Tutorial

GPADC Adapters

For the DA1468x SoC

Abstract

This tutorial should be used as a reference guide to gain a deeper understanding of the 'GPADC Adapters' concept. As such, it covers a broad range of topics including an introduction to Adapter mechanism as well as a detailed description of the various GPADC conversion schemes. Furthermore, it covers a number of sections containing in depth software analysis of a complete demonstration example.

Contents

Figures

Tables

Terms and Definitions

DevKit	Development Kit
FSM	Finite-State-Machine
GPADC	General Purpose Analog-to-Digital Converter
ISR	Interrupt Service Routine
I2C	Inter-Integrated Circuit
LLD	Low Level Drivers
ms	millisecond
OS	Operating System
SDK	Software Development Kit
SPI	Serial Peripheral Interface
SW	Software

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 adapters concept
- Explain the different APIs and configurations of GPADC adapters
- Give a complete sample project demonstrating the usage of GPADC and battery adapters

1.2 GPADC Adapters Introduction

This tutorial explains GPADC adapters and how to configure the DA1468x family of devices to perform analog-to-digital conversions, including battery measurements. The latter can be considered as a special use case of the GPADC interface.

The GPADC adapter is an intermediate layer between the GPADC Low Level Drivers (LLDs) and a user application. It allows the user to utilize the GPADC interface in a simpler way than when using pure LLDs functions. The key features of GPADC adapters are:

- Synchronous writing/reading operations block the calling freeRTOS task while the operation is performed using semaphores rather than relying on a polling loop approach. This means that while the hardware is busy transferring data, the operating system (OS) scheduler may select another task for execution, utilizing processor time more efficiently. When the transfer has finished, the calling task is released and resumes its execution.
- It ensures that only one application task can use the GPADC interface after acquiring it.
- Placing code between ad_gpadc_acquire() and ad_gpadc_release() ensures that only one task can use the GPADC interface to perform analog-to-digital conversions. During this period no other task can use the GPADC interface until the ad_gpadc_release() function is called by the owning task.
- Power Manager (PM) of the chip is aware of the GPADC peripheral usage and, before the system enters sleep, it checks whether or not there is activity on the GPADC block.

Note: Adapters are not implemented as separate tasks and should be considered as an additional layer between the application and the LLDs. It is recommended to use adapters for accessing a hardware block.

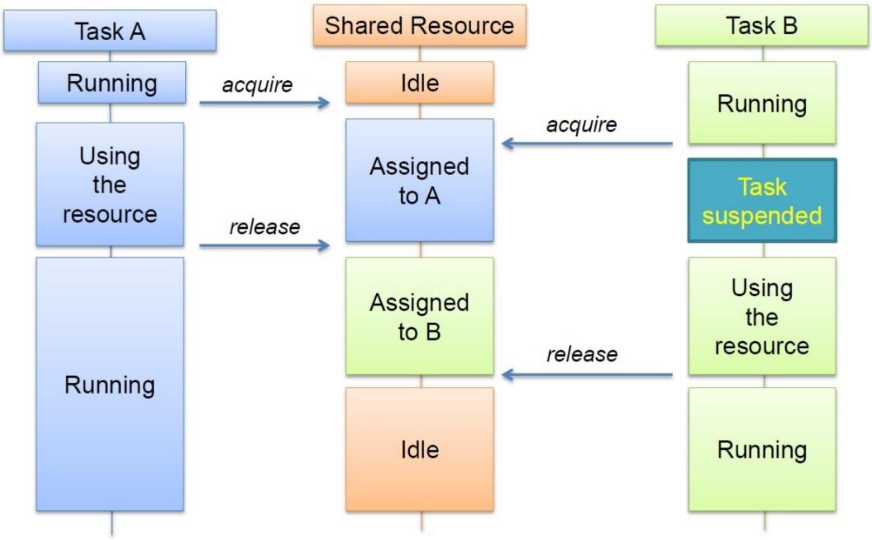

Figure 1: Adapters Communication

2 GPADC Adapters Concept

This section explains the key features of GPADC adapters as well as the steps to enable and correctly configure the peripheral adapters for the analog-to-digital functionality. The procedure is a four-step process which can be applied to almost every type of adapter including serial peripheral adapters (I^2C, SPI, UART).

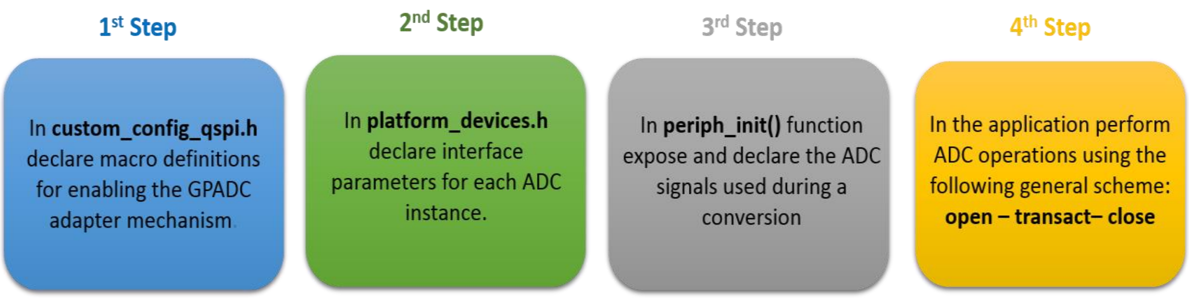

Figure 2: The Four-Step Process for Setting an Adapter Mechanism

2.1 Header Files

The header files related to adapter functionality can be found in /sdk/adapters/include. These files contain the APIs and macros for configuring the majority of the available hardware blocks. In particular, this tutorial focuses on the adapters that are responsible for the GPADC hardware block. Table 1 briefly explains the header files related to GPADC adapters (red indicates the path under which the files are stored, and green indicates which ones are used for GPADC operations.).

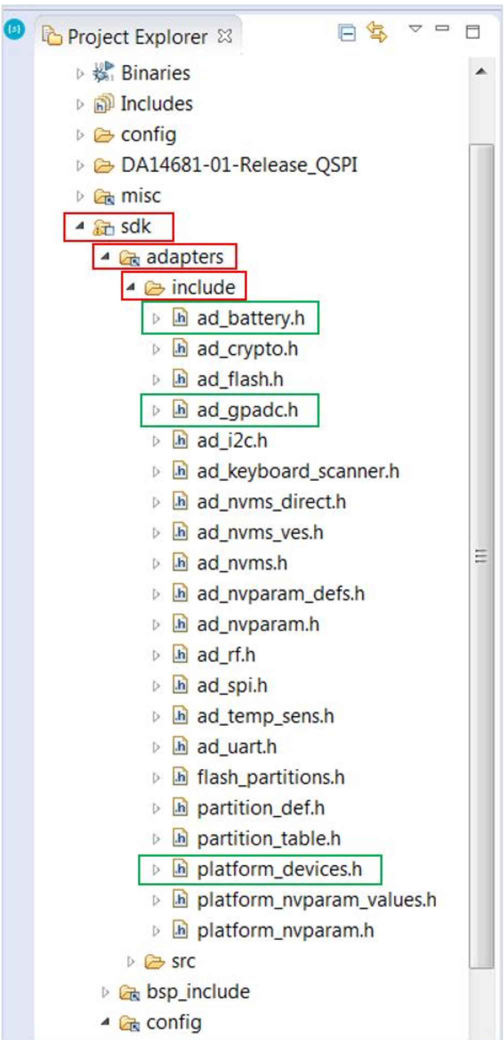

Figure 3: Headers for GPADC Adapters.

