Tutorial

SPI Adapters

For the DA1468x SoC

Abstract

This tutorial should be used as a reference guide to gain a deeper understanding of the 'SPI 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 SPI 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

CS	Chip Select
DAC	Digital-to-Analog Converter
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
MISO	Master Input Slave Output
MOSI	Master Output Slave Input
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 SPI peripheral Adapters
- Give a complete sample project demonstrating the usage of SPI peripheral Adapters

1.2 SPI Adapters Introduction

This tutorial explains SPI adapters and how to configure the DA1468x family of devices as an SPI Master device. Adopting an SPI Slave role is not used for the majority of situations and so is not covered in this tutorial. The SPI adapter is an intermediate layer between the SPI Low Level Drivers (LLDs) and a user application. It allows the user to utilize the SPI interface in a simpler way than when using APIs from LLDs. The key features of SPI 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 SPI bus after acquiring it.
- Placing code between ad_spi_bus_acquire() and ad_spi_bus_release() ensures that only one task can use the SPI bus to communicate with an external connected device. During this period no other device or task can use the SPI interface until the ad_spi_bus_release() function is called by the owning task.
- The Power Manager (PM) of the chip is aware of the SPI peripheral usage and, before the system enters sleep, it checks whether or not there is activity on the SPI 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.

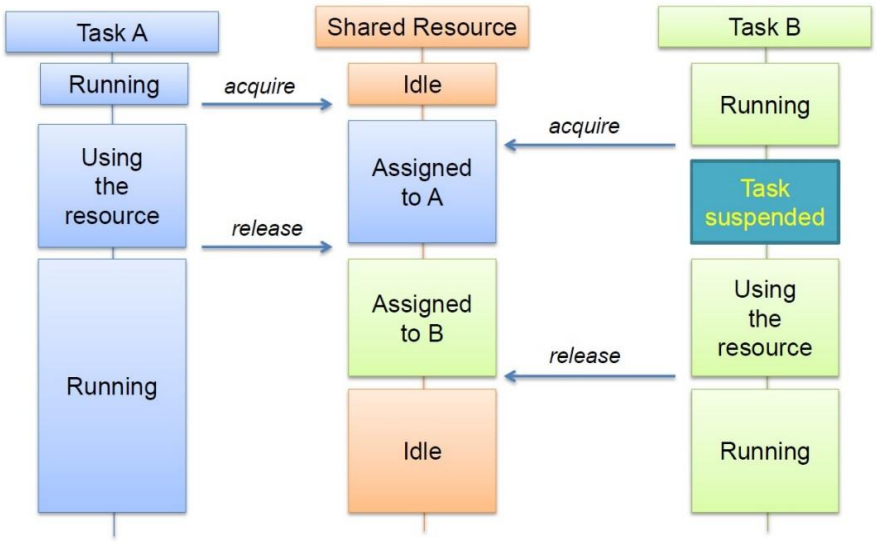

Figure 1: Adapters Communication

2 SPI Adapters Concept

This section describes the key features of SPI peripheral adapters as well as the procedure to enable and correctly configure the peripheral adapters for the SPI 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) and GPADC adapters.

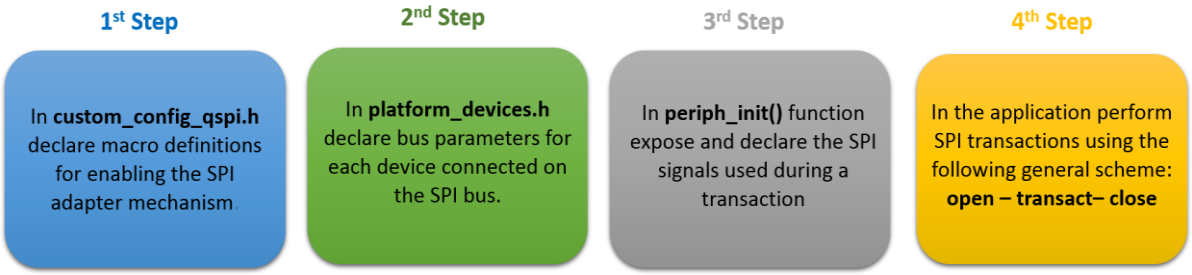

Figure 2: The Four-Step Process for Setting an Adapter Mechanism

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 peripheral hardware blocks. In particular, this tutorial focuses on the adapters that are responsible for the SPI peripheral hardware block. Table 1 briefly explains the header files related to SPI adapters (red indicates the path under which the files are stored while green indicates which ones are used for SPI operations).

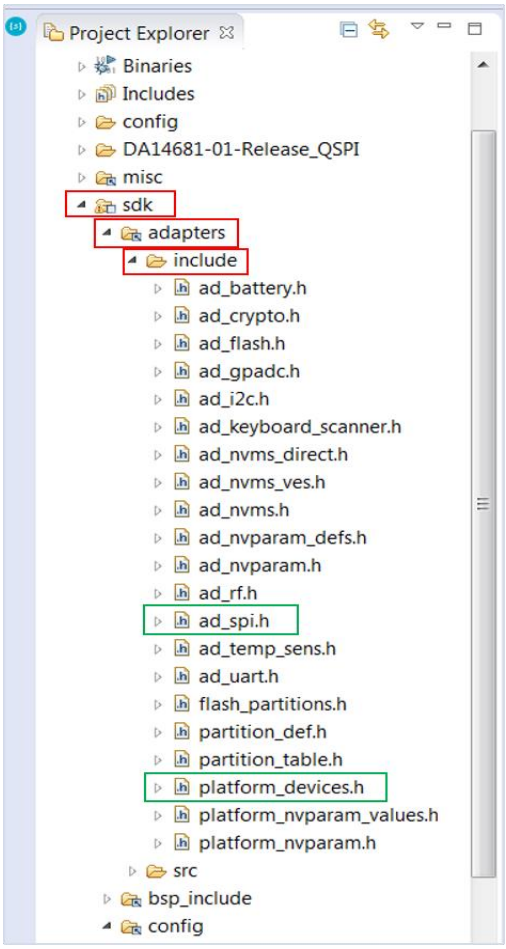

Figure 3: Headers for SPI Adapters

Table 1: Header Files used by SPI Adapters

Filename	Description
ad_spi.h	This file contains the recommended APIs and macros for performing SPI related operations. Use these APIs when accessing the SPI peripheral bus.
platform_devices.h	This file contains all the device configurations. 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 an SPI Operation

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

```
/*
 * Macros for enabling SPI operations using Adapters
 */
#define dg_configUSE_HW_SPI                     (1)
#define dg_configSPI_ADAPTER                    (1)
```

