg->min = value;
            alert_cfg->repeat_m = repeat;
            break;

        /* Set alerts for Seconds Alert Class */
        case SET_ALERT_CLASS_Sec:
            /* Then check whether the new value assigned is non-zero */
            if (value > 0) {
                /*
                 * Recalculate timer period, that is,
                 * (seconds * 1000 milliseconds/second)
                 *
                 * Note: changing the period of a dormant timer will also
                 * start the timer!
                 */

                timer_ticks = (((uint64_t)value * 1000 * configTICK_RATE_HZ) / 1000);

                /* Perform a sanity check for 32-bit max number */
                if (timer_ticks >= (uint64_t)4294967295UL) {
                    OS_ASSERT(0);
                }

                ret = OS_TIMER_CHANGE_PERIOD(timer_s,
                        (uint32_t)timer_ticks, OS_TIMER_FOREVER);

                OS_ASSERT(ret == OS_OK);

               /* Otherwise the new value is zero and hence timer should be halted */
             } else {
                 ret = OS_TIMER_STOP(timer_s, OS_TIMER_FOREVER);
                 OS_ASSERT(ret == OS_OK);
               }

            /* Store the new parameters for Seconds Alert Class */
            alert_cfg->sec = value;
            alert_cfg->repeat_s = repeat;
            break;

        default:
            OS_ASSERT(0);
        }

}


/*
 * Function responsible for settings alerts of all the available alert classes
 */
void time_alert_class_set_all(time_alert_t *alert)
{
        time_alert_class_set(alert->hour, alert->repeat_h, SET_ALERT_CLASS_Hour);
```

```
        time_alert_class_set(alert->min,  alert->repeat_m, SET_ALERT_CLASS_Min);
        time_alert_class_set(alert->sec,  alert->repeat_s, SET_ALERT_CLASS_Sec);
}


/*
 * Function responsible for clearing alerts
 */
void time_alert_class_clear(CLEAR_ALERT_CLASS alert)
{

        switch(alert) {

        /* Disable alerts for Hours Alert Class */
        case CLEAR_ALERT_CLASS_Hour:
                /* Clear the associated variables */
                alert_cfg->hour = 0;
                alert_cfg->repeat_h = false;

                /*Stop the associated OS timer */
                OS_TIMER_STOP(timer_h, OS_TIMER_FOREVER);
                break;

        /* Disable alerts for Minutes Alert Class */
        case CLEAR_ALERT_CLASS_Min:
                /* Clear the associated variables */
                alert_cfg->min = 0;
                alert_cfg->repeat_m = false;

                /*Stop the associated OS timer */
                OS_TIMER_STOP(timer_m, OS_TIMER_FOREVER);
                break;

        /* Disable alerts for Seconds Alert Class */
        case CLEAR_ALERT_CLASS_Sec:
                /* Clear the associated variables */
                alert_cfg->sec = 0;
                alert_cfg->repeat_s = false;

                /*Stop the associated OS timer */
                OS_TIMER_STOP(timer_s, OS_TIMER_FOREVER);
                break;

        /* Disable alerts for all classes */
        case CLEAR_ALERT_CLASS_All:
                /* Clear all the alert variables */
                memset((uint8_t *)alert_cfg, 0x00, sizeof(time_alert_t));

                /*Stop all the associated OS timers */
                OS_TIMER_STOP(timer_h, OS_TIMER_FOREVER);
                OS_TIMER_STOP(timer_m, OS_TIMER_FOREVER);
                OS_TIMER_STOP(timer_s, OS_TIMER_FOREVER);
                break;

        default:
```

```
            OS_ASSERT(0);
        }

}



/*
 * Function responsible for registering the user-defined callback function
 */
void time_alert_register_cb(alert_cb user_cb)
{
        cb = user_cb;
}

/*
 * Function responsible for deleting the user-defined callback function
 */
void time_alert_unregister_cb(void)
{
        cb = NULL;
}
```

6.10 Alert Implementation Header File

Create a new header file, for example alert_mechanism.h and add the following code:

```
#include <stdbool.h>
#include <stdlib.h>
#include <stdio.h>


/**
 *
 * \brief Alert related parameters.
 *
 * \note There are three distinct ALERT classes that can be configured individually.
 *
 */
typedef struct time_alert {
        /*
         * A value equal to zero means that the corresponding alert class is disabled.
         */
        uint16_t hour;
        uint16_t min;
        uint16_t sec;

        /*
         * Define whether an alert will be repeatable. Valid values are:
         * true: for repeatable alarms, false otherwise.
         */
        bool repeat_h;
```

```
        bool repeat_m;
        bool repeat_s;
} time_alert_t;


/**
 *
 * \brief Clear an alert class
 *
 */
typedef enum {
    CLEAR_ALERT_CLASS_Hour = 0,
    CLEAR_ALERT_CLASS_Min  = 1,
    CLEAR_ALERT_CLASS_Sec  = 2,
    CLEAR_ALERT_CLASS_All  = 3,
} CLEAR_ALERT_CLASS;