Table 1: Header Files used by GPADC Adapters

Filename	Description
ad_gpadc.h	This file contains APIs and macros for performing ADC operations. Use these APIs when accessing the GPADC interface to perform general-purpose conversions.
ad_battery.h	This file contains APIs and macros for performing battery measurements. Use these APIs when accessing the GPADC interface to perform battery voltage measurements.
platform_devices.h	This file contains macros for declaring virtual devices. These devices may be connected to the Dialog family of devices via a peripheral bus (for example, SPI, I^2C, UART) or a peripheral hardware block (for example, GPADC).

2.2 Preparing a GPADC Operation

1. As illustrated in Figure 4, the first step for configuring the GPADC adapter mechanism is to enable it by defining the following macros in /config/custom_config_qspi.h. Battery adapters are part of the GPADC adapter implementation and should be considered as a special analog-to-digital conversion.

```
/*
 * Macros for enabling GPADC + Battery measurement operations using Adapters
 */
#define dg_configUSE_HW_GPADC                   (1)
#define dg_configGPADC_ADAPTER                  (1)
#define dg_configBATTERY_ADAPTER                (1)

/*
 * This macro should be already written in the file so just make sure
 * it is set to 1
 */
#define dg_configUSE_HW_TEMPSENS                (1)
```

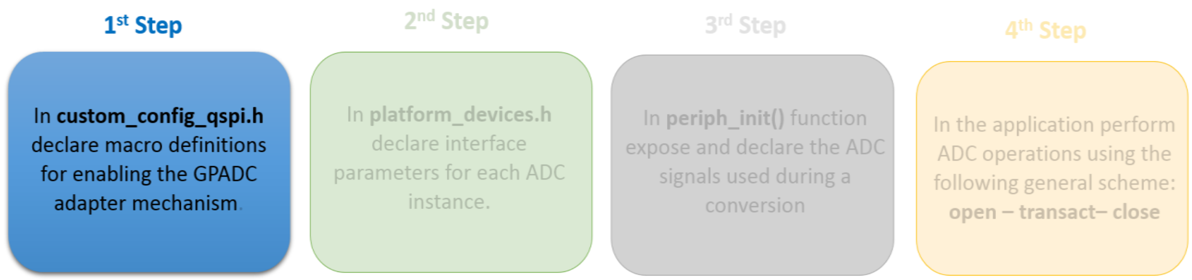

Figure 4: First Step for Configuring the GPADC Adapter Mechanism

From this point onwards, the overall adapter implementation with all its integrated functions is available.

2. The second step is to declare interface parameters for all the GPADC instances. As instance can be considered a set of settings describing the complete GPADC interface. These settings are applied every time the instance is selected and used. To do this, the SDK uses a macro, named GPADC_SOURCE:

```
/*
 * Macro for creating an instance of the GPADC interface
 */
GPADC_SOURCE(name, _clock_source, _input_mode, input, sample_time,
                                     cancel_offset,   _oversampling, _input_voltage)
```

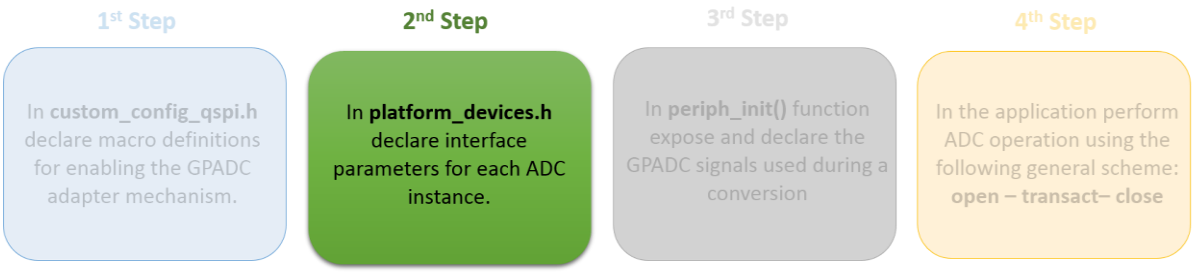

Figure 5: Second Step for Configuring the GPADC Adapter Mechanism

Table 2: Description of the Macro Fields Used for Declaring a GPADC Instance

Argument Name	Description
name	Declare an arbitrary alias for the GPADC interface (for instance, ADC_Channel_1). This name should be used for opening that specific instance.
_clock_source	Clock source of the GPADC controller. This can be either the internal high-speed or the system clock source. The first option selects the internal 200 MHz clock source while the second option the system clock source, that is 16 MHz or 96 MHz. Valid values are those from HW_GPADC_CLOCK enum in /sdk/peripherals/include/hw_gpadc.h.
_input_mode	This can be either single-ended or differential mode. Valid values are those from HW_GPADC_INPUT_MODE enum in /sdk/peripherals/include/hw_gpadc.h.
input	The analog pin of the DA1468x chip to use for the current measurement. In contrast with the digital GPIOs, analog pins are mapped on specific pins on DA1468x SoC. Valid values are those from HW_GPADC_INPUT enum in /sdk/peripherals/include/hw_gpadc.h.
sample_time	Sampling time. Valid values are from 0 to 15. A value of 0 corresponds to one GPADC clock cycle while the maximum permitted value corresponds to 15*32 GPADC clock cycles. This parameter is application-specific and depends on the selected clock source as well as the preferred accuracy of analog-to-digital conversions.
cancel_offset	The DA1468x SoC exhibits a feature for cancelling the offset on the provided analog signals. This is done via a chopping operation where two samples with opposite signal polarity are sampled at every conversion. This feature is recommended for DC and slowly changing signals. Valid values are TRUE or FALSE.

_oversampling	In this mode, multiple successive conversions are executed and the results are added together to increase the effective number of bits. By default, the GPADC controller is a 10-bit controller and, depending on the selected oversampling, the default 10 bits can be expanded up to 16 bits. Valid values are those from HW_GPADC_OVERSAMPLING enum in /sdk/adapters/include/ad_gpadc.h.
_input_voltage	The maximum permitted analog value present on an analog pin. The GPADC controller has its own 1.2 V voltage regulator (LDO) which represents the full scale reference voltage. The GPADC controller features an attenuator capable of attenuating a signal three times, thus an analog source up to 3.6 V can also be measured. Valid values are those from HW_GPADC_INPUT_VOLTAGE enum in /sdk/adapters/include/ad_gpadc.h.

3. As illustrated in Figure 6, the third step is the declaration of the GPADC signals.

```
static void prvSetupHardware( void )
{

   /* Init hardware */
   pm_system_init(periph_init)

}
```

Note: When the system enters sleep it loses its pin configurations. Thus, it is essential for the pins to be reconfigured to their last state as soon as the system wakes up. To do this, all pin configurations must be declared in periph_init() which is supervised by the Power Manager of the system.