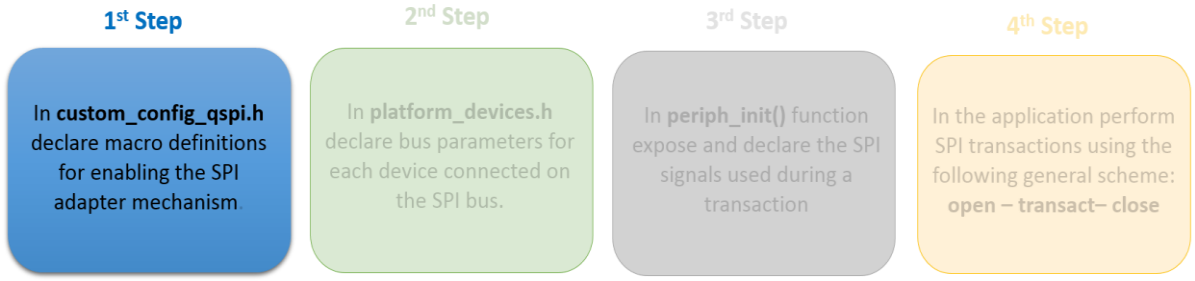

Figure 4: First Step for Configuring the SPI Adapter Mechanism.

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 SPI bus. A device can be considered a set of settings describing the complete SPI interface. These settings are applied every time the device is selected and used. To do this, the SDK uses a macro, named **SPI_SLAVE_DEVICE**:

```
/*
 * Macro for settings SPI bus parameters
 */

SPI_SLAVE_DEVICE(bus, name, cs_port, cs_pin, _word_mode, pol_mode,

                                    _phase_mode,  xtal_div, dma_channel)
```

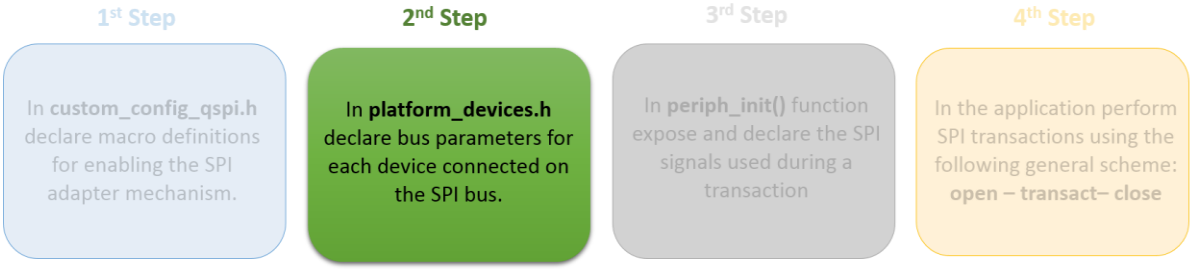

Figure 5: Second Step for Configuring the SPI Adapter Mechanism

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

Argument Name	Description
bus	The DA1468x family of devices features two distinct SPI hardware blocks. Valid values are SPI1 and SPI2.
name	Declare an arbitrary alias for the SPI interface (for instance, My_slave_device). This name should be used for opening that specific device.
cs_port	Slave Chip Select line: this can be any available port on the DA1468x chip.

cs_pin	Slave Chip Select line: this can be any available pin on the DA1468x chip.
_word_mode	The size of each data packet sent over the bus. Valid values are those from **HW_SPI_WORD** enum in /sdk/peripherals/include/hw_spi.h.
pol_mode	Clock polarity. Valid values are those from **HW_SPI_POL** enum in /sdk/peripherals/include/hw_spi.h.
_phase_mode	Clock phase. Valid values are those from **HW_SPI_PHA** enum in /sdk/peripheral/include/hw_spi.h.
xtal_div	SPI interface speed. Valid values are those from **HW_SPI_FREQ** enum in /sdk/peripherals/include/hw_spi.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: In contrast with the I2C adapters, SPI adapters share the same macro whether or not a DMA channel is used for a transaction. Therefore, if a DMA channel is not needed a value equal to -1 should be declared. Also note that DMA RX/TX channels must be set in pairs, that is, 0/1, 2/3, 4/5 and 6/7. Thus, RX channel must always be set to an even number (0, 2, 4, 6).

The DA1468x family of devices incorporates two distinct SPI blocks namely SPI1 and SPI2. Depending on the SPI interface used, device declarations must be placed between the correct macro indicators in platform_devices.h:

```
/* Declare SPI bus configurations for devices connected to SPI1 hardware block */
SPI_BUS(SPI1)

// Use SPI_SLAVE_DEVICE()for each device declaration

SPI_BUS_END


/* Declare SPI bus configurations for devices connected to SPI2 hardware block */
SPI_BUS(SPI2)

// Use SPI_SLAVE_DEVICE() for each device declaration

SPI_BUS_END
```

3. As illustrated in Figure 6, the third step is the declaration of the SPI signals. The user can multiplex and expose SPI signals on any available pin on DA1468x SoC.

```
static void prvSetupHardware( void )
{

   /* Init hardware */
   pm_system_init(periph_init)

}
```

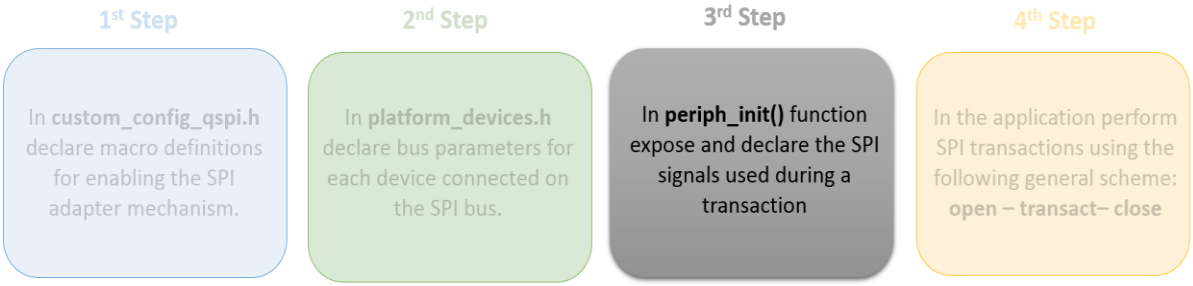

Figure 6: Third Step for Configuring the SPI Adapter Mechanism

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. Having enabled the SPI adapter mechanism, the developer is able to use all the available APIs for performing SPI transactions. The following describes the required sequence of APIs in an application to successfully execute an SPI write/read operation.

