Tutorial

I2C Adapters

For the DA1468x SoC

Abstract

This tutorial should be used as a reference guide to gain a deeper understanding of the 'I2C 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 I2C transaction 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
DMA	Direct Memory Access
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 I^2C peripheral adapters
- Give a complete sample project demonstrating the usage of I^2C peripheral adapters

1.2 I^2C Adapters Introduction

This tutorial explains I^2C adapters and how to configure the DA1468x family of devices as an I^2C Master device. Adopting an I^2C Slave role is not used for the majority of situations and so is not covered in this tutorial. The I^2C adapter is an intermediate layer between the I^2C Low Level Drivers (LLDs) and a user application. It allows the user to utilize the I^2C interface in a simpler way than when using APIs from LLDs. The key features of I^2C 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.
- A DMA channel can be used among various peripherals (for example, I^2C, UART). Interconnected peripherals may use the same DMA channel if necessary. The adapter takes care of DMA channel resource management.
- It ensures that only one device can use the I^2C bus after acquiring it.
- Placing code between ad_i2c_bus_acquire() and ad_i2c_bus_release() ensures that only one task can use the I^2C bus to communicate with an external connected device. During this period no other device or task can use the I^2C interface until the ad_i2c_bus_release() function is called by the owning task.
- Power Manager (PM) of the chip is aware of the I^2C peripheral usage and before the system enters sleep, it checks whether or not there is activity on the I^2C bus.

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.

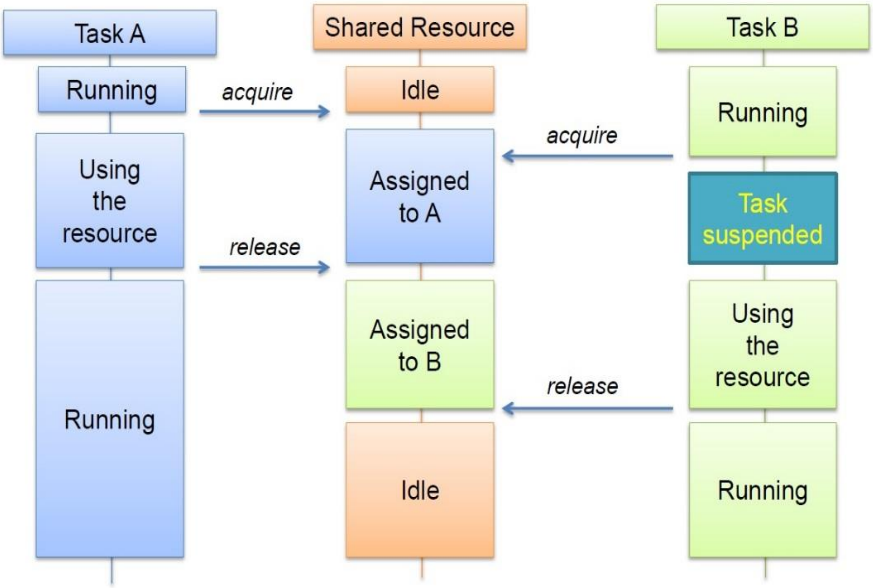

Figure 1: Adapters Communication

2 I²C Adapters Concept

This section explains the key features of I²C peripheral adapters as well as the procedure to enable and correctly configure the peripheral adapters for I²C functionality. The procedure is a four-step process which can be applied to almost every type of adapter including serial peripheral adapters (I²C, SPI, UART) and GPADC adapters.

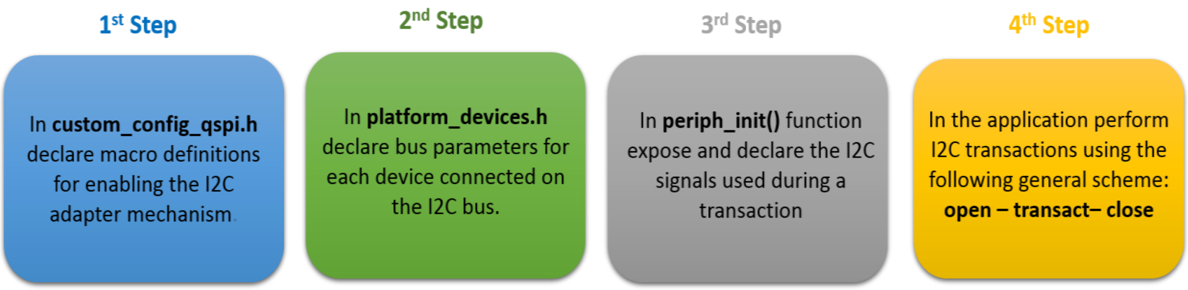

Figure 2: The Four-Step Process for Setting an Adapter Mechanism

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 I2C peripheral hardware block. Table 1 briefly explains the header files related to I2C adapters (red indicates the path under which the files are stored while green indicates which ones are used for I2C operations).

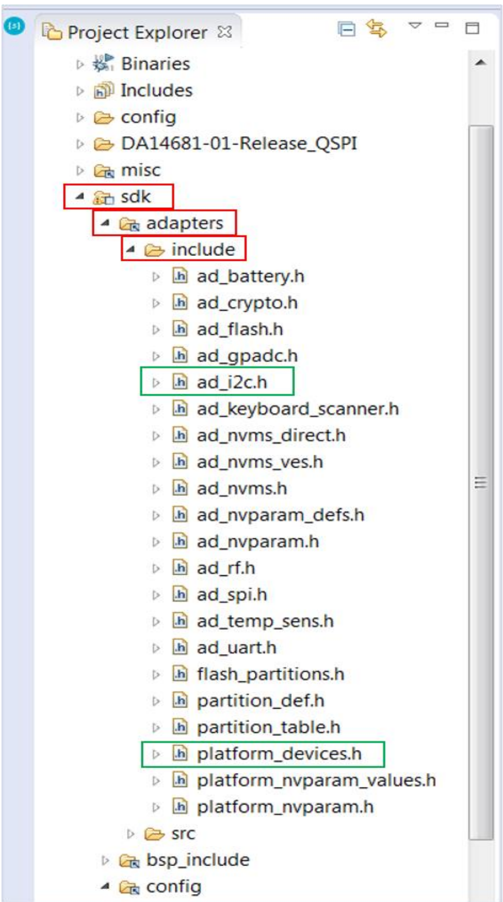

Figure 3: Headers for I2C Adapters.

Table 1: Header Files used by I2C Adapters

Filename	Description
ad_i2c.h	This file contains the recommended APIs and macros for performing I²C operations. Use these APIs when accessing the I²C peripheral bus.
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, I²C, UART) or a peripheral hardware block (for example, GPADC).