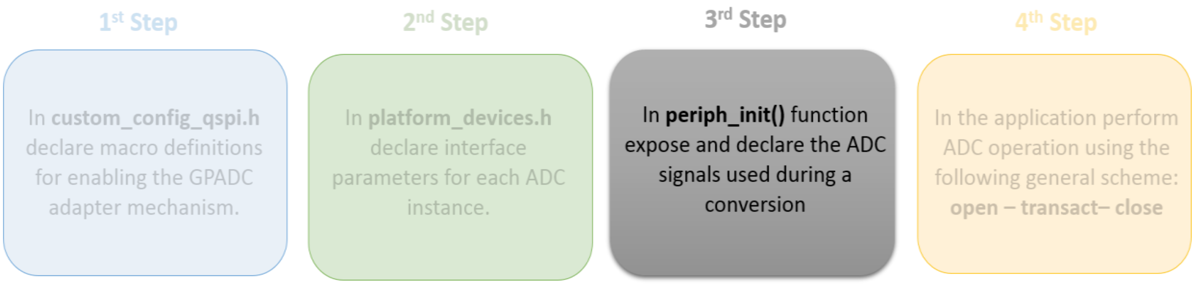

Figure 6: Third Step for Configuring the GPADC Adapter Mechanism

Note: In contrast with the digital GPIOs, analog pins are mapped on specific pins on the DA1468x SoC.

4. Having enabled the GPADC adapter mechanism, the developer is able to use all the available APIs for performing analog-to-digital conversions. The following describes the required sequence of APIs in an application to successfully execute GPADC operations.

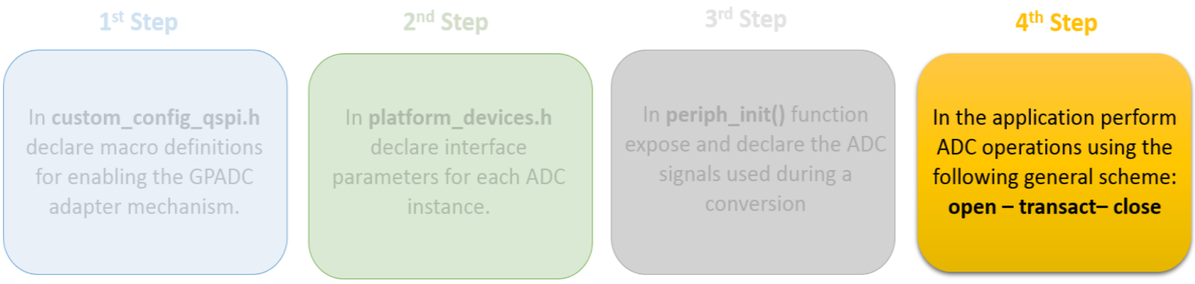

Figure 7: Fourth Step for Configuring the GPADC Adapter Mechanism

a. ad_gpadc_init() or GPADC_INIT()

Must be called once at either platform start (for instance, in system_init()) or task initialization to perform all the necessary initialization routines.

b. ad_gpadc_open()

Before using a GPADC instance, the application task must first open it. The function returns a handler to the main flow for use in subsequent adapter functions. Subsequent calls from other tasks simply return the existing handler.

c. ad_gpadc_acquire()

This API is optional since it is automatically called upon a GPADC operation and it is used for locking the GPADC block for the given opened instance. This function should be called when the application task wants to communicate to the GPADC block directly using low level drivers.

Note: This function can be called several times. It is essential that the number of ad_gpadc_acquire() calls matches the number of ad_gpadc_release() calls.

d. Perform analog-to-digital operations either synchronously or asynchronously.

After opening an instance, the application task(s) can perform any analog-to-digital conversion either synchronously or asynchronously. Please note that all the available APIs for accessing the GPADC interface, nest the corresponding APIs for acquiring and releasing the GPADC block.

e. ad_gpadc_release()

This function must be called for each call to ad_gpadc_acquire().

f. ad_gpadc_close()

After all user operations are done and the interface is no longer needed, it should be closed by the task that has currently acquired it. The application can then switch to other GPADC instances (as needed). Remember that GPADC adapter implementation follows a single instance scheme, that is only one instance can be opened at a time.

2.3 Battery Adapters

Battery voltage measurements can be considered as a special use case of the GPADC interface. The DA1468x chip has a dedicated analog pin internally connected to the terminals of the DA1468x daughterboard's battery holder. The SDK comes with additional APIs responsible for executing battery voltage measurements and it is recommended that the user should use these APIs when accessing the GPADC controller to perform such an operation.

a. ad_gpadc_init() or GPADC_INIT()

   Must be called once at either platform start (for instance, system_init()) or task initialization to perform all the necessary initialization routines.

b. ad_battery_open()

   Before using the battery instance the application task must first open it. In contrast with the ad_gpadc_open() API, this function has no input parameters since it is considered that only one interface is defined for battery voltage measurements.

Note: The function calls the ad_gpadc_open() API using the BATTERY_LEVEL id which is the default id for the battery interface configurations (declared in platform_devices.h).

c. Perform a battery voltage measurement either synchronously or asynchronously.

   – To perform synchronous measurements, use the ad_battery_read() function. This function waits until the GPADC controller becomes available.

   – To perform asynchronous measurements, use the ad_battery_read_async() function. When an analog-to-digital conversion is finished, a callback function is called and the results of the conversion are available within this callback.

d. ad_battery_raw_to_mvolt()

   After performing a successful analog-to-digital conversion, the developer can use this function to convert the raw ADC value to millivolts (mV).

e. ad_battery_close()

   After all user operations are done and the interface is no longer needed, it should be closed by the task that has currently acquired it.

2.4 Handling ADC Measurements

This section describes the basic operation principles of the ADC controller and how to handle the resulting data upon a successful analog-to-digital conversion.

Consider an ADC controller with a voltage reference of 5 V and a resolution of 10 bits, that is, it has the ability to detect 1024 discrete analog levels. In this case, any analog value in between 0 V and 5 V is converted into its equivalent ADC value as shown in Figure 8. The 0 V to 5 V range is divided into 2^{10} = 1024 steps. Thus, a 0 V input results in an ADC output of 0, a 5 V input gives an ADC value equal to 1023, and a 2.5 V input results in an ADC output of around 512.