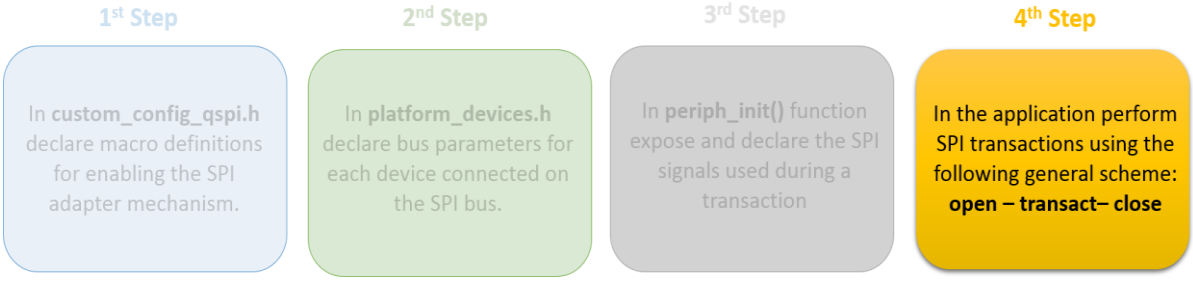

Figure 7: Fourth Step for Configuring the SPI Adapter Mechanism.

a. ad_spi_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_spi_open()

   Before using the SPI interface, the application task must open the device that will access the bus. Opening a device, involves enabling the SPI controller. If the device is the only connected device on the SPI bus, configuration of the SPI controller also takes place. The 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_spi_bus_acquire()

   This API is optional since it is automatically called upon a write/read transaction and is used for locking the SPI bus for the given opened device. This function should be called when the application task wants to communicate to the SPI 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_spi_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 SPI transaction either synchronously or asynchronously. Please note that all the available APIs for writing/reading over the SPI bus, nest the corresponding APIs for acquiring and releasing a device.

e. ad_spi_bus_release()

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

f. ad_spi_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 SPI bus. Remember that the SPI adapter implementation follows a single device scheme, that is only one device can be opened at a time.

2.3 SPI 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. The mechanism initially waits for the SPI bus to become available and then blocks the calling task until a transaction is completed. Once a write/read process is finished, the SPI bus is freed and further write/read transactions over the SPI bus can take place.

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

```
// Open the device that will utilize the SPI bus
spi_device dev = ad_spi_open(My_Slave_Device);

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

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

The above code performs a write transaction followed by a read transfer. First, the chip select line is activated for the slave device and then data is sent over the SPI bus. When the current transaction is finished, the device changes to read mode and reads data from the connected device. Finally, the chip select line is deactivated when the transaction has finished.

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_spi_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. SPI adapters allow developer to perform SPI transactions that consist of a number of reads, writes, and callback calls. This provides a time-efficient way to manage all SPI 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 SPI transaction schemes. Table 3 explains all the available arguments that can be used to declare an SPI transaction scheme.

Table 3: Available Arguments for Configuring SPI Asynchronous Transactions

Argument Name	Description
SPI_CSA	Use this argument to select the Chip Select line.
SPI_CSD	Use this argument to deselect the Chip Select line.
SPI_SND()	Use this argument to send data over SPI bus.
SPI_RCV()	Use this argument to read data over SPI bus.
SPI_SRCV()	This argument is a combination of a write followed by a read SPI transaction.
SPI_CB0()	Declare a callback function that should be called when finishing with all defined SPI actions. Developer **cannot** pass any data in the callback function.
SPI_CB1()	Declare a callback function that should be called when finishing with all defined SPI actions. Developer **can** pass data in the callback function.
SPI_END	Use this argument to mark the end of an SPI transaction scheme. This argument should be the last declared action.

Code Snippet of a typical write followed by a read asynchronous SPI transaction:

```
// Open the device that will utilize the SPI bus
spi_device dev = ad_spi_open(My_Slave_Device);

// Perform SPI transactions to the already opened device
ad_spi_async_transact(dev, SPI_CSA,             // Activate the  Chip Select line
                      SPI_SND(command,  sizeof(command) ),  // SPI write operation
                      SPI_RCV(response, sizeof(response)),  // SPI read  operation
                      SPI_CB0(final_callback) , // User-defined callback function
                      SPI_CSD,                  // Deactivate the Chip Select line
                      SPI_END);                 // Indicate the end of SPI operations

// Make sure that the current transaction completes

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

When performing SPI operations in asynchronous mode, the following should be considered:

- Callback functions are executed from within Interrupt Service Routine (ISR) context. Therefore, a 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 (SPI 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.

2.4 Duplex Transmissions

The SPI hardware block supports a **Duplex** mode separately from write or read only mode. In Duplex mode, at the same time as data is shifted out from the Master Output pin (MOSI), data is shifted in from the Master Input pin (MISO). This feature can be useful when loopback tests are needed to validate the functionality of the SPI interface or when an SPI transaction between devices needs to do so. SDK provides an API along with a data structure that should be used for duplex SPI transactions.

Table 4: APIs for Duplex Transactions

Name	Description
ad_spi_complex_transact()	Use this function to perform duplex transactions on an SPI bus. Please note that buffers for both written and read data should be provided. This function can also be used for complex transactions, that is, it can perform several transactions in one Chip Select activation. The total number of transactions is determined by the last input parameter. For instance, when two identical transactions are needed, set this parameter to '2'.
spi_transfer_data	Use this structure when performing either a complex or duplex SPI transaction.

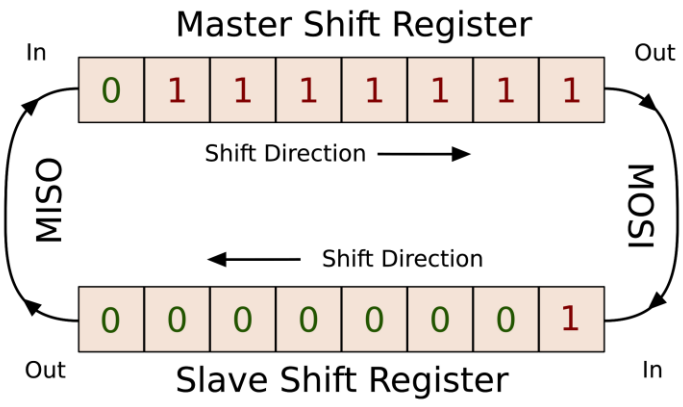

Figure 8: Duplex SPI Transaction

2.5 SPI Related Macros

SPI 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_spi.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 5.

Table 5: SPI Macros

Macro Name	Description
CONFIG_SPI_EXCLUSIVE_OPEN	Set this macro to '1' to prevent multiple tasks from opening the same device. When set to '1' both ad_spi_device_acquire() and ad_spi_device_release() are no longer necessary.
CONFIG_SPI_ONE_DEVICE_ON_BUS	Set this macro to '1' if only one device is connected on the SPI bus (one on SPI1 and one on SPI2). This will reduce code size and improve performance.