2.2 Preparing the I²C Adapter

1. As illustrated in Figure 4, the first step for configuring the I2C adapter mechanism is to enable it by defining the following macros in /config/custom_config_qspi.h:

```
/*
 * Macros for enabling I2C operations using Adapters
 */
#define dg_configUSE_HW_I2C                     (1)
#define dg_configI2C_ADAPTER                    (1)
```

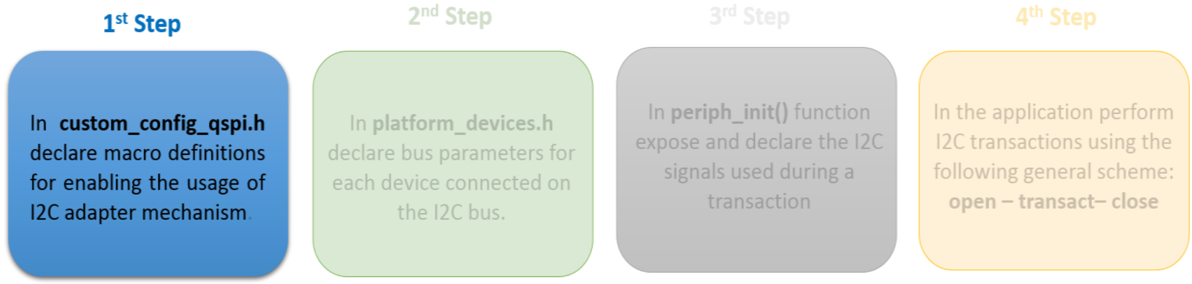

Figure 4: First Step for Configuring the I2C Adapter Mechanism

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

2. The second step is to declare all the devices externally connected on the I²C bus. A device can be considered as a set of settings describing the complete I²C interface. These settings are applied every time the device is selected and used. To do this, the SDK exhibits two macros, named **I2C_SLAVE_DEVICE_DMA** and **I2C_SLAVE_DEVICE** respectively. The first one is used when a DMA mechanism is used during a transaction.

```
/*
 * Macro for setting I2C bus parameters
 */
I2C_SLAVE_DEVICE_DMA(bus, name, _address_, _addr_mode, _speed,

                                                      _dma_channel)
```

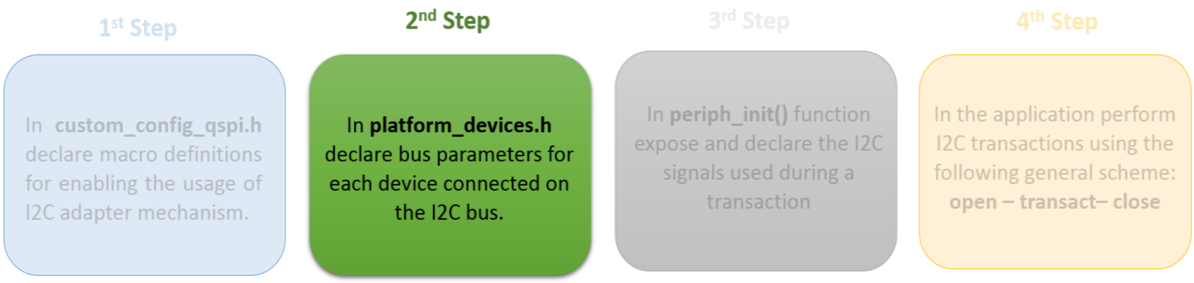

Figure 5: Second Step for Configuring the I2C Adapter Mechanism

Table 2: Description of the Macro Fields, Used for Declaring an I2C Device

Argument Name	Description
bus	The DA1468x family of devices features two distinct I²C hardware blocks. Valid values are I2C1 and I2C2.
name	Declare an arbitrary alias for the I²C interface (for instance, My_slave_device). This name should be used for opening that specific device.
_address	The address to which the externally connected slave device listens. The value is device-specific information and is usually found in the manufacturer's datasheet.

_addr_mode	The I^{2}C controller supports both the 7-bit and 10-bit addressing modes. Valid values are those from HW_I2C_ADDRESSING enum in /sdk/peripherals/include/hw_i2c.h.
_speed	The I^{2}C controller supports two different speed modes: 100 kHz and 400 kHz respectively. Valid values are those from HW_I2C_SPEED enum in /sdk/peripherals/include/hw_i2c.h.
_dma_channel	The DA1468x family of devices features eight general-purpose DMA channels that can be used for various transactions. This field defines the DMA number for the RX channel. TX will have the next number and it is automatically assigned by the adapter mechanism.

Note: The I2C_SLAVE_DEVICE() macro has the same syntax as I2C_SLAVE_DEVICE_DMA except for the last parameter, that is _dma_channel. Also, note that DMA RX/TX channels must be used in pairs, that is, 0/1, 2/3, 4/5, and 6/7. Thus, the RX channel must always be set to an even number (0, 2, 4, 6).

The DA1468x family of devices features two distinct I^{2}C blocks namely I2C1 and I2C2. Depending on the I^{2}C interface used, device configurations must be placed between the correct macro indicators:

```
/* Declare I2C bus configurations for devices connected to I2C1 hardware block */
I2C_BUS(I2C1)

/*
 * Use I2C_SLAVE_DEVICE() and/or I2C_SLAVE_DEVICE_DMA() for each
 * device declaration.
 */

I2C_BUS_END


/* Declare I2C bus configurations for devices connected to I2C2 hardware block */
I2C_BUS(I2C2)

/*
 * Use I2C_SLAVE_DEVICE() and/or I2C_SLAVE_DEVICE_DMA() for each
 * device declaration.
 */

I2C_BUS_END
```

3. As illustrated in Figure 6, the third step is the declaration of the I^{2}C signals. The user can multiplex and expose I^{2}C signals on any available pin on DA1468x SoC.

```
static void prvSetupHardware( void )
{

   /* Init hardware */
   pm_system_init(periph_init)

}
```

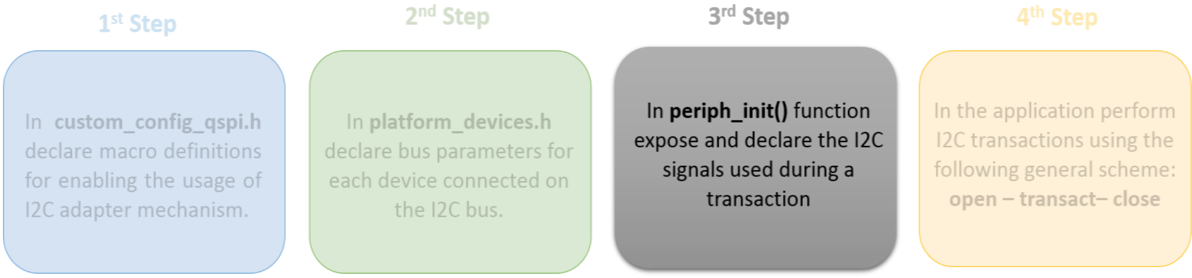

Figure 6: Third Step for Configuring the I2C Adapter Mechanism

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.

4. Once the I²C adapter mechanism is enabled, the developer can use all the available APIs for performing I²C transactions. The following steps describe the required sequence of APIs in an application to successfully execute an I²C write/read operation.

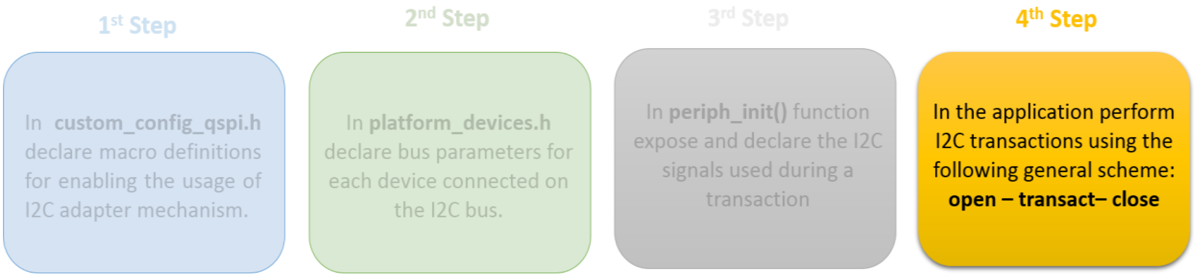

Figure 7: Fourth Step for Configuring the I2C Adapter Mechanism

a. ad_i2c_init()

   This 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_i2c_open()

   Before using the I²C interface, the application task must open the device that will access the bus. Opening a device involves enabling the I²C controller. If the device is the only connected device on the I²C bus, configuration of the I²C controller also takes place. This function returns a handler to the main flow for use in subsequent adapter functions. Subsequent calls from other tasks simply return the already existing handler.

c. ad_i2c_bus_acquire()

   This API is optional since it is automatically called upon a write/read transaction and is used for locking the I²C bus for the given opened device. This function should be called when the application task wants to communicate to the I²C bus directly using low level drivers.