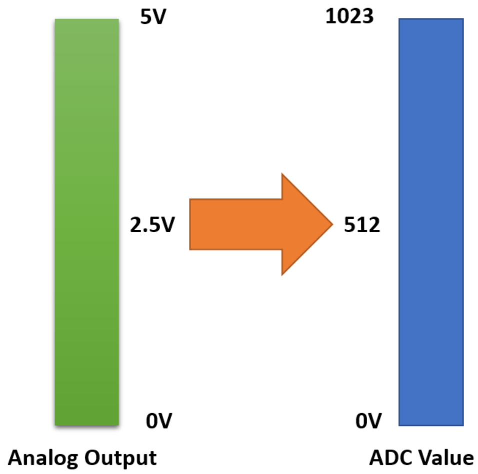

Figure 8: Analog-to-Digital Conversion Process

Figure 9 depicts the relationship between analog and raw ADC values.

$$\frac{\text{GPADC resolution}}{\text{Voltage reference}} = \frac{\text{Raw ADC result}}{\text{Analog measured voltage}}$$

Figure 9: Basic Relationship between Analog and Raw ADC Values

The resolution of the ADC controller is dependent on the oversampling used. GPADC adapters come with the following API which returns the ADC resolution depending on the selected oversampling: ad_gpadc_get_source_max(). Possible returned values are: 0x3FF (10 bits), 0x7FF (11 bits), 0xFFF (12 bits), 0x1FFF (13 bits), 0x3FFF (14 bits) 0x7FFF (15 bits), or 0xFFFF (16 bits)

Following is an example which demonstrates the computation of the raw ADC value for an analog input source of 3.05V. As described, the ADC voltage reference can be the default 1.2 V or 3.6 V when 3X attenuator is selected, or 5 V in the case of the VBAT channel. Assuming the VBAT channel is selected and the oversampling results in 12-bit resolution, the raw ADC value can be easily computed as illustrated in Figure 10

$$\frac{\text{GPADC resolution} * \text{Analog measured voltage}}{\text{Voltage reference}} = \text{Raw ADC result}$$

$$\frac{4096 * 3.05}{5} = 2498$$

Figure 10: Resulting Raw ADC Value

3 Analyzing The Demonstration Example

This section analyzes an application example which demonstrates using both the GPADC and battery adapters. The example is based on the **freertos_retarget** sample code found in the SDK. It adds two additional freeRTOS tasks which are responsible for various GPADC operations. One task performs analog-to-digital measurements on a dedicated pin on the Pro DevKit (P1.2), while the second task performs analog-to-digital measurements on the internal pin which is connected to the attached battery. The code also enables the wake-up timer for handling external events. Both synchronous and asynchronous GPADC operations are demonstrated.

3.1 Application Structure

1. The key goal of this demonstration is for the device to perform a few GPADC operations following an event. For demonstration purposes the **K1** button on the Pro DevKit has been configured as a wake-up pin. For more detailed information on how to configure and set a pin for handling external events, read the External Interruption tutorial.

   At each external event (produced at every **K1** button press), a dedicated callback function named wkup_cb() is triggered. In this function, the task responsible for the battery measurements is signaled so that it can unblock. This freeRTOS task performs a synchronous battery measurement and, upon finishing the analog-to-digital conversion, the raw ADC result is converted into millivolt (mV) so that it is easier for the developer to interpret the results. A debugging message with the converted data is also printed out on the serial console.

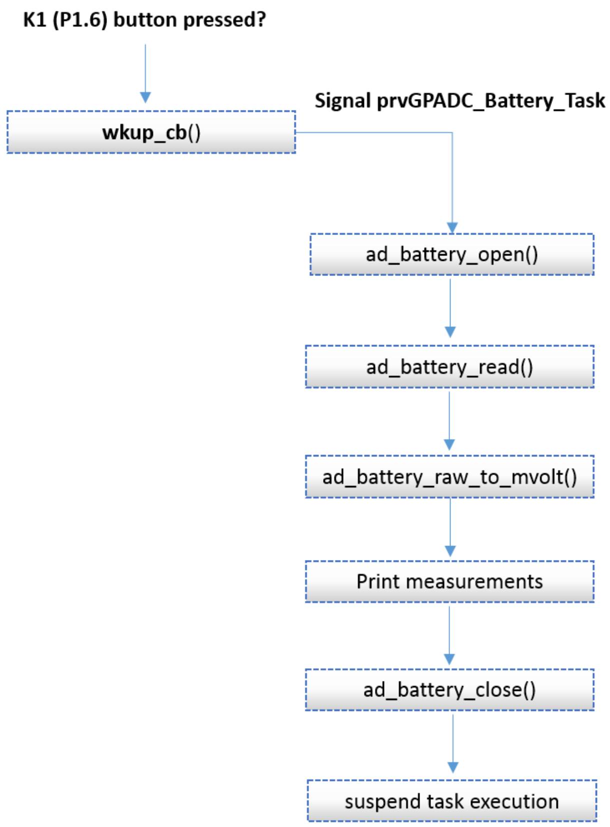

Figure 11: Battery Measurements SW FSM – Main Execution Path

2. At the same time, at every 1 second time interval, the task responsible for performing GPADC operations is executed. Depending on the value of the **POT_ASYNC_EN** macro, a synchronous or an asynchronous GPADC conversion is performed. At the end of every GPADC operation, the raw ADC value is converted into mV and a debugging message with the converted data is sent to the serial console.