/**
 *
 * \brief Set an alert class
 *
 */
typedef enum {
        SET_ALERT_CLASS_Hour = 0,
        SET_ALERT_CLASS_Min  = 1,
        SET_ALERT_CLASS_Sec  = 2,
} SET_ALERT_CLASS;



/**
 *
 * \brief Alert Class Masks
 *
 * These masks should be used within callback functions to identify
 * the source of an alert event. This is necessary since all the
 * ALERT classes will call the same callback function.
 *
 */
#define ALERT_SOURCE_HOURS_Msk   (0x01)
#define ALERT_SOURCE_MINUTES_Msk (0x02)
#define ALERT_SOURCE_SECONDS_Msk (0x04)



/**
 *
 * \brief User-defined callback function
 *
 * This function will be called upon an alert event. It should have that specific
prototype!
 *
 * \note All the alert classes will call the same callback. User is responsible to identify
 *       the source of an alert event. Following the recommended callback usage:
```

```
 *
 * \code{.c}
 * void my_alert_cb(uint8_t source)
 * {
 *     if (source & ALERT_SOURCE_HOURS_Msk) {
 *          // Do something...
 *     }
 *
 *     if (source & ALERT_SOURCE_MINUTES_Msk) {
 *          // Do something...
 *     }
 *
 *     if (source & ALERT_SOURCE_SECONDS_Msk) {
 *          // Do something...
 *     }
 * }
 * \endcode
 */
typedef void (*alert_cb)(uint8_t source);



/**
 *
 * \brief Alert mechanism initialization
 *
 * This function initializes all the resources required for the alert mechanism.
 *
 * \note It should be the first invoked API before any other alert related operation.
 *
 */
void time_alert_init(void);



/**
 *
 * \brief Callback function registration
 *
 *
 * This function should be used for registering a callback function, called upon an alert
 * event.
 *
 *
 * \param [in] user_cb  The user-defined callback function.
 *
 * \note If \p user_cb is set to NULL, none callback function will be called upon an alert
 *       event.
 *
 */
void time_alert_register_cb(alert_cb user_cb);


/**
 *
```

```
 * \brief Callback function deletion
 *
 * This function should be used for unregistering any previously declared callback.
 *
 */
void time_alert_unregister_cb(void);



/**
 *
 * \brief Set alerts for a specific ALERT class
 *
 * This function should be used for declaring alerts of a specific ALERT class.
 * There are three distinct alert classes that can be configured individually.
 *
 * The following code demonstrates alert configurations, expressed in minutes.
 * Assuming we want alerts to be triggered every 5 minutes (triggered repeatedly).
 *
 * \code{.c}
 * time_alert_set_class(5, true, SET_ALERT_CLASS_Min)
 * \endcode
 *
 * The following code demonstrates alert configurations, expressed in seconds.
 * Assuming we  want an alert after 5 minutes (triggered once)
 *
 * \code{.c}
 * time_alert_set_class(5, false, SET_ALERT_CLASS_Min)
 * \endcode
 *
 *
 * \param [in] value   Time interval of alert events.
 * \param [in] repeat  If set to false, alerts will be triggered once, repeatedly otherwise.
 * \param [in] alert   One of the three alert classes.
 *
 *
 * \note If \p value is set to zero, alerts for that specific alert class will be disabled.
 *
 */
void time_alert_class_set(uint16_t value, bool repeat, SET_ALERT_CLASS alert);


/**
 * \brief Set alert events for all the three alert classes
 *
 * This is a shortcut function since it can configure all the three alert classes.
 *
 * \param [in] alert  Parameters for all the tree alert classes.
 *
 */
void time_alert_class_set_all(time_alert_t *alert);



/**
```

```
 *
 * \brief Clear alert events
 *
 * This function should be used for disabling alert events of an individual or all the three
 *  alert classes
 *
 * \param [in] alert  The alert class
 *
 */
void time_alert_class_clear(CLEAR_ALERT_CLASS alert);
```

Revision History

Revision	Date	Description
1.0	18-May-2018	First released version
2.0	19-Sep-2018	Updated source code and APIs, Improved figures in *Demonstration Example* section.

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

Contacting Dialog Semiconductor

United Kingdom (Headquarters)
Dialog Semiconductor (UK) LTD
Phone: +44 1793 757700

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

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 5769 5100

Taiwan
Dialog Semiconductor Taiwan
Phone: +886 281 786 222

Web site:
www.dialog-semiconductor.com

Hong Kong
Dialog Semiconductor Hong Kong
Phone: +852 2607 4271

Korea
Dialog Semiconductor Korea
Phone: +82 2 3469 8200

China (Shenzhen)
Dialog Semiconductor China
Phone: +86 755 2981 3669

China (Shanghai)
Dialog Semiconductor China
Phone: +86 21 5424 9058