3 Analyzing The Demonstration Example

This section analyzes an application example which demonstrates using the SPI 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 SPI operations. One task performs loopback tests using the SPI interface in duplex mode, while the second task controls an external SPI module, connected on SPI1 bus. The code also enables the wake-up timer for handling external events. Both synchronous and asynchronous SPI operations are demonstrated.

3.1 Application Structure

1. The key goal of this demonstration is for the device to perform a few SPI 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.

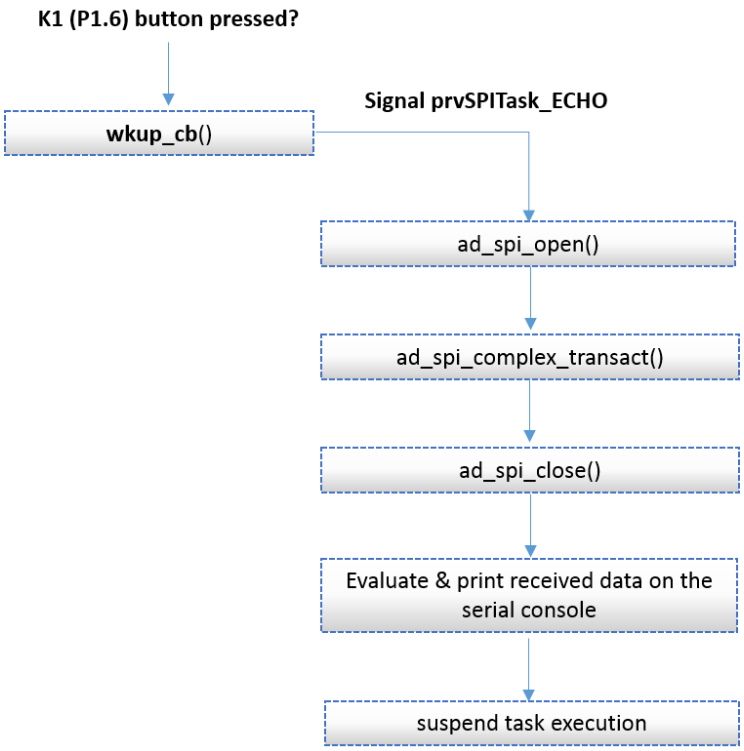

Figure 9: Echo Task – Main Execution Path

In this function, the task responsible for the SPI loopback tests is signaled so that it can unblock. In this freeRTOS task, a duplex transaction takes place, that is, the data shifted out from the MOSI pin is shifted in from the MISO pin at the same time. Upon finishing the transaction, a data integrity check is performed and the corresponding debugging message is printed out on the serial console.

2. At the same time, at every 100 ms time interval, the task which is responsible for controlling the MCP4822 DAC module is executed. Depending on the value of the SPI_ASYNC_EN macro, a synchronous or an asynchronous SPI write operation is performed. A 2-byte data is sent over the SPI bus to update the analog output value of the MCP4822 DAC module. A variable named data holds both the control bits as well as the raw data. At the end of every SPI transaction, the raw data increments thus increasing the output analog value of the DAC module.

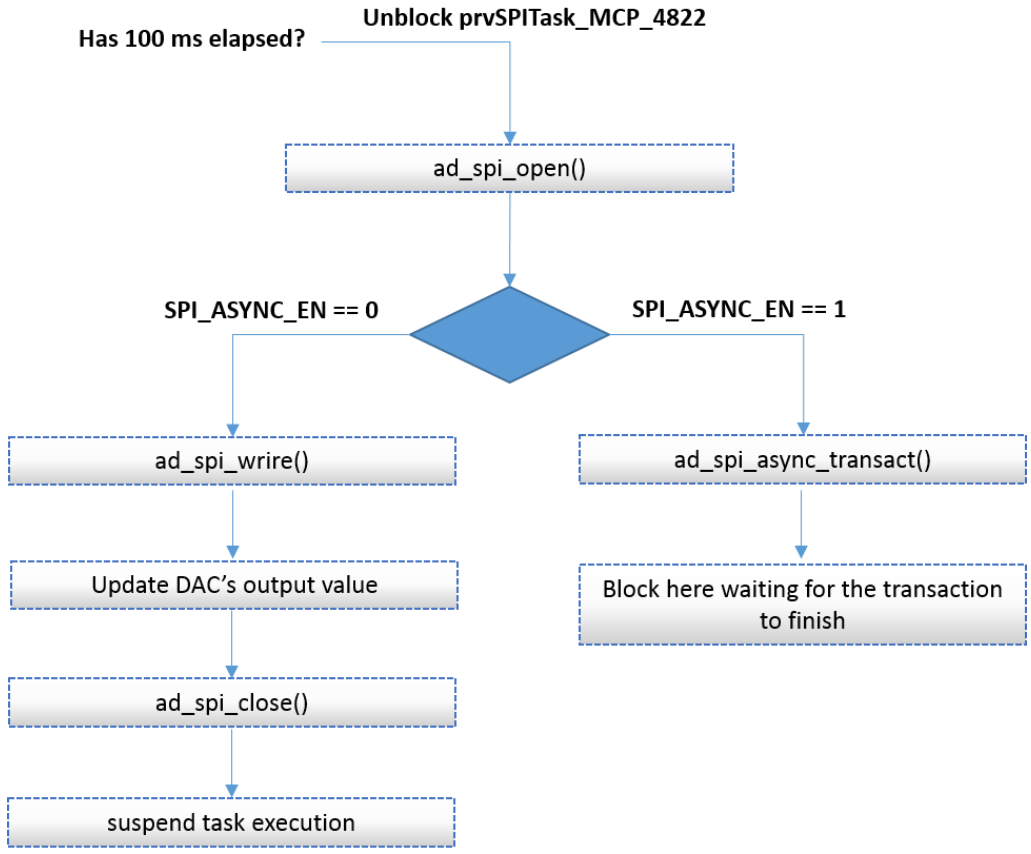

Figure 10: MCP4822 Write SW FSM – Main Execution Path

3. As mentioned, the SPI_ASYNC_EN macro can be used to enable asynchronous SPI write operations. As described in SPI Transactions, developers must not call asynchronous related APIs without guaranteeing that the previous asynchronous transaction is finished. To ensure this, after calling the ad_spi_async_transact() function, the code waits for the arrival of a signal, indicating the end of the current SPI operation.