Note: The function can be called several times. However, it is essential that the number of calls must match the number of calls to ad_i2c_bus_release().

d. Perform a write/read transaction either synchronously or asynchronously.

After opening a device, the application task(s) can perform any read/write I²C transaction either synchronously or asynchronously. Please note that all the available APIs for writing/reading over an I²C bus, nest the corresponding APIs for acquiring and releasing a device.

e. ad_i2c_bus_release()

This function must be called for each call to ad_i2c_bus_acquire().

f. ad_i2c_close()

After all user operations are done and the device is no longer needed, it should be closed by the task that has currently acquired it. The application can then switch to other devices connected on the same I²C bus. Remember that the I²C adapter implementation follows a single device scheme, that is only one device can be opened at a time.

2.3 I²C Transactions

Write and read functions can be divided into two distinct categories:

- Synchronous Mode
- Asynchronous Mode

2.3.1 Synchronous Mode

In synchronous mode, the calling task is blocked for the duration of the write/read access but other tasks are not. Code initially waits for the I²C bus to become available and then blocks the calling task until a transaction is completed. Once a write/read process is finished, the I²C bus is freed and further write/read transactions over the I²C bus can take place.

Code snippet of a typical write followed by a read synchronous I²C transaction:

```
// Open the device that will utilize the I2C bus
i2c_device dev = ad_i2c_open(My_Slave_Device);

// Perform I2C transactions to the already opened device
ad_i2c_transact(dev, command, sizeof(command), response, sizeof(response));

// Close the already opened device
ad_i2c_close(dev);
```

The above code performs a write transaction followed by a read transfer, an operation which is typical when reading data from I²C peripherals. In such cases, an address needs to be specified through a write before reading data. The function first waits for both the device and bus resources to become available, before proceeding with the write without waiting for the STOP condition. If no error occurs by the time the last byte is placed in the transmit FIFO of the DA146x SoC, the function continues with the read operation and waits until it is completed.

Note: The aforementioned API can also be used for write only or read only transactions by providing a NULL pointer in the corresponding input parameter. For example, to perform a write only operation: ad_i2c_transact(dev, command, sizeof(command), NULL, 0);

2.3.2 Asynchronous Mode

In asynchronous mode, the calling task is not blocked by the write or read operation. It can continue with other operations while waiting for a dedicated callback function to be called, signaling the completion of the read or write transaction. I2C adapters allow a developer to perform I2C transactions that consist of a number of reads, writes, and callback calls. This provides a time-efficient way to manage all I2C related actions. Most of the actions are executed within ISR context. There are a number of arguments-actions that should be used to perform various I2C transaction schemes. Table 3 explains all the available arguments that can be used to configure an I2C transaction scheme.

Table 3: Available Arguments for Configuring I2C Asynchronous Transactions

Argument Name	Description
I2C_SND()	Use this argument to send data over the I^2C bus, **without waiting** for a STOP condition to be issued by the master device.
I2C_SND_ST()	Use this argument to send data over the I^2C bus and **wait** for a STOP condition to be detected.
I2C_RCV()	Use this argument to read data over the I^2C bus. A STOP condition **is generated** after receiving the last byte.
I2C_RCV_NS()	Use this argument to read data over the I^2C bus. A STOP condition **is not generated** after receiving the last byte.
I2C_CB()	Declare a callback function that should be called when finishing with all defined I^2C actions. The developer **cannot** pass data in the callback function.
I2C_CB1()	Declare a callback function that should be called when finishing with all defined I^2C actions. The developer **can** pass data in the callback function.
I2C_END	Use this argument to mark the end of an I^2C transaction scheme. This argument should be the last argument passed.

Code Snippet of a typical write followed by a read asynchronous I^2C transaction:

```
// Open the device that will utilize the I2C bus
i2c_device dev = ad_i2c_open(My_Slave_Device);

// Perform I2C transactions to the already opened device
ad_i2c_async_transact(dev, I2C_SND(command,  sizeof(command)) , // I2C write operation
                           I2C_RCV(response, sizeof(response)), // I2C read  operation
                           I2C_CB (final_callback), // User-defined callback function
                           I2C_END);            // Indicate the end of I2C operations

// Close the already opened device
ad_i2c_close(dev);
```

When using I²C operations in asynchronous mode, the following should be considered:

- Callback functions are called from within Interrupt Service Routine (ISR) context. Therefore, callback's execution time should be as short as possible and not contain complex calculations. Please note that for as long as a system interrupt is serviced, the main application is halted.
- If the callback function is the last action to be performed, then resources (I²C device and bus) are released before the callback is called.
- Do not call asynchronous related APIs consecutively without guaranteeing that the previous asynchronous transaction is finished.
- After the callback function is called, it is not guaranteed that the scheduler will give control to the freeRTOS task waiting for that transaction to complete. This is important to consider if several tasks are using this API.

Note: All the write/read I2C related APIs return a code which can be used to indicate whether an I2C operation has been successfully executed or not. All the possible values are declared in HW_I2C_ABORT_SOURCE enum located in /sdk/peripherals/include/hw_i2c.h.

2.4 I²C Related Macros

I²C adapters have macros for facilitating various management schemes and can be used as required by the developer. The available macros can be found in /sdk/adapters/include/ad_i2c.h. It is recommended that any macro definition is put in the platform_devices.h header file. The most frequently used macros are explained in Table 4.

Table 4: I2C Macros

Macro Name	Description
CONFIG_I2C_EXCLUSIVE_OPEN	Set this macro to '1' to prevent multiple tasks from opening the same device. When set to '1' ad_i2c_device_acquire() and ad_i2c_device_release() are no longer necessary.
CONFIG_I2C_ONE_DEVICE_ON_BUS	Set this macro to '1' if only one I²C device is connected on the bus (one on I2C1 and one on I2C2). This will reduce code size and improve performance.

3 Analyzing The Demonstration Example

This section analyzes an application example which demonstrates using the I²C adapters. The example is based on the **freertos_retarget** sample code found in the SDK. It adds an additional freeRTOS task which is responsible for controlling an external I²C module, connected on I2C1 bus. It also enables the wake-up timer for handling external events. Both synchronous and asynchronous I²C operations are demonstrated.

3.1 Application Structure

1. The key goal of this demonstration is for the device to perform a few I²C operations following an event. For demonstration purposes, the **K1** button on the Pro DevKit has been configured as a wake-up input 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, a variable named i2c_status is toggled. It can take two different values which are interpreted as follows:

- **i2c_status = 1**
  - An asynchronous I²C write operation is attempted. A whole page of 64 bytes is written in EEPROM starting from the physical address 0x0000. Note that the page size of an EEPROM is device-specific and we recommend reading the manufacturer datasheet for more information on this. At the end of the transaction, a debugging message is printed on the serial console indicating whether or not the I²C operation was successfully executed.