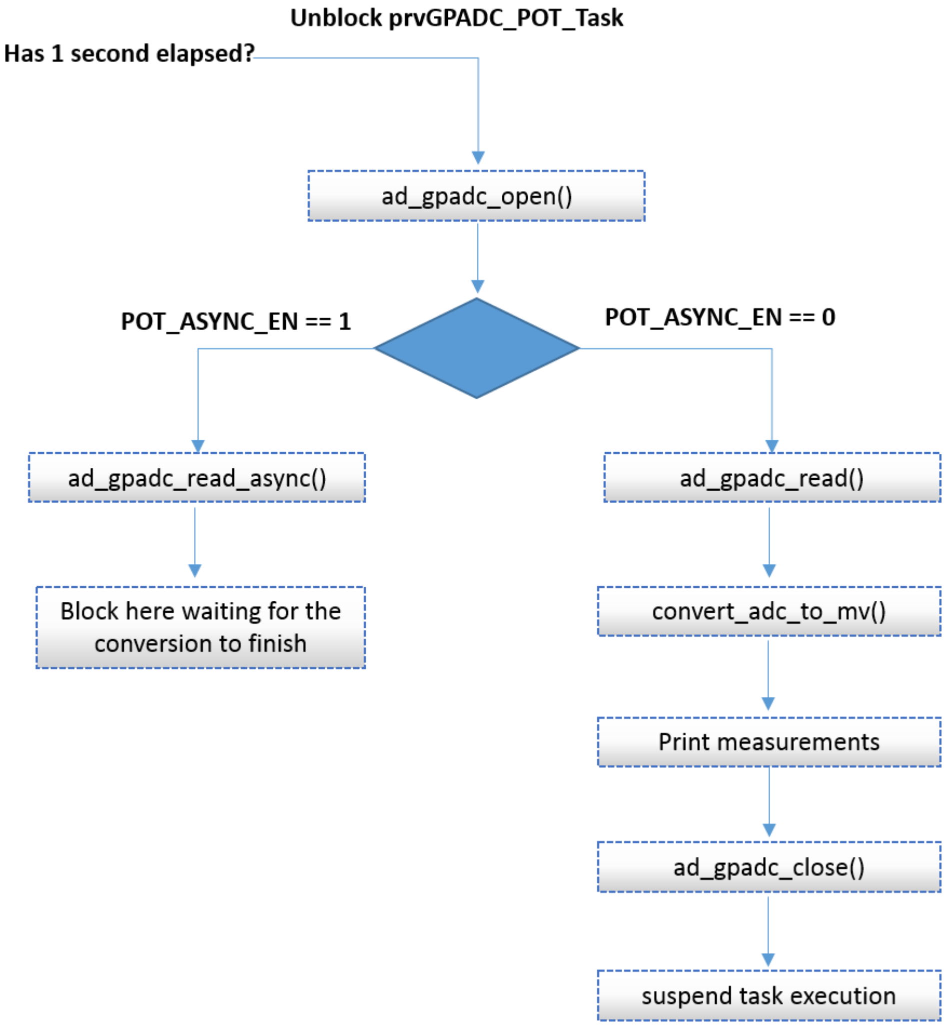

Figure 12: GPADC Conversions SW FSM – Main Execution Path

3. The **POT_ASYNC_EN** macro can be used to enable asynchronous GPADC operations. Please note that for asynchronous operations, developers must not call asynchronous related APIs without guaranteeing that the previous asynchronous operation is finished. To ensure this, after calling the ad_gpadc_read_async() function, the code waits for the arrival of a signal, indicating the end of the current GPADC operation.

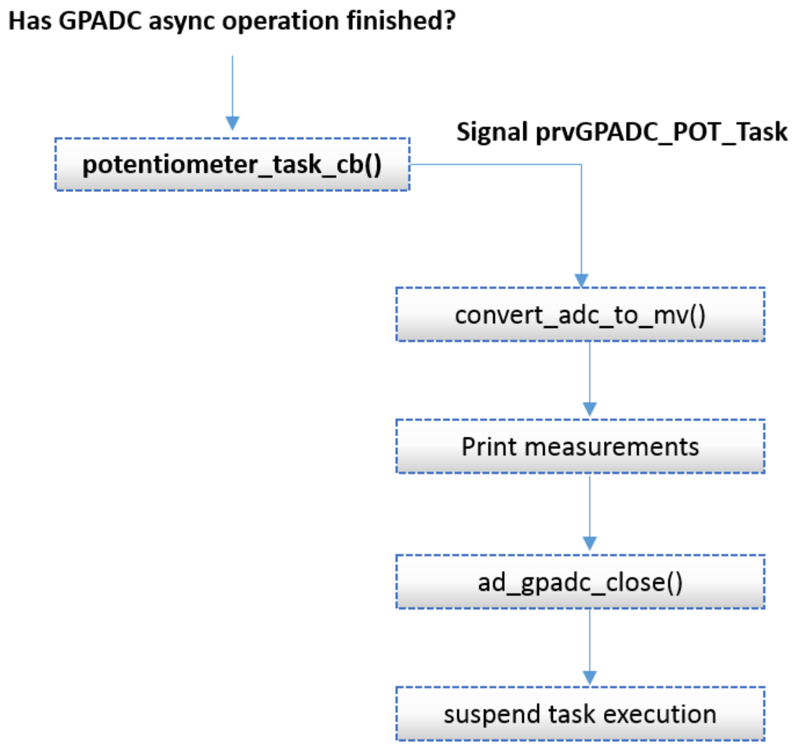

Figure 13: GPADC Async Conversions SW FSM – Callback Function Execution Path

4 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, a breadboard, a few jumper wires, a potentiometer, and a coin cell battery are required for testing and verifying the code. If you are not familiar with the recommended process on how to clone a project or configure a serial terminal, read the Starting a Project tutorial.

4.1 Verifying with a Serial Terminal

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 to the DA1468x SoC. For this tutorial a Pro DevKit is used.

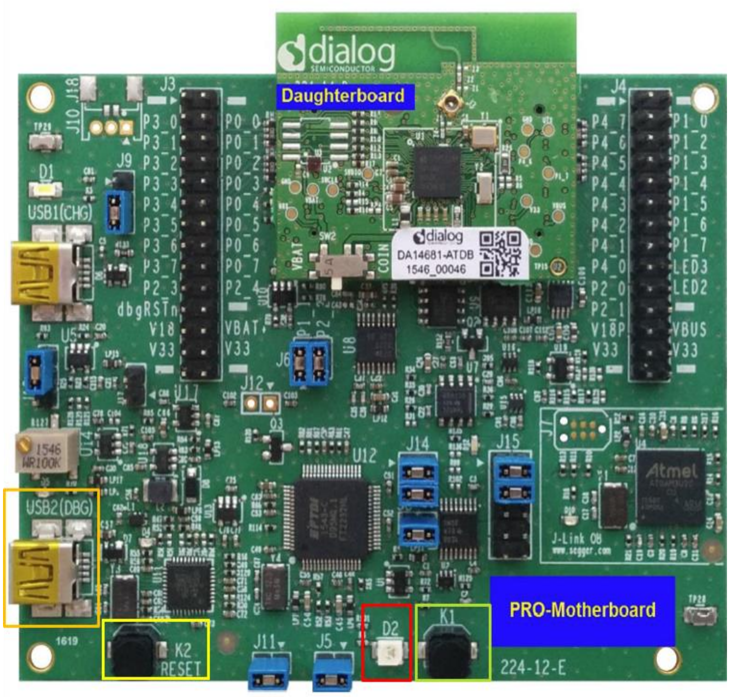

Figure 14: DA1468x Pro DevKit

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 newly created project, create a new **platform_devices.h** header file under the project's **/config** folder. To do this:

   a. Right-click on the /sdk/adapters/include/platform_devices.h header file (1) and select **Copy** (2).

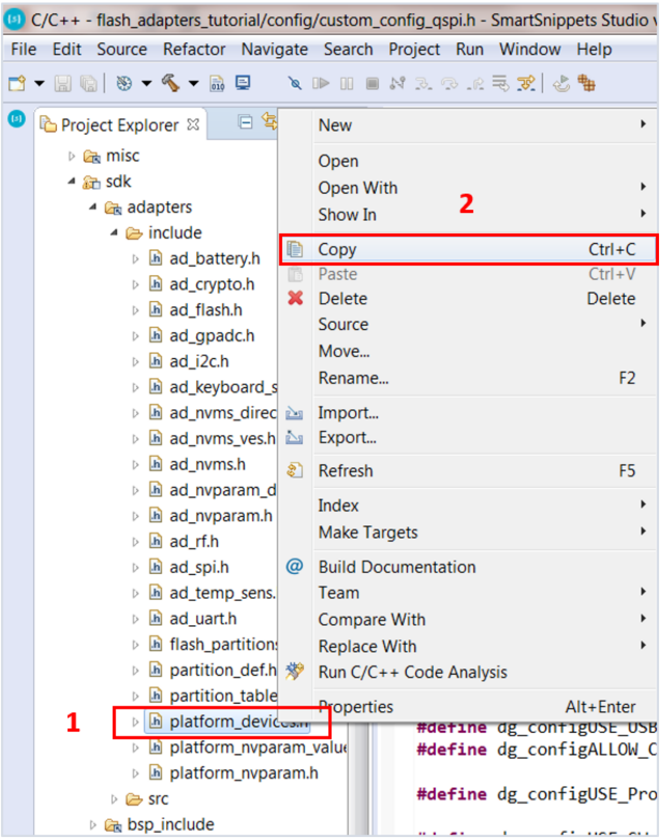

Figure 15: Creating platform_devices.h Header File, Step 1

b. Right-click on the **/config** folder (3) and select **Paste** (4).