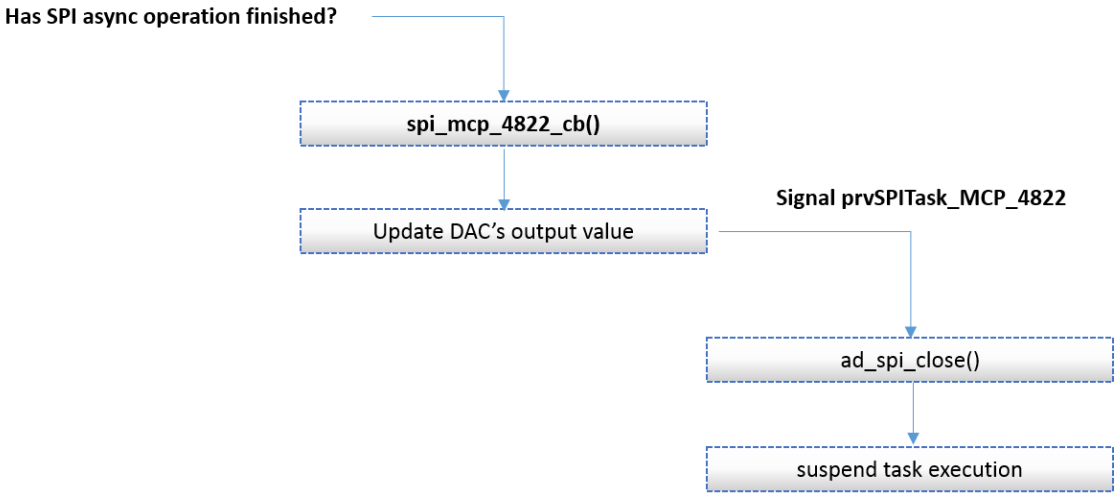

Figure 11: MCP4822 Async Write SW FSM – Callback Function Execution Path

4. The selected DAC module consists of a 2-byte register, in which the first four significant bits (MSbits) control the behavior of the module. Since the demonstrated example masks all four configuration bits with 0xF000, it is expected that the module is in active mode and VoutB is activated. For more information on the control bits, read the Serial Interface section in the manufacturer datasheet.

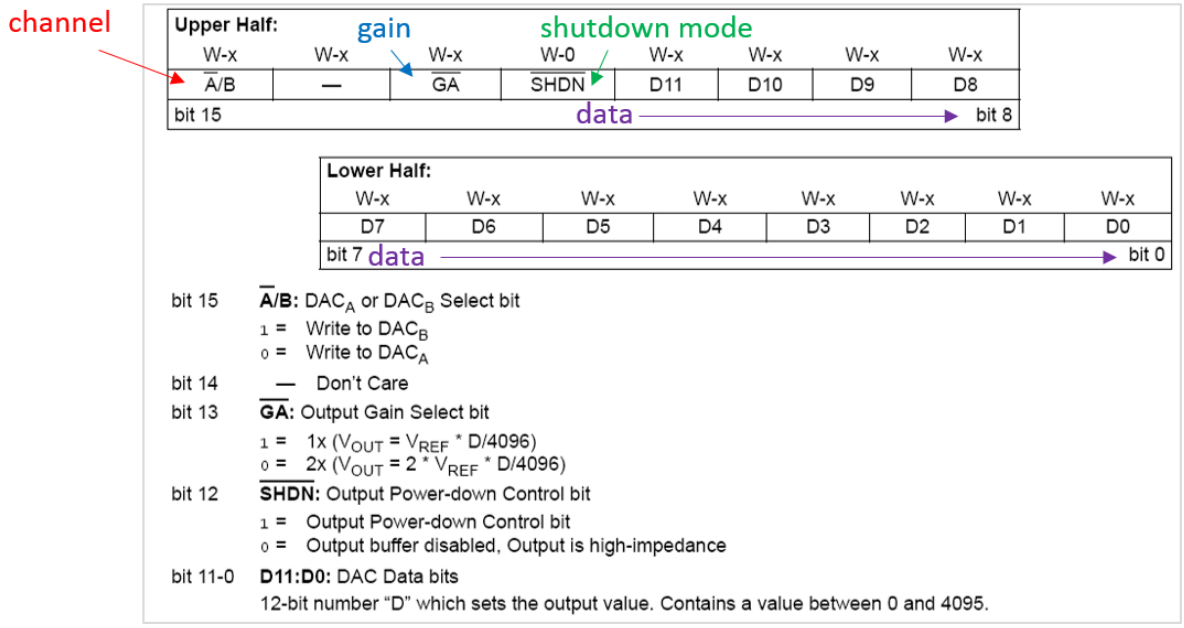

Figure 12: Control and Data bits of MCP4822 DAC Chip

Note: The MCP4822 DAC 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, an MPC4822 DAC module, a digital multimeter, and optionally a logic analyzer are required for testing and verifying the code. In addition, a breadboard and a few jumper wires are required to connect the SPI module to the Pro DevKit. 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.

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.

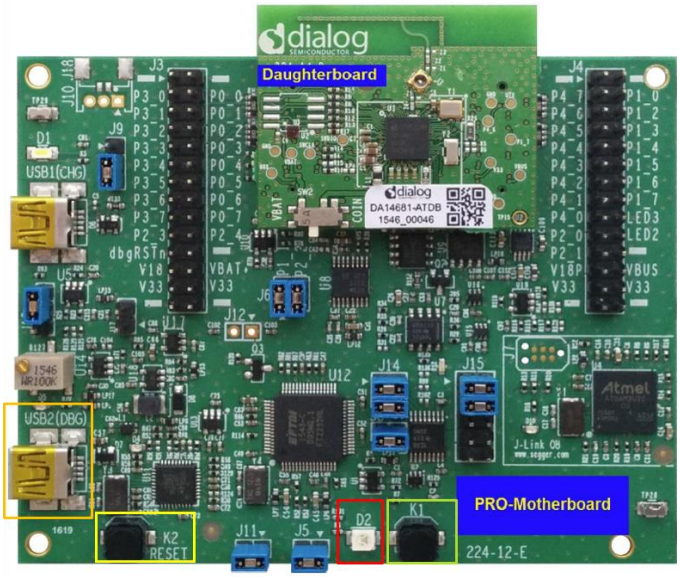

Figure 13: DA1468x Pro DevKit

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).