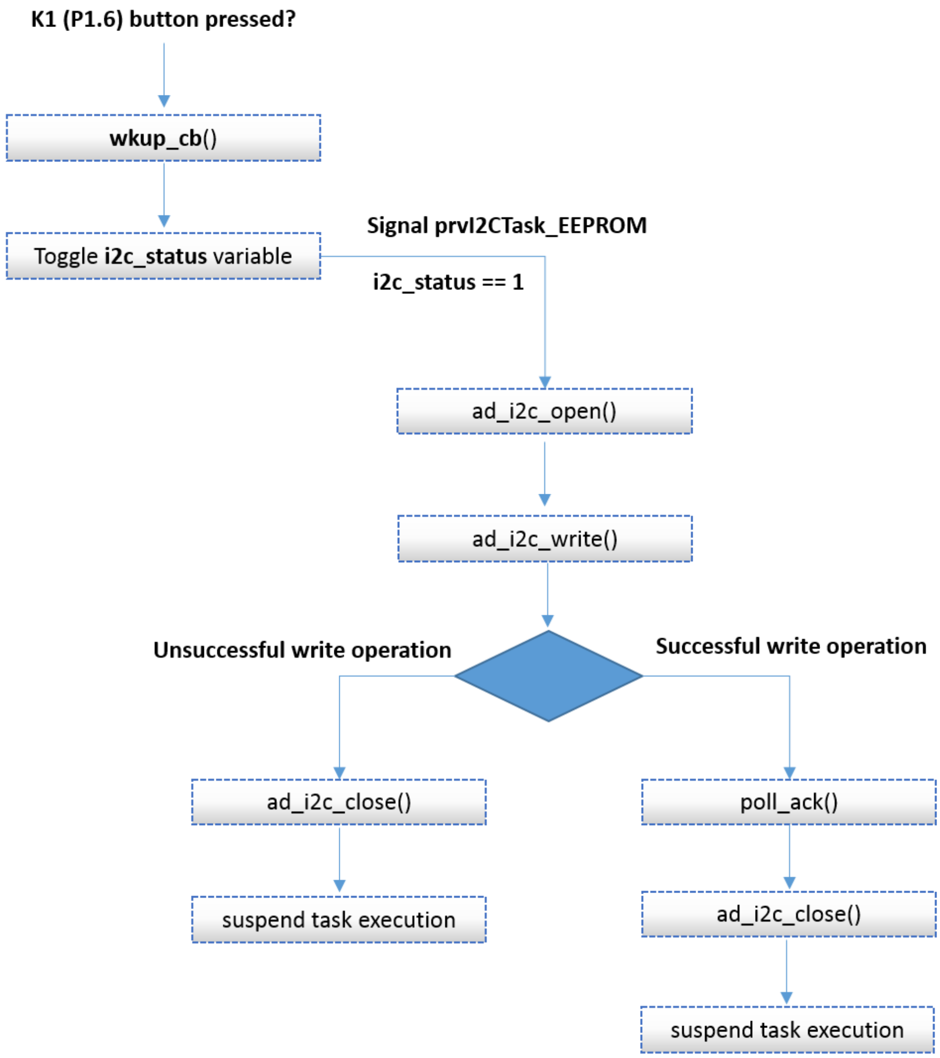

Figure 8: EEPROM Write SW FSM – Main Execution Path

- **i2c_status = 0**
  - Depending on the value of the I2C_ASYNC_EN macro, a synchronous or asynchronous I²C read operation is attempted. A whole page of 64 bytes is read from EEPROM starting from the physical address 0x0000. In fact, the program attempts to read the previously written data in EEPROM. At the end of the transaction, a debugging message is printed on the serial console indicating whether or not the I²C operation was successfully executed. In addition, a data integrity check is performed and a corresponding debugging message is also printed on the serial console.

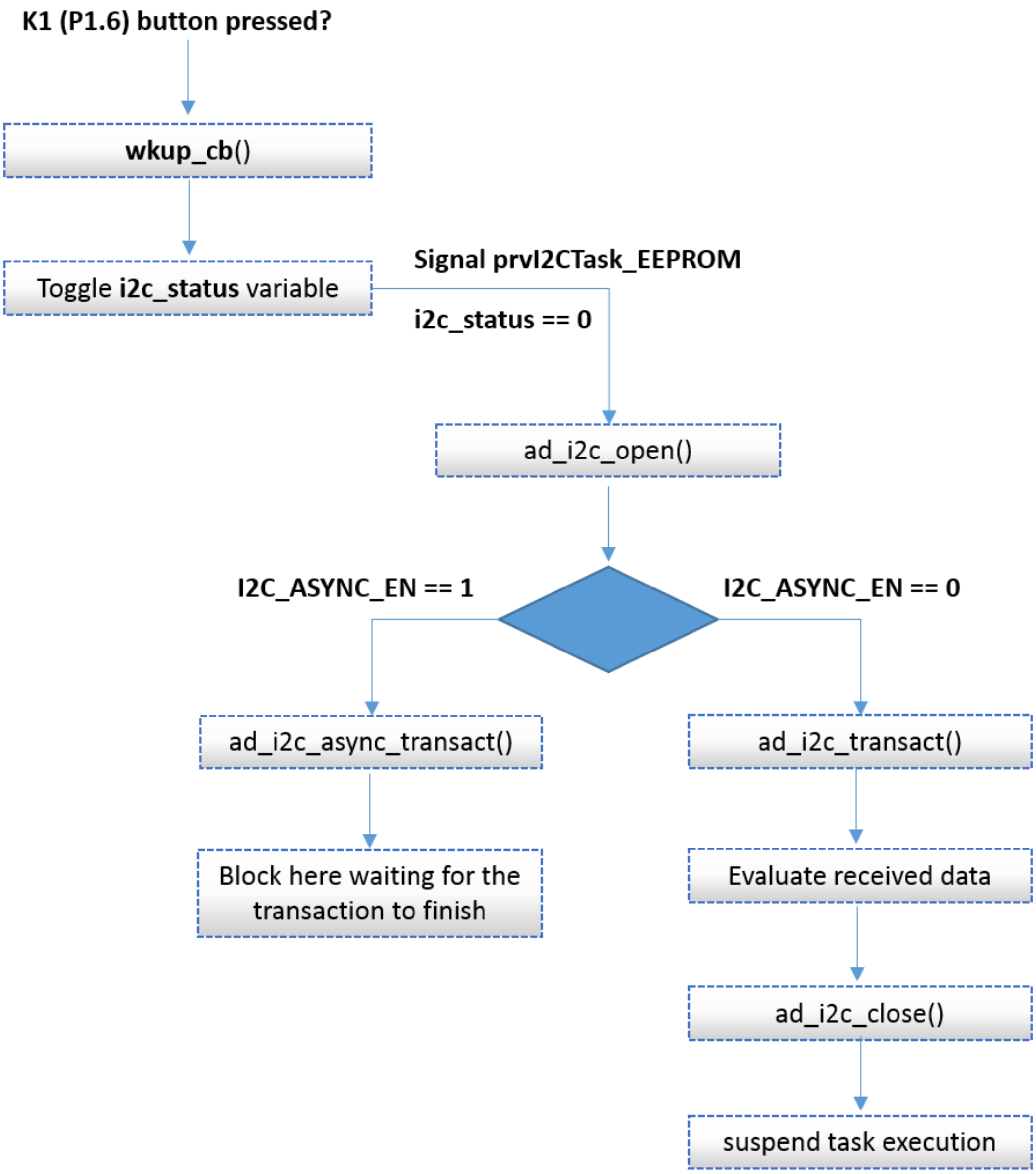

Figure 9: EEPROM Read SW FSM – Main execution path

2. The I2C_ASYNC_EN macro can be used to enable asynchronous I^2C read operations. As described in I2C Transactions, developers must not call asynchronous related APIs without guaranteeing that the previous asynchronous transaction is finished. To ensure this, after calling the ad_i2c_async_transact() function, the code waits for the arrival of a signal, indicating the end of the current I^2C operation.