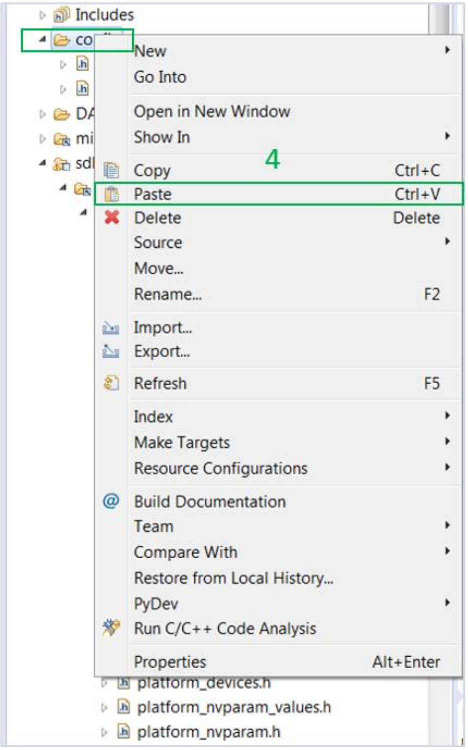

Figure 16: Creating platform_devices.h Header File, Step 2

Note: If a new platform_devices.h file is not created in the /config directory, the application will inherit the default macro definitions from /sdk/adapters/include/platform_devices.h.

4. 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. Hence, 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.

5. Build the project either in **Debug_QSPI** or **Release_QSPI** mode and burn the generated image to the chip.
6. Insert the coin cell battery (either rechargeable or non-rechargeable) in the daughterboard's battery holder.
   a. For this demonstration, a typical CR2023 non-rechargeable coin cell battery has been selected. The battery is placed at the bottom of the daughterboard in the dedicated battery holder as illustrated in Figure 15. The battery is inserted by first sliding it under the metallic clip of the battery holder. Extra attention is needed when removing the coin cell battery from its holder, if this is not done properly then the plastic battery holder can be subject to breaking.

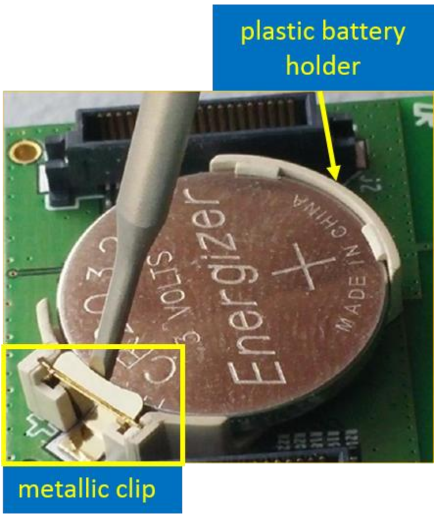

Figure 17: Rear View of the DA1468x Daughterboard

b. Select the power supply source through switch **SW2**. Power supply selection is available between the motherboard and the coin cell battery, thus allowing the daughterboard to operate as a standalone device once programmed. For this tutorial, the position of the **SW2** switch does not matter.

Figure 18: Power Supply Selection

7. Connect the potentiometer to the Pro DevKit as illustrated in Figure 17. For this demonstration, a simple 3-terminal potmeter has been selected. Two terminals are connected to the main power source, that is 3.3 V and GND, and the third to the selected ADC pin on Pro DevKit, that is **P1_2**.

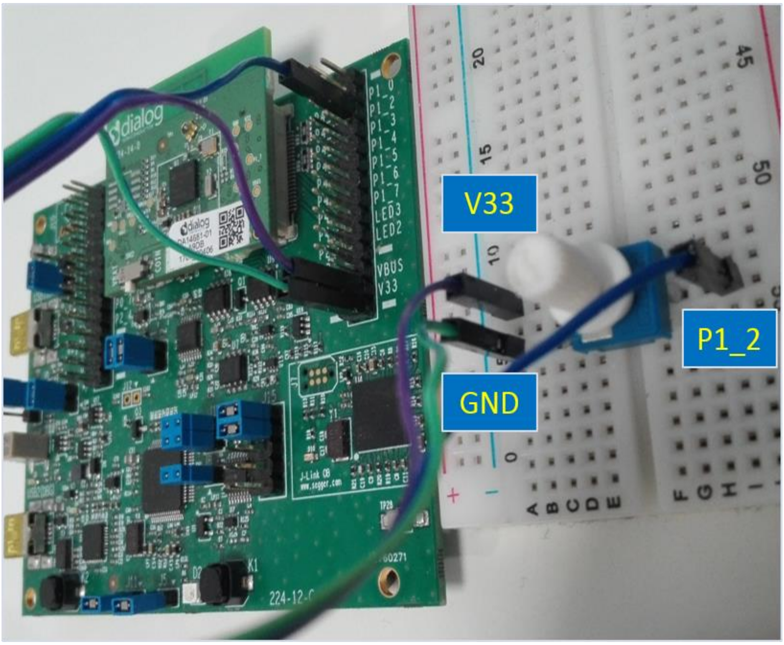

Figure 19: Snapshot of the Potmeter Connected to Pro DevKit

8. Open a serial terminal (115200, 8-N-1) and press the **K2** button on Pro DevKit. This step starts the chip executing its firmware.
9. Change the input voltage level of the **P1_2** pin (set as a GPADC pin) by rotating the potentiometer position both right and left. Observe the resulting analog-to-digital measurements on the console (1). Results should be updated every 1 second.

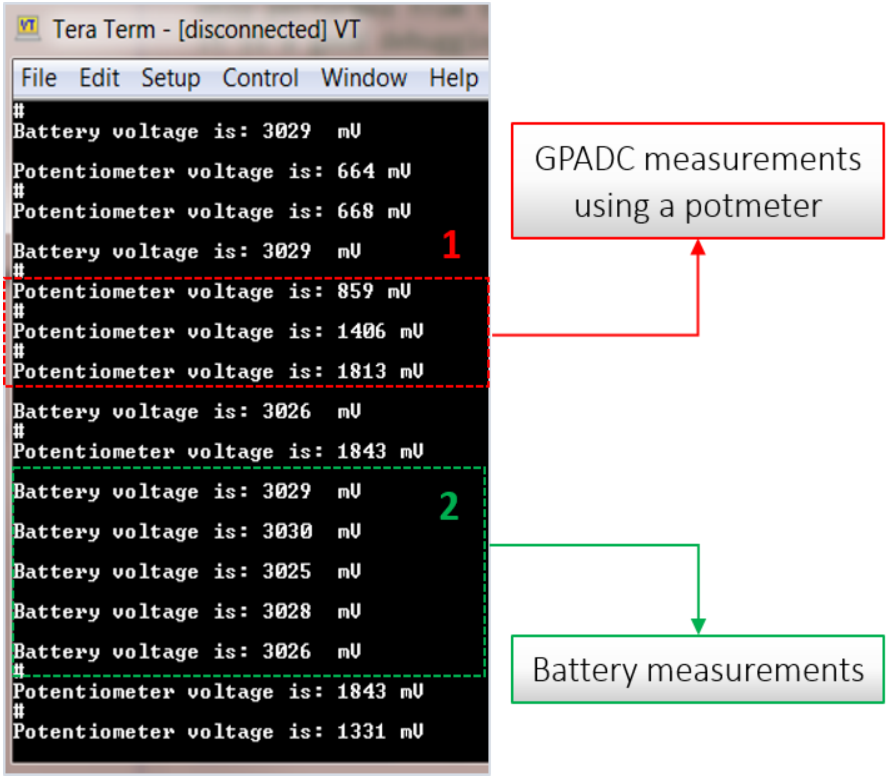

Figure 20: Debugging Messages for the Various Analog-to-Digital Operations

10. Press the **K1** button on Pro DevKit. A battery measurement is triggered and the result of the analog-to-digital conversion is printed out on the console (2).