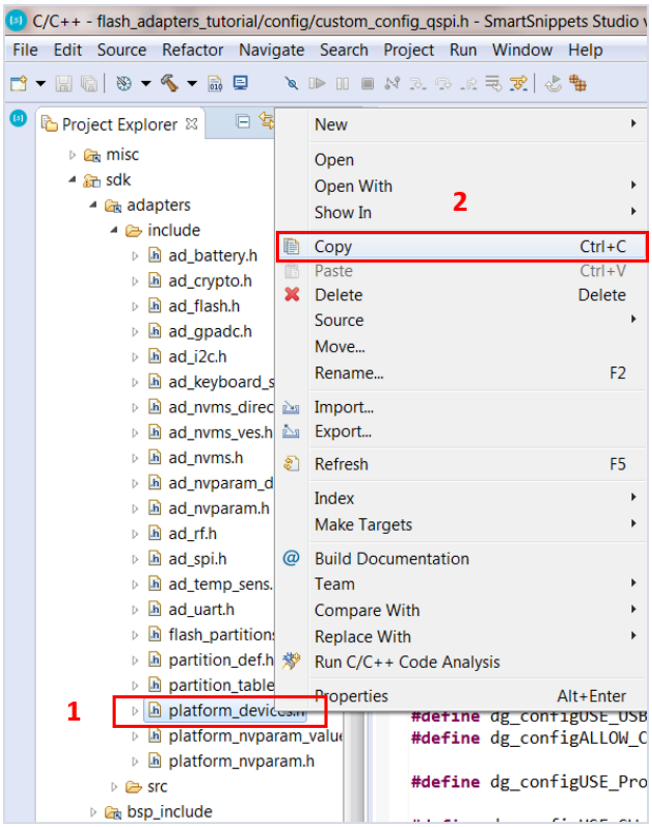

Figure 14: Creating platform_devices.h Header File, Step 1

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

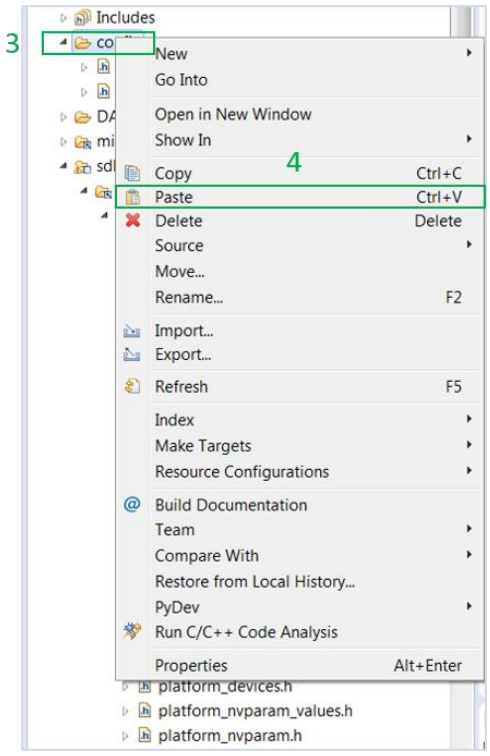

Figure 15: Creating platform_devices.h Header File, Step 2

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. 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. Connect the DAC module to the Pro DevKit. Figure 15 illustrates the pin connections required to configure the MCP4822 module. For more information on the DAC module used, read the manufacturer datasheet.

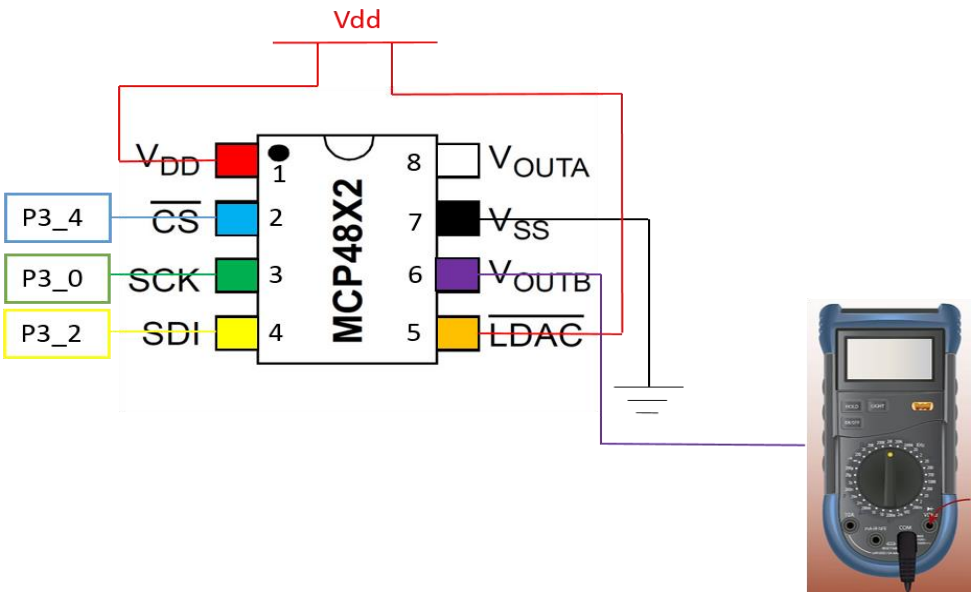

Figure 16: Configuring the MCP4822 DAC Slave Device

Note: A shortcut between MOSI (P3_2) and MISO (P3_1) pins on the Pro DevKit must be done for testing the chip in duplex mode.

7. Connect a digital multimeter to **VoutB** pin of the DAC module.
8. Press the **K2** button on Pro DevKit. This step starts the chip executing its firmware.
9. Open a serial terminal (115200, 8-N-1) and press the **K1** button on Pro DevKit. A debugging message is displayed on the console indicating whether or not the duplex SPI transaction has been successfully executed.