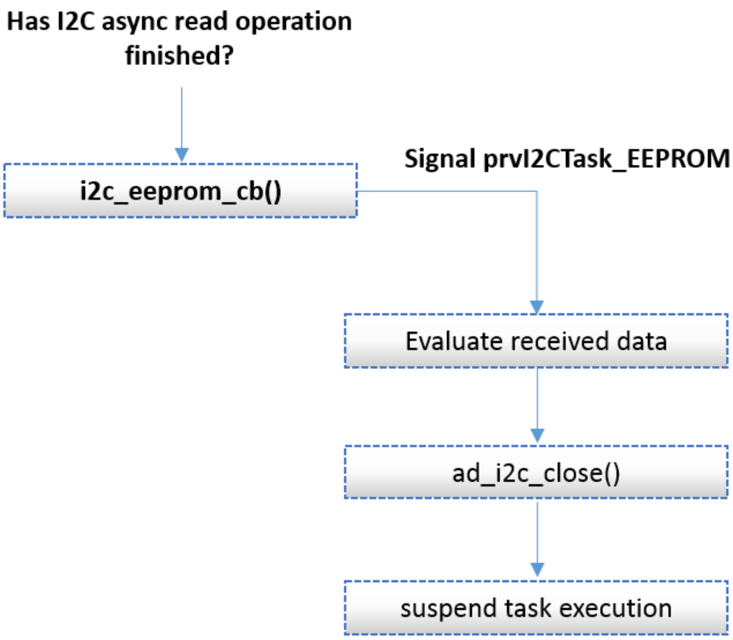

Figure 10: EEPROM Async Read SW FSM – Callback Function Execution Path

3. At this point, it is important to highlight one peculiarity encountered in EEPROM devices. Data sent from a master device to an EEPROM slave device is actually written in EEPROM cells after a STOP condition is issued by the master. At this point, the EEPROM starts its write cycle for storing the previous written data to its memory cells. This means that the code should not continue with the next write/read cycle without guaranteeing that the EEPROM device is ready to handle a new command. To ensure this, the function poll_ack() is called after a successful I^2C write operation. For more information on how to poll the EEPROM device to check its readiness, read the Acknowledge Polling section in the manufacturer datasheet.

Note: The 24LC256 EEPROM module has been selected for demonstration purposes only. Providing complete drivers for this module is out of the scope of this tutorial.

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 24LC256 EEPROM module, and optionally a logic analyzer are required for testing and verifying the code. In addition, two 2.4 kΩ resistors, a breadboard, and a few jumper wires are required to connect the I^{2}C module to the Pro DevKit. For information on configuring a serial terminal, as well as a Pro DevKit, read the Starting a Project tutorial.

There are two main methods to verify the correct behavior of the demonstrated code. The first method is to use a Serial Terminal and the second is to use a logic analyzer. Both cases are given below as a logic analyzer can be quite an expensive tool.

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.

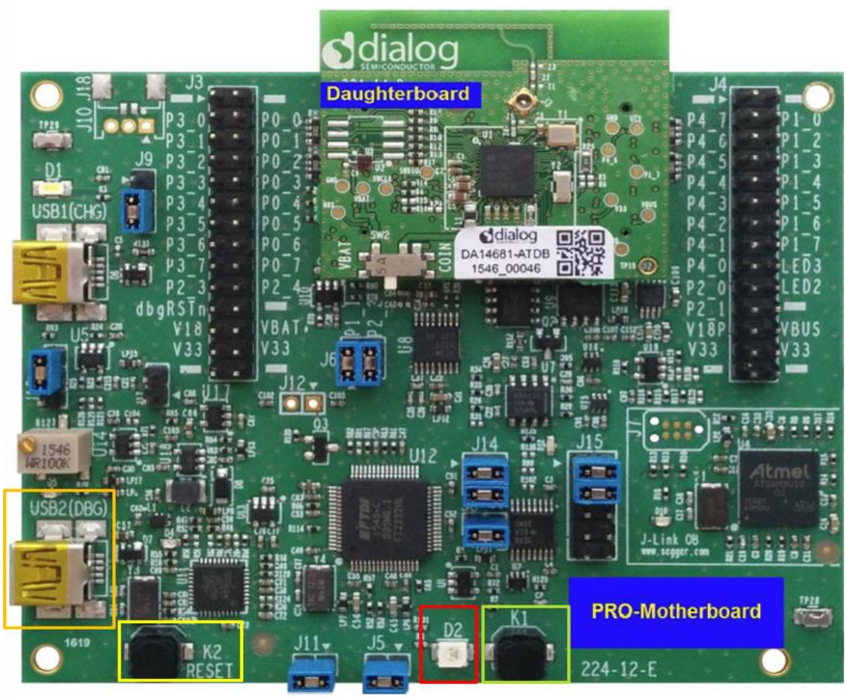

Figure 11: DA1468x Pro DevKit

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:

I2C Adapters

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

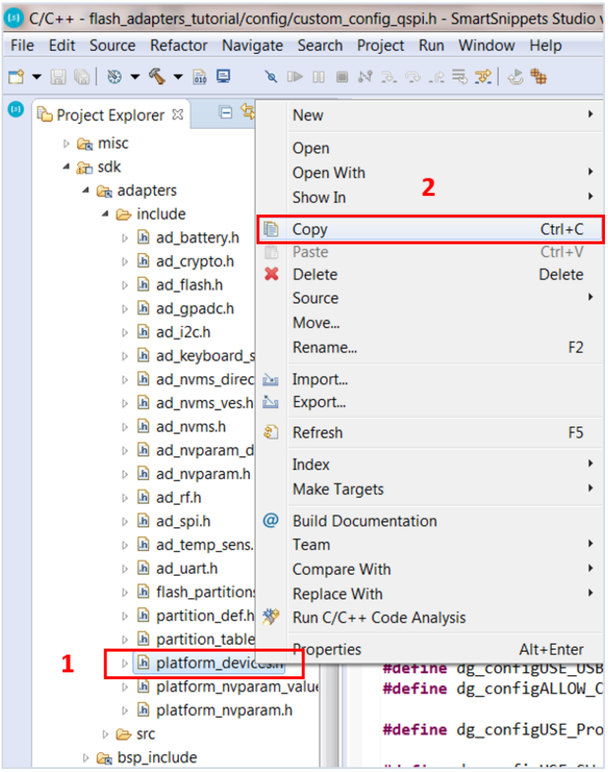

Figure 12: Creating a platform_devices.h Header File, Step 1

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

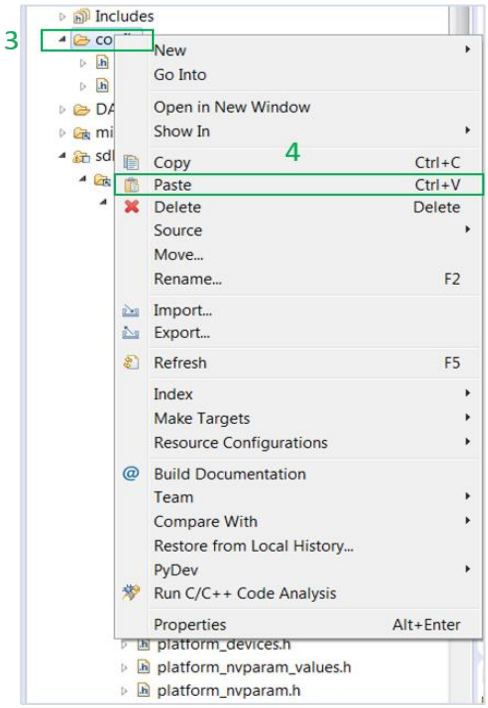

Figure 13: Creating a platform_devices.h Header File, Step 2

Note: If a new platform_devices.h file is not created in /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. 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.

5. Build the project either in **Debug_QSPI** or **Release_QSPI** mode and burn the generated image to the chip.
6. Connect the EEPROM module to the Pro DevKit. Figure 12 illustrates the pin connections required to configure the 24LC256 module. For more information on the EEPROM module used, read the manufacturer datasheet.