5 Code Overview

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

5.1 Header Files

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

```
#include "ad_gpadc.h"
#include "hw_wkup.h"

#include <platform_devices.h>
```

5.2 System Init Code

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

```
/*
 * Macro for enabling asynchronous GPADC measurements.
 *
 * Valid values:
 * 0: GPADC conversions will follow a synchronous conversion scheme
 * 1: GPADC conversions will follow an asynchronous conversion scheme
 *
 */
#define POT_ASYNC_EN   (1)

/* OS signals used for synchronizing OS tasks */
OS_EVENT signal_bat;
OS_EVENT signal_pot;

/* GPADC Task priority */
#define mainGPADC_TASK_PRIORITY            ( OS_TASK_PRIORITY_NORMAL )

/*
 * GPADC application tasks - Function prototype
 */
static void prvGPADC_Battery_Task( void *pvParameters );
static void prvGPADC_POT_Task( void *pvParameters );

int convert_adc_to_mv(gpadc_source src, uint16_t value);
int convert_mv_to_adc(gpadc_source src, uint16_t value) __attribute__((unused));


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

        /* Set the desired sleep mode. */
        pm_set_sleep_mode(pm_mode_extended_sleep);

        /* Initialize the OS event signals. */
        OS_EVENT_CREATE(signal_bat);
        OS_EVENT_CREATE(signal_pot);

        /* 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 battery voltage measurements
         */
        OS_TASK_CREATE("GPADC_BATTERY",

                prvGPADC_Battery_Task,
                NULL,
                200 * OS_STACK_WORD_SIZE,

                mainGPADC_TASK_PRIORITY,
                task_bat_h );
        OS_ASSERT(task_bat_h);

        /*
         * Task responsible for GPADC voltage measurements
         */
        OS_TASK_CREATE("GPADC_POT",

                prvGPADC_POT_Task,
                NULL,
                200 * OS_STACK_WORD_SIZE,
```

```
            mainGPADC_TASK_PRIORITY,
            task_pot_h );
    OS_ASSERT(task_pot_h);


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

}
```

5.3 Wake-Up Timer Code

In **main.c**, after system_init(), add the following code for handling external events via the wake-up controller:

```
/*
 * 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-specified
     * interrupt callback, to clear the interrupt flag.
     */
    hw_wkup_reset_interrupt();

    /*
     * Notify [prvGPADC_Battery_Task] that time for
     * execution has elapsed.
     */
    OS_EVENT_SIGNAL_FROM_ISR(signal_bat);
}

/*
 * 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 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);
}
```

5.4 Hardware Initialization

In **main.c**, replace both **periph_init()** and **prvSetupHardware()** with the following code responsible for configuring pins after a power-up/wake-up cycle. Please note that every time the system enters sleep, it loses all its pin configurations.

```
/**
 * @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);


    /* LED D2 on ProDev Kit for debugging purposes */
    hw_gpio_set_pin_function(HW_GPIO_PORT_1, HW_GPIO_PIN_5,
                    HW_GPIO_MODE_OUTPUT, HW_GPIO_FUNC_GPIO);

    /* Configure P1.2 pin for ADC measurements */
    hw_gpio_set_pin_function(HW_GPIO_PORT_1, HW_GPIO_PIN_2,
                      HW_GPIO_MODE_INPUT, HW_GPIO_FUNC_ADC);
}

/**
 * @brief Hardware Initialization
 */
static void prvSetupHardware( void )
{
    /* Init hardware */
    pm_system_init(periph_init);
    init_wkup();
}
```

5.5 Raw ADC to mV Conversion Code

In **main.c**, after system_init(), add the following code responsible for converting a raw ADC value to mV:

```
/*
 * Function for converting a raw ADC value to mV
 *
 * \param [in]  src      The GPADC instance
 * \param [in]  value    The raw ADC value
 *
 * \return The converted raw ADC value in millivolt
 *
 */
int convert_adc_to_mv(gpadc_source src, uint16_t value)
{
    gpadc_source_config *cfg = (gpadc_source_config *)src;
    const uint16 adc_src_max =  ad_gpadc_get_source_max(src);
    uint32_t mv_src_max = (cfg->hw_init.input_attenuator ==
              HW_GPADC_INPUT_VOLTAGE_UP_TO_1V2) ? 1200 : 3600;
```

```
        int ret = 0;

        switch (cfg->hw_init.input_mode) {
        case HW_GPADC_INPUT_MODE_SINGLE_ENDED:
                if (cfg->hw_init.input == HW_GPADC_INPUT_SE_VBAT) {
                        mv_src_max = 5000;
                }
                ret =  (mv_src_max * value) / adc_src_max;
                break;
        case HW_GPADC_INPUT_MODE_DIFFERENTIAL:
                ret = ((int)mv_src_max * (value - (adc_src_max >> 1))) / (adc_src_max >> 1);
                break;
        default:
                /* Invalid input mode */
                OS_ASSERT(0);
        }

        return ret;
}
```

5.6 mV to Raw ADC Value Conversion Code

In **main.c**, after system_init(), add the following code responsible for converting mV to raw ADC values:

```
/*
 * Function for converting millivolt to raw ADC value
 *
 * \param [in]  src    The handler returned when calling the ad_battery_open() API
 * \param [in]  value  The ADC value expressed in mVolt e.g. 3050 (3.05V)
 *
 * \return    In single-ended mode: the converted raw ADC value.
 *            In differential mode: -1 (not supported)
 *
 */
int convert_mv_to_adc(gpadc_source src, uint16_t value)
{
        gpadc_source_config *cfg = (gpadc_source_config *)src;
        const uint16_t adc_src_max = ad_gpadc_get_source_max(src);
        uint32_t mv_src_max = (cfg->hw_init.input_attenuator ==
                        HW_GPADC_INPUT_VOLTAGE_UP_TO_1V2) ? 1200 : 3600;

        int ret = 0;

        switch(cfg->hw_init.input_mode) {
        case HW_GPADC_INPUT_MODE_SINGLE_ENDED:
                if (cfg->hw_init.input == HW_GPADC_INPUT_SE_VBAT) {
                        mv_src_max = 5000;
                }
                ret = (value * (adc_src_max + 1)) / (mv_src_max);
```

```
            break;
        case HW_GPADC_INPUT_MODE_DIFFERENTIAL:
            ret = -1; // Not supported!
            break;
        default:
            // Invalid input mode
            OS_ASSERT(0);
        }

        return ret;
}
```