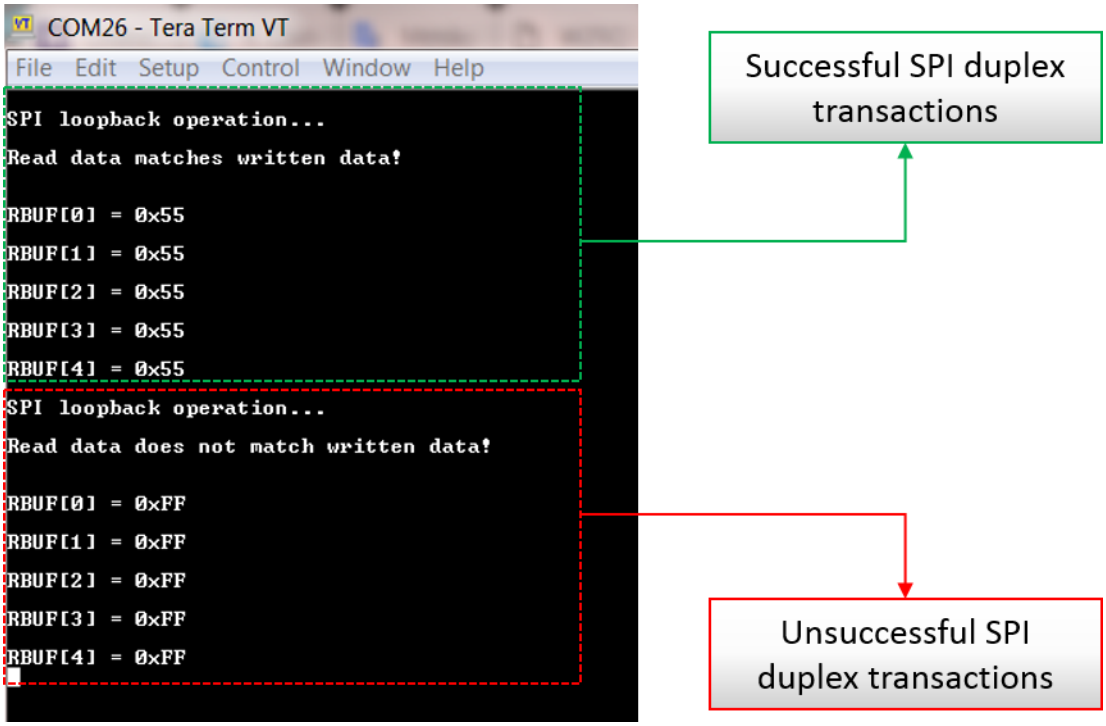

Figure 17: Debugging Messages Indicating the Status of the Duplex SPI Transaction

10. Verify that the analog output value of the DAC module increments until it reaches a value close to 2.5 V. The read value is then wrapped around starting from 0 V. Note that the internal Vref signal of the module is set to 2.5 V.

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 SPI 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 **P3_0** pin of Pro DevKit. This is the Clock signal (SCK).
    b. Connect a channel from the logic analyzer to **P3_2** pin of Pro DevKit. This is the Master Output Slave Input signal (MOSI).
    c. Connect a channel from the logic analyzer to **P3_4** pin of Pro DevKit. This is the Chip Select signal (CS).
3. Press the **K1** button on Pro DevKit and capture the SPI duplex transaction (loopback test).

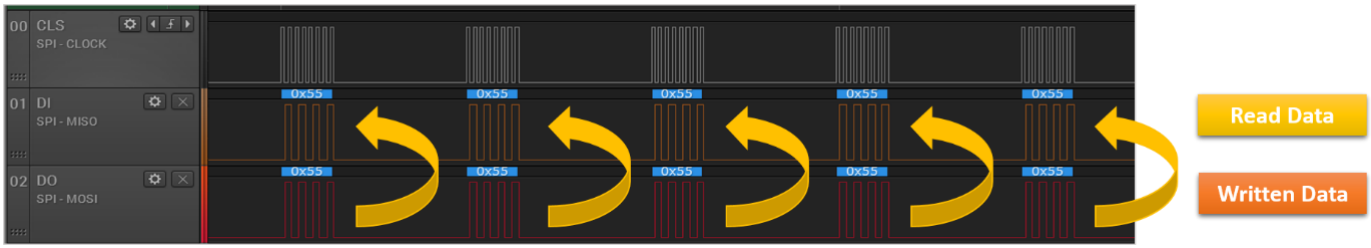

Figure 18: SPI Duplex Transaction Captured using a Logic Analyzer.

4. At any time, capture an SPI write transaction between the system and the MCP4822.

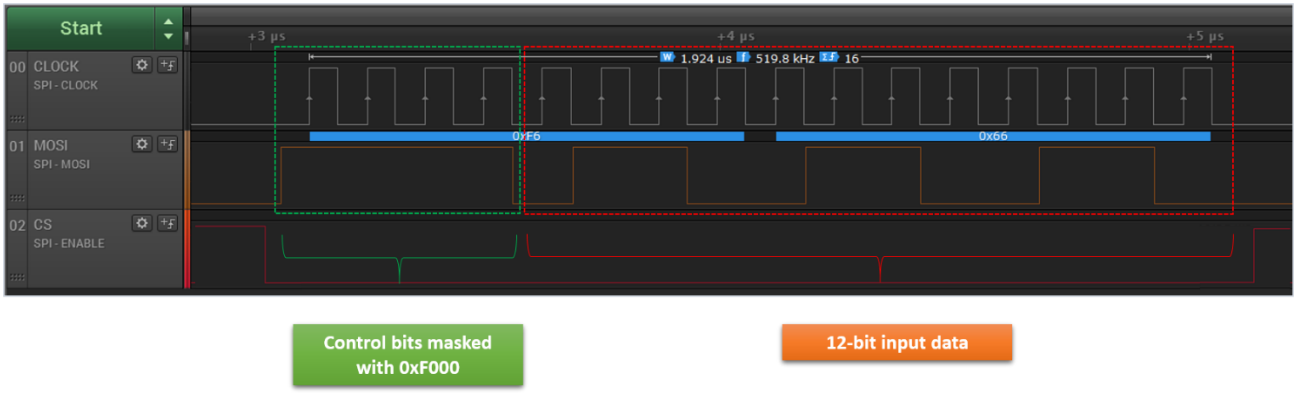

Figure 19: SPI Write Transaction Captured using a Logic Analyzer.

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 "ad_spi.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 SPI transactions.
 *
 * Valid values:
 * 0: SPI transactions will follow a synchronous scheme
 * 1: SPI transactions will follow an asynchronous scheme
```

```
 *
 */
#define SPI_ASYNC_EN     (1)

/* OS signals used for synchronizing OS tasks*/
static OS_EVENT signal_echo;
static OS_EVENT signal_mcp_4822;

/* SPI Task priority */
#define mainSPI_TASK_PRIORITY       ( OS_TASK_PRIORITY_NORMAL )

/*
 * SPI application tasks - Function prototype
 */
static void prvSPITask_ECHO( void *pvParameters );
static void prvSPITask_MCP_4822( void *pvParameter );

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

      /* Handler for SPI freeRTOS tasks */
      OS_TASK echo_h    = NULL;
      OS_TASK mcp4822_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_echo);
```

```
    OS_EVENT_CREATE(signal_mcp_4822);

    /* 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 SPI task responsible for SPI loopback tests
     */
    OS_TASK_CREATE( "SPI_ECHO",

            prvSPITask_ECHO,
            NULL,
            200 * OS_STACK_WORD_SIZE,

            mainSPI_TASK_PRIORITY,
            echo_h );
    OS_ASSERT(echo_h);

    /*
     * Create an SPI task responsible for controlling the
     * externally connected MCP4822 module.
     */
    OS_TASK_CREATE( "SPI_MCP_4822",

            prvSPITask_MCP_4822,
            NULL,
            200 * OS_STACK_WORD_SIZE,

            mainSPI_TASK_PRIORITY,
            mcp4822_h );
    OS_ASSERT(mcp4822_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 the [prvSPITask_ECHO] task that time for performing
         * a loopback test has elapsed.
         */
        OS_EVENT_SIGNAL_FROM_ISR(signal_echo);
}