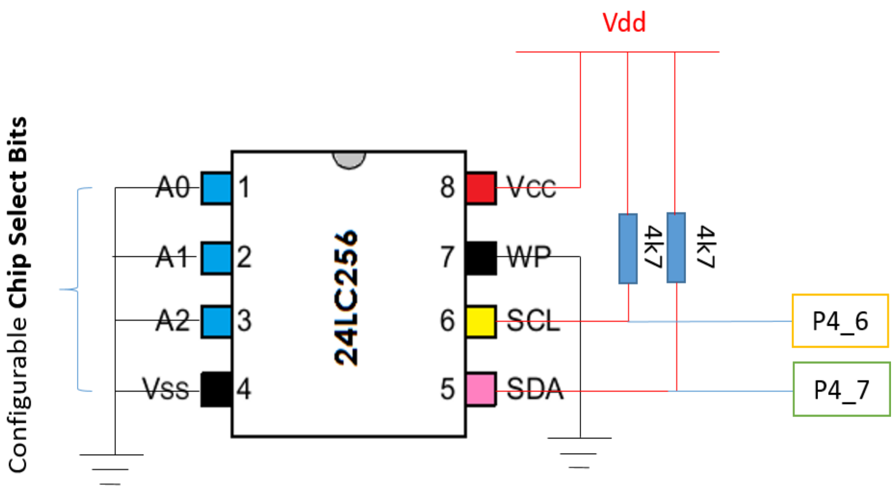

Figure 14: Configuring the 24LC256 EEPROM Slave Device

Since all the **Chip Select Bits** are connected to ground (that is, logic '0'), the resulting device address is set to **0b1010000** (0x50). Thus, the master device (DA1468x chip) should reference the EEPROM using this address.

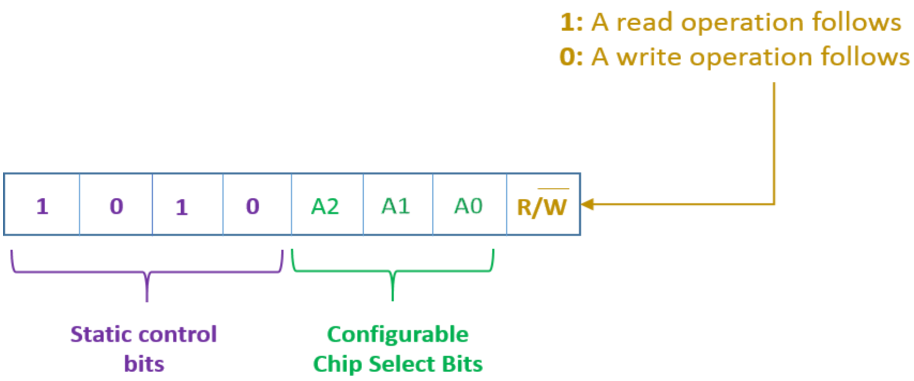

Figure 15: Control Byte of 24LC256 EEPROM Chip

Note: The Control Byte is the first byte sent by the I2C master controller (DA1468x) at each I2C operation. Given the current EEPROM configurations possible values are: 0b10100001 (0xA1) when a read command is executed or 0b10100000 (0xA0) when a write command is executed.

7. Press the **K2** button on Pro DevKit to start the chip executing its firmware.
8. Open a serial terminal (115200, 8-N-1) and press the **K1** button on Pro DevKit. A debugging message is displayed on the console (1) indicating both the progress and status of the I^2C write operation.
9. Press the **K1** button on Pro DevKit again. A debugging message is displayed on the console (2) indicating the status of the I^2C read operation. A message indicating whether or not read data matches the previous written data is also displayed.

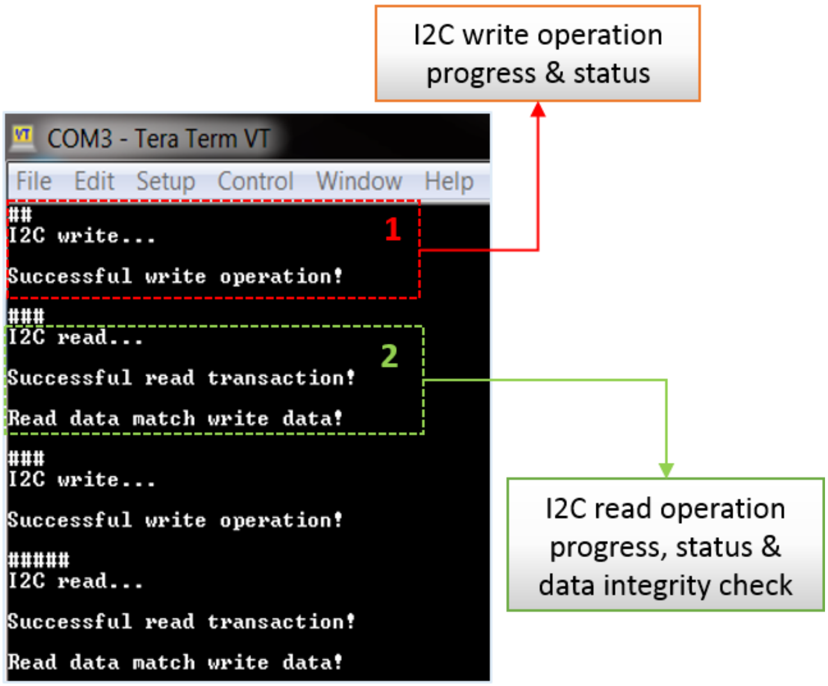

Figure 16: Debugging Messages Indicating both the Progress and Status of an I2C Transaction

4.2 Verifying with a Logic Analyzer

This step is optional and is intended for those who are interested in using an external logic analyzer to capture the I²C signals during a transaction.

1. With the whole system up and running, open the software that controls the logic analyzer. For this step a logic analyzer from Saleae Incorporation and its official software was used.
2. Connect the logic analyzer to the Pro DevKit. To do this, you should:
   a. Connect a channel from the logic analyzer to **P4_6** pin of Pro DevKit. This is the Clock signal (SCL).
   b. Connect a channel from the logic analyzer to **P4_7** pin of Pro DevKit. This is a bidirectional line both for sending and receiving data (SDA).
3. Press the **K2** button on Pro DevKit to reset the device.
4. Press the **K1** button on Pro DevKit and capture the I²C write transaction. Figure 15 illustrates only the first two of the total written bytes.