5.7 Task Code for BAT Measurements

Code snippet of prvGPADC_Battery_Task task responsible for performing battery measurements. In **main.c**, add the following code (after system_init()):

```
/* Task responsible for battery voltage measurements */
static void prvGPADC_Battery_Task( void *pvParameters )
{
        battery_source bat_hdr;

        uint16_t bat_raw_adc;  // raw ADC value
        uint16_t bat_value_mv; // converted value to mVolt

        /*
         * Buffer to hold the debugging message
         */
        char dbg_message[40];
        memset(dbg_message, 0x20, sizeof(dbg_message));


        /*
         * GPADC adapter initialization should be done once at the beginning.
         * Alternatively, this function could be called during system initialization in
         * system_init().
         */
        GPADC_INIT();


        for (;;) {

            /*
             * Suspend task execution - As soon as WKUP callback function
             * is triggered the task resumes its execution.
             */
            OS_EVENT_WAIT(signal_bat, OS_EVENT_FOREVER);
```

```
            /* Open battery interface */
            bat_hdr = ad_battery_open();

            /*
             * Read the raw ADC value of the battery. Based on the oversampling used
             * the returned value can vary from 10 to 16 bits,
            */
            bat_raw_adc = ad_battery_read(bat_hdr);


            /*
             * Convert the previously measured raw ADC value to millivolt (mV)
             */
            bat_value_mv = ad_battery_raw_to_mvolt(bat_hdr, (uint32_t) bat_raw_adc);


            /*
             * Prepare the message to be sent out on the serial console
             */
            sprintf(dbg_message, "\n\rBattery voltage is: %d  mV\n\r", bat_value_mv);

            printf("%s", dbg_message);
            fflush(stdout);

            /* Close the already opened battery interface */
            ad_battery_close(bat_hdr);

        }
}
```

5.8 Task Code for GPADC Measurements

Code snippet of prvGPADC_POT_Task task responsible for performing voltage measurements from an analog GPIO pin. In **main.c**, add the following code (after system_init()):

```
#if POT_ASYNC_EN == 1
/*
 * Callback function for GPADC asynchronous operations:
 *
 * \param [in]  user_data  Data that can be passed and used within the function
 * \param [out] value       The raw ADC value
 *
 */
void potentiometer_task_cb(void *user_data, int value)
{
      /* User can pass in and process data from here */
      uint16_t *user_ptr = (uint16_t *) user_data;

      /* Read the raw ADC value */
```

```
        *user_ptr = (uint16_t) value;

        /*
         * Signal [prvGPADC_POT_Task] task that time
         * for resuming has elapsed.
         */
        OS_EVENT_SIGNAL_FROM_ISR(signal_pot);
}
#endif

/* Task responsible for GPADC voltage measurements */
static void prvGPADC_POT_Task( void *pvParameters )
{
        gpadc_source pot_hdr;

        uint16_t pot_raw_adc;  // raw ADC value
        uint16_t pot_value_mv; // converted value to mV

        /*
         * Buffer for holding the debugging message
         */
        char dbg_message[40];
        memset(dbg_message, 0x20, sizeof(dbg_message));

        /*
         * Initialize GPADC interface. This should be done once at the beginning.
         * Alternatively, this function could be called during system initialization in
         * system_init() function.
         */
        ad_gpadc_init();


        for (;;) {

                /* Suspend task's execution for 1000 ms */
                OS_DELAY(OS_MS_2_TICKS(1000));

                /*
                 * Turn on the LED D2 on Pro DevKit indicating the beginning of a process.
                 */
                hw_gpio_set_active(HW_GPIO_PORT_1, HW_GPIO_PIN_5);

               /*
                * Open the GPADC interface used (POT_LEVEL)
                */
                pot_hdr = ad_gpadc_open(POT_LEVEL);


#if POT_ASYNC_EN == 0
                /*
                 * Perform a synchronous GPADC operation.
```

```
         */
        ad_gpadc_read(pot_hdr, &pot_raw_adc);
#else
        /*
         * Perform an asynchronous GPADC operation.
         */
        ad_gpadc_read_async(pot_hdr, potentiometer_task_cb, (void *)
                                                &pot_raw_adc);

        /*
         * Wait until the current GPADC operation is finished
         */
        OS_EVENT_WAIT(signal_pot, OS_EVENT_FOREVER);
#endif
        /*
         * Convert the previously measured raw ADC value to millivolt (mV)
         */
        pot_value_mv = (uint16_t) convert_adc_to_mv(pot_hdr, pot_raw_adc);

        /*
         * Prepare the message to be sent out on the serial console.
         */
        sprintf(dbg_message, "\n\rPotentiometer voltage is: %d mV\n\r",
                                                pot_value_mv);

        printf("%s", dbg_message);
        fflush((stdout));

        /* Close the already opened device */
        ad_gpadc_close(pot_hdr);

        /*
         * Turn off LED D2 on ProDev kit indicating the end of a process.
         */
        hw_gpio_set_inactive(HW_GPIO_PORT_1, HW_GPIO_PIN_5);

    }
}
```

5.9 Macro Definitions

In /config/custom_config_qspi.h, add the following macro definitions:

```
/* Enable the GPADC instance used for POT measurements */
#define GPADC_POT

/*
 * Macros for enabling GPADC + Battery measurement operations using Adapters
 */
#define dg_configUSE_HW_GPADC         (1)
```

```
#define dg_configGPADC_ADAPTER        (1)
#define dg_configBATTERY_ADAPTER      (1)
/*
 * This macro should already be declared in the file.
 * Just make sure it is set to 1
 */
#define dg_configUSE_HW_TEMPSENS      (1)
```

5.10 GPADC HW Configuration Macros

In the newly created platform_devices.h, add the following macro definition between #if dg_configGPADC_ADAPTER and #endif

```
#ifdef GPADC_POT
GPADC_SOURCE(POT_LEVEL, HW_GPADC_CLOCK_INTERNAL,
 HW_GPADC_INPUT_MODE_SINGLE_ENDED, HW_GPADC_INPUT_SE_P12, 5, true,
                        HW_GPADC_OVERSAMPLING_2_SAMPLES,
HW_GPADC_INPUT_VOLTAGE_UP_TO_3V6)
#endif
```

Note: By default, the SDK comes with a few predefined device settings in platform_devices.h. Therefore, the developer should check whether an entry matches with a device connected to the controller.

Revision History

Revision	Date	Description
1.0	19-Mar-2018	First released version
1.1	14-May-2018	Correction of typos, Updated code in section 5.7 (missing configurations)
2.0	23-July-2018	More descriptive steps to follow, figures and examples.
2.1	19-Sep-2018	Updated figures, Minor improvements in *prvGPADC_POT_Task.*

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:

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:

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

enquiry@diasemi.com www.dialog-semiconductor.com