/*
 * 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.

```
/* SPI pin configurations */
static const gpio_config gpio_cfg[] = {

    // The system is set to [Master], so it outputs the clock signal
    HW_GPIO_PINCONFIG(HW_GPIO_PORT_3, HW_GPIO_PIN_0,
                                          OUTPUT, SPI_CLK, true),


    // Pin for capturing data
    HW_GPIO_PINCONFIG(HW_GPIO_PORT_3, HW_GPIO_PIN_1,
                                            INPUT,  SPI_DI,  true),

    // Pin for outputting data
    HW_GPIO_PINCONFIG(HW_GPIO_PORT_3, HW_GPIO_PIN_2,
                                         OUTPUT, SPI_DO,  true),

    /*
     * CS pin used when performing the loopback tests. Since the system is set
     * to [Master], it drives this line.
     */
    HW_GPIO_PINCONFIG(HW_GPIO_PORT_3, HW_GPIO_PIN_3,
                                          OUTPUT, GPIO,    true),

    /*
     * CS pin used when performing transactions with the MCP4822 module, externally
     * connected on the SPI1 bus. Since the system is set to [Master], it drives this line.
     */
    HW_GPIO_PINCONFIG(HW_GPIO_PORT_3, HW_GPIO_PIN_4,
                                         OUTPUT, GPIO,    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);


      /* Configure the SPI pins */
      hw_gpio_configure(gpio_cfg);
}


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

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

5.5 Task Code for MCP4822

Code snippet of the **prvSPITask_MCP_4822** task responsible for interacting with the MCP4822 DAC module, externally connected on SPI1 bus. In **main.c**, add the following code (after system_init()):

```
/*
 * Data structure for the DAC4822 module
 */
static struct spi_mcp_4822 {

      // DAC4822 consists of a 2-byte register
      uint16_t data;
```

```
        // Control bits mask
        uint16_t mask;

        // Device handle
        spi_device spi_dev;

} spi_mcp_4822_t = {  // Create an instance of the above structure

        .data = 0x0000,

        // Control bits occupy the 4 MSbits and are set to '1'
        .mask = 0xF000,

        .spi_dev = NULL,
};

#if SPI_ASYNC_EN == 1

/*
 * Callback function called upon an SPI asynchronous transaction.
 *
 * \param[in] user_data  User data that can be passed and used within the function
 */
void spi_mcp_4822_cb( void *user_data )
{
        /*
         * Just to show how parameters can be passed
         * within callback functions!
         */
        struct spi_mcp_4822 *spi_param = (struct spi_mcp_4822 *)user_data;

        /*
         * Increment the analog output value of the DAC module. When data reaches
         * its maximum value, it wraps around starting from zero value.
         */
        spi_param->data += 10;


        /* Signal [prvSPITask_MCP_4822] that time for resuming has elapsed */
        OS_EVENT_SIGNAL_FROM_ISR(signal_mcp_4822);
}
#endif


/* Task responsible for controlling the MCP4822 module */
static void prvSPITask_MCP_4822(  void *pvParameters )
{
        /*
         * SPI adapter initialization should be done once at the beginning. Alternatively,
         * this function could be called during system initialization in system_init().
         */
        ad_spi_init();
```

```
        for (;;) {

                /* Suspend task's execution for 100 ms */
                OS_DELAY(OS_MS_2_TICKS(100));


                /*
                 * Set DAC's control bits using a mask.
                 */
                spi_mcp_4822_t.data |= spi_mcp_4822_t.mask;


                /*
                 * 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 SPI bus
                 */
                spi_mcp_4822_t.spi_dev = ad_spi_open(MCP_4822);

#if SPI_ASYNC_EN == 0
                /*
                 * Perform a synchronous SPI write operation, that is, the task is blocking
                 * waiting for the transaction to finish. Upon transaction completion,
                 * the blocked task unblocks and resumes its operation.
                 */
                ad_spi_write(spi_mcp_4822_t.spi_dev, (uint8_t *)&spi_mcp_4822_t.data,
                                                                        sizeof(uint16_t));

                /*
                 * Increment the analog output value of the DAC module. When data reaches
                 * its maximum value, it wraps around starting from zero value.
                 */
                spi_mcp_4822_t.data += 10;

#else
                /*
                 * Perform an asynchronous SPI write operation, that is, the task does not
                 * block waiting for the transaction to finish. Upon transaction completion
                 * callback function is triggered indicating the completion of the SPI operation
                 */
                ad_spi_async_transact(spi_mcp_4822_t.spi_dev,
                        SPI_CSA, // Activate CS signal

                        /* Send some data... */
                        SPI_SND((uint8_t *)&spi_mcp_4822_t.data, sizeof(uint16_t)),

                        /* Declare callback function and data to be passed inside it. */
                        SPI_CB1(spi_mcp_4822_cb, &spi_mcp_4822_t),

                        SPI_CSD, // Deactivate CS signal
                        SPI_END
```

```
                );

            /*
             * In the meantime and while SPI transactions are performed in the
             * background, application task can proceed to other operations/calculation.
             * It is essential that, the new operations do not involve SPI transactions
             * on the already occupied bus!!!
             */

            /*
             * Make sure that the current SPI operation has finished,
             * blocking here forever.
             */
            OS_EVENT_WAIT(signal_mcp_4822, OS_EVENT_FOREVER);
#endif

            /* Close the already opened device */
            ad_spi_close(spi_mcp_4822_t.spi_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
```

5.6 Task Code for Loopback Test

Code snippet of the **prvSPITask_ECHO** task responsible for performing loopback tests. In **main.c**, add the following code (after prvSPITask_MCP_4822()):

```
/* Number of transmitted/received data */
#define BUFFER_SIZE 5

/*
 * Create an SPI task responsible for SPI loopback tests
 */
static void prvSPITask_ECHO( void *pvParamters )
{
    /* Buffer for storing the transmitted data */
    uint8_t wbuf[BUFFER_SIZE];

    /*  Buffer for storing the received data */
    uint8_t rbuf[BUFFER_SIZE];


    /* Configure the transfer scheme used during the SPI duplex transaction */
    spi_transfer_data spi_transfer_cfg = {
        .wbuf = (void *)wbuf,
        .rbuf = (void *)rbuf,
```

```
        .length = BUFFER_SIZE,
};

/* Initialize transmitted data - All data are set to 0x55 */
memset(wbuf, 0x55, BUFFER_SIZE);

/*
 * Initialization should be done once at the beginning. Alternatively,
 * this function could be called during system initialization in [system_init]
 * function.
 */
ad_spi_init();

spi_device spi_dev;

for (;;) {

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


        /* Open the device that will access the SPI bus */
        spi_dev = ad_spi_open(ECHO_LOOP);

        printf("\n\rSPI loopback operation...\n\r");

        /* Perform a duplex SPI transaction */