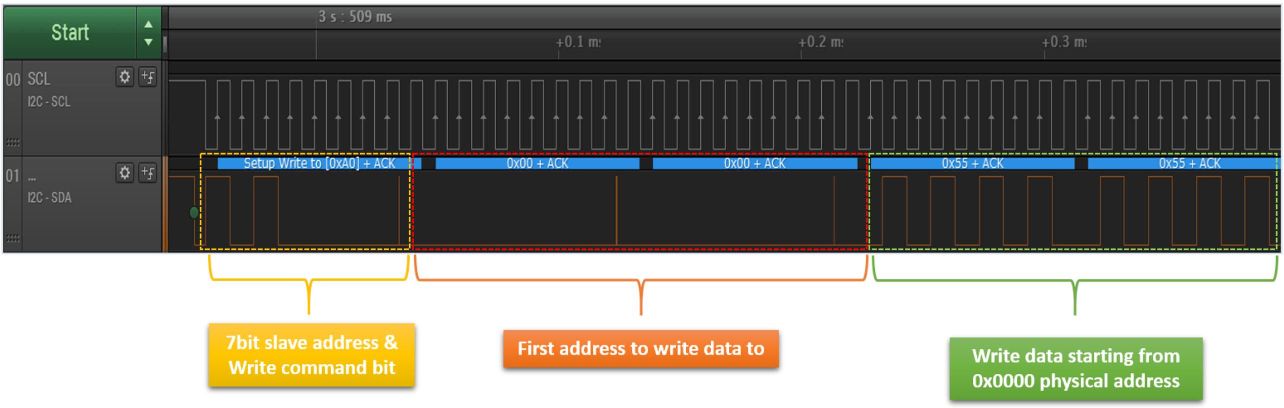

Figure 17: I2C Write Transaction Captured using a Logic Analyzer

5. If the I²C write operation was successfully executed, the Acknowledge Polling procedure takes place.

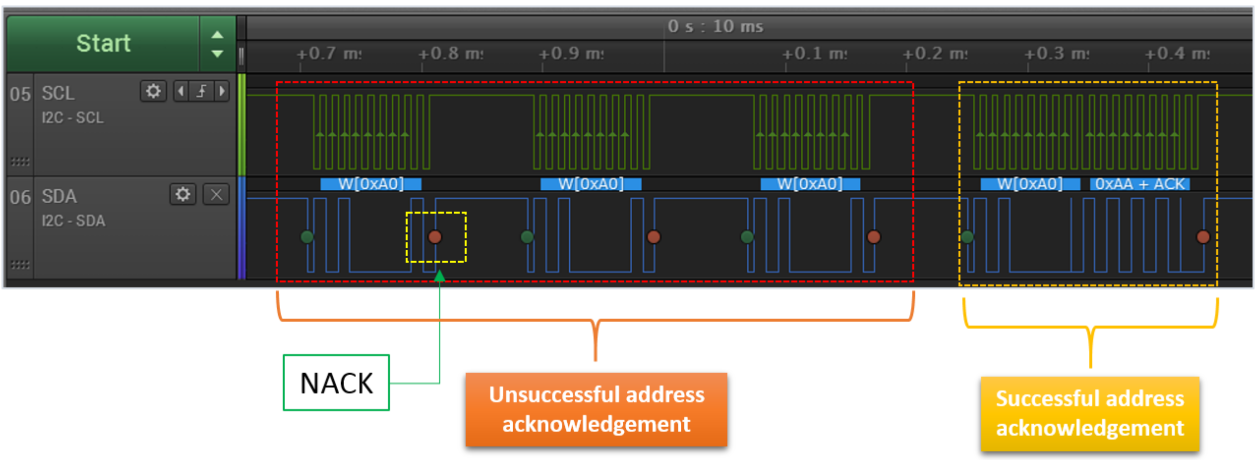

Figure 18: Polling ACK Procedure Captured using a Logic Analyzer

6. Press the **K1** button again on Pro DevKit and capture the I²C read transaction. Figure 15 illustrates only the first two of the total read bytes.

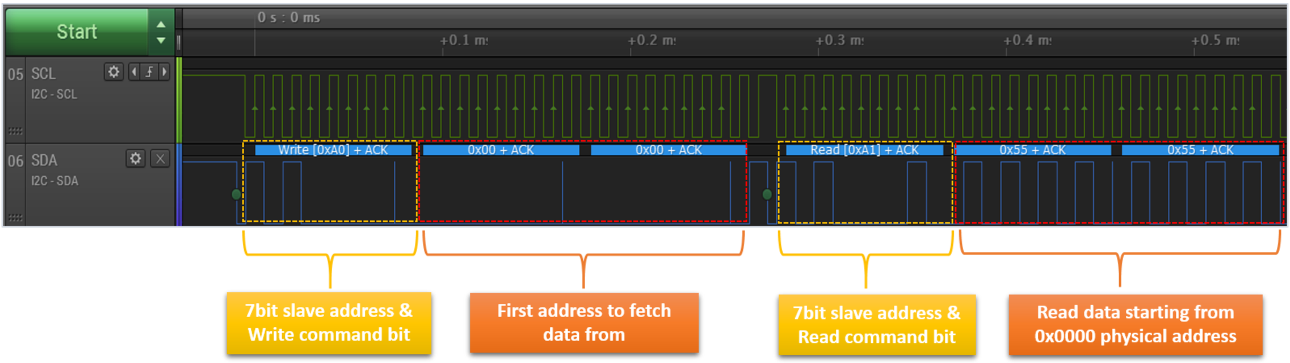

Figure 19: I2C Read Transaction Captured using a Logic Analyzer

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 "hw_wkup.h"

#include "ad_i2c.h"
#include <platform_devices.h>
```

5.2 System Init Code

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

```
/*
 * Macro for enabling asynchronous I2C transactions.
 *
 * Valid values:
 * 0: I2C transactions will follow a synchronous scheme
 * 1: I2C transactions will follow an asynchronous scheme
 *
 */
#define I2C_ASYNC_EN    (0)

/* OS signals used for synchronizing OS tasks*/
static OS_EVENT siganl_i2c_eeprom;
static OS_EVENT signal_i2c_eeprom_async;

/*
 * Status flag used for indicating whether a read
```

```
 * or write I2C operation will be performed.
 */
volatile static bool i2c_status = 0;

/* I2C task priority */
#define mainI2C_TASK_PRIORITY          ( OS_TASK_PRIORITY_NORMAL )

/*
 * I2C application tasks - Function prototype
 */
static void prvI2CTask_EEPROM ( void *pvParameters );
static void poll_ack (i2c_device dev);

static void system_init( void *pvParameters )
{
        OS_TASK task_h = NULL;
        OS_TASK i2c_eeprom = 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



        /* Initialize the OS event signals */
        OS_EVENT_CREATE(siganl_i2c_eeprom);
        OS_EVENT_CREATE(signal_i2c_eeprom_async);

        /* 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);


        /*
         * Create an I2C task responsible for controlling the
         * externally connected 24LC256 EEPROM module.
         */
        OS_TASK_CREATE("EPPROM_24LC256",

                    prvI2CTask_EEPROM,
                    NULL,
                    200 * OS_STACK_WORD_SIZE,

                    mainI2C_TASK_PRIORITY,
                    i2c_eeprom );
        OS_ASSERT(i2c_eeprom);

        /* 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();

        /*
         * Toggle I2C status:
         *
         * 1: a Write transaction will be performed
         * 0: a Read transaction will be performed
         */
        i2c_status ^= 1;

        /*
```

```
         * Notify [prvI2CTask_EEPROM] that time for
         * performing I2C operations has elapsed.
         */
        OS_EVENT_SIGNAL_FROM_ISR(siganl_i2c_eeprom);
}

/*
 * 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]
         * disables hardware debouncing mechanism. 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 to configure pins after a power-up/wake-up cycle. Please note that every time the system enters sleep, it loses all its pin configurations.

```
/* I2C pin configuration */
static const gpio_config gpio_cfg[] = {

      // The system is set to [Master], so it outputs the clock signal
      HW_GPIO_PINCONFIG(HW_GPIO_PORT_4, HW_GPIO_PIN_6, OUTPUT,
                                                        I2C_SCL, true),

      // Bidirectional signal both for sending and receiving data
      HW_GPIO_PINCONFIG(HW_GPIO_PORT_4, HW_GPIO_PIN_7, INPUT,
                                                       I2C_SDA, true),
       // This is critical for the correct termination of the structure
      HW_GPIO_PINCONFIG_END

};

/**
 * @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);

      /* This is a shortcut to configure multiple GPIOs in one call */
      hw_gpio_configure(gpio_cfg);
}

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

```
{

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

5.5 EEPROM ACK Code

Code snippet of the EEPROM polling ACK routine. In **main.c**, add the following code (after system_init()):

```
/*
 * Function for polling the EEPROM status. This function should be invoked
 * after a successful I2C write operation to check when the EEEPROM device
 * is ready to accept the next write/read cycle.
 */
static void poll_ack(i2c_device dev)
{
       bool no_ack;

       /* Get hardware ID */
       HW_I2C_ID id = ad_i2c_get_hw_i2c_id(dev);

       ad_i2c_device_acquire(dev);
       ad_i2c_bus_acquire(dev);

       do {

              /*
               * Make sure TX ABORT interrupt status flag is reset.
               * At the beginning, EEPROM will not ACK the address
               * byte (the first byte sent by the master controller).
               * So, a TX ABORT will be issued in I2C controller.
               */
              hw_i2c_reset_int_tx_abort(id);

              /*
               * Clear STOP interrupt status flag that is used
               * for waiting ACK or NO ACK.
               */
              hw_i2c_reset_int_stop_detected(id);

              /*
               * We only need to check if EEPROM returns ACK for address byte, however we
               * cannot simply send address byte since it's the controller who takes care
               * of this once TX FIFO is filled with data. A simple solution is to send a
               * dummy byte which  will be ignored by EEPROM but will make the I2C
               * controller to generate a START condition and send the address byte.
               */
              hw_i2c_write_byte(id, 0xAA);
```

```
            /*
             * Wait until a STOP condition is detected and
             * the corresponding status bit is asserted.
             */
            while ((hw_i2c_get_raw_int_state(id) & HW_I2C_INT_STOP_DETECTED) == 0);

            /*
             * Check whether transmission was successful. If EEPROM did not
             * ACK the address byte, the appropriate abort source will be set.
             */
            no_ack = (hw_i2c_get_abort_source(id) &
                                        HW_I2C_ABORT_7B_ADDR_NO_ACK);


        } while (no_ack);

        ad_i2c_bus_release(dev);
        ad_i2c_device_release(dev);
}
```

5.6 Task Code for 24LC256

Code snippet of the **prvI2CTask_EEPROM** task responsible for interacting with the 24LC256 EEPROM module, externally connected on I2C1 bus. In **main.c**, add the following code (after system_init()):

```
/*
 * Page size of the selected EEPROM module. This is device-specific information!
 * I2C operations must be restricted within a page size. For example,
 * 0x0000 - 0x0040, 0x0041 to 0x0080, 0x0081 to 0x00C0 etc.
 */
#define PAGE_SIZE  64

/*
 * Error code returned after an I2C operation. It
 * can be used to identify the reason of a failure.
 */
HW_I2C_ABORT_SOURCE error_code;


#if I2C_ASYNC_EN == 1
/*
 * Callback function for the I2C asynchronous transactions:
 *
 * \param [in] error   Error code returned at the end of an I2C transaction.
 * \param [in] user_data  User data that can be passed and used within the function.
 */
void i2c_eeprom_cb(void *user_data, HW_I2C_ABORT_SOURCE error)
{
        /* Read the error status code */
        error_code = error;

```

```
        /*
         * Signal the [prvI2CTask_EEPROM] task that time
         * for resuming has elapsed.
         */
        OS_EVENT_SIGNAL_FROM_ISR(signal_i2c_eeprom_async);
}
#endif


/* Task responsible for controlling the 24LC256 module */
static void prvI2CTask_EEPROM ( void *pvParamters )
{
        i2c_device i2c_dev;

        /* Starting address (2 bytes) */
        uint8_t starting_addr[2] = {0x00, 0x00}; // or {0x00, 0x41} or {0x00, 0x81}

        /*
         * Data to be transferred: 64 bytes raw data + 2 bytes for the starting address
         */
        uint8_t eeprom_wd[PAGE_SIZE + 2];

        /* Buffer for storing the received data */
        uint8_t eeprom_rd[PAGE_SIZE];

       /*
        * Half of data (32 bytes) is set to [0x55] and half to [0xAA].
        * Here you can declare your own preferred values.
        */
        memcpy(eeprom_wd,   starting_addr, sizeof(starting_addr));
        memset(eeprom_wd + sizeof(starting_addr),    0x55,   (PAGE_SIZE/2));
        memset(eeprom_wd + sizeof(starting_addr) + (PAGE_SIZE/2), 0xAA,
                                                                  (PAGE_SIZE/2));


        /*
         * I2C adapter initialization should be done once at the beginning. Alternatively,
         * this function could be called during system initialization in system_init().
         */
        ad_i2c_init();


        for (;;) {

                /*
                 * Suspend task execution - As soon as WKUP callback function
                 * is triggered, the task resumes its execution.
                 */
                OS_EVENT_WAIT(siganl_i2c_eeprom, OS_EVENT_FOREVER);

                /*
                 * Turn on LED D2 on ProDev Kit indicating the start of a process.
                 */
                hw_gpio_set_active(HW_GPIO_PORT_1, HW_GPIO_PIN_5);
```

```
        /*
         * Open the device that will access the I2C bus.
         */
        i2c_dev =  ad_i2c_open(MEM_24LC256);


        /*
         * Write a whole page (64 bytes) in EEPROM.
         */
        if (i2c_status == 1) {

                printf("\n\rI2C write...\n\r");

                /*
                 * Write some data in EEPROM, starting from physical address 0x0000
                 */
                error_code = ad_i2c_write(i2c_dev, eeprom_wd, sizeof(eeprom_wd));


                /* Check the status of the I2C write operation */
                if (error_code == 0) {
                        printf("\n\rSuccessful write operation!\n\r\n\r");

                    /*
                     * Wait until data is actually written in EEPROM cells!
                     */
                        poll_ack(i2c_dev);
                } else {
                        printf("\n\rError in write operation with error code: %d!\n\r\n\r",
                                                                        error_code);
                }

        /*
         * Read a whole page (64 bytes) from EEPROM
         */
        } else {

                printf("\n\rI2C read...\n\r");
/*
 * Perform the read operation synchronously!
 */
#if I2C_ASYNC_EN == 0
                /*
                 * This function performs a write followed by a read transaction. An
                 * operation which is typical when reading data from I2C peripherals,
                 * where an address needs to be specified through a write before
                 * reading data.
                 */
                error_code = ad_i2c_transact(i2c_dev, starting_addr,
                                sizeof(starting_addr), eeprom_rd, sizeof(eeprom_rd));

/*
 * Perform the read operation asynchronously!
 */
```

```
#else
		/* Perform an I2C read operation asynchronously */
		ad_i2c_async_transact(i2c_dev,
				I2C_SND(starting_addr, sizeof(starting_addr)),
				I2C_RCV(eeprom_rd, sizeof(eeprom_rd)),
				I2C_CB(i2c_eeprom_cb),
				I2C_END
			);

		/* Wait until the current I2C operation is finished */
		OS_EVENT_WAIT(signal_i2c_eeprom_async, OS_EVENT_FOREVER);
#endif

		/* Check the status of the I2C read operation */
		if (error_code == 0) {
			printf("\n\rSuccessful read transaction!\n\r");

			/*
			 * Check if read data match written data. If there is a match
			 * [strncmp] returns [0]
			 */
			if (!strncmp(((char *)eeprom_wd + 2), (char *)eeprom_rd,
								PAGE_SIZE)) {
				printf("\n\rRead data match written data!\n\r\n\r");
			} else {
				printf("\n\rRead data do not match written data!\n\r\n\r");
			}

		} else {
			printf("\n\rUnsuccessful read transaction with error code:
							%d!\n\r\n\r",  error_code);
		}

	} // end of else()


	/* Close the already opened device */
	ad_i2c_close(i2c_dev);

	/*
	 * 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);

      } // end of for()
} // end of task
```

5.7 Macro Definitions

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

```
/*
```

```
 * Enable the preferred devices, declared in "platform_devices.h"
 */
#define CONFIG_24LC256

/*
 * Macros for enabling I2C operations using Adapters
 */
#define dg_configUSE_HW_I2C            (1)
#define dg_configI2C_ADAPTER           (1)
```

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
2.0	23-July-2018	More descriptive steps to follow, figures and examples.
2.1	20-Sep-2018	Updated figures, Minor improvements in *prvI2CTask_EEPROM*.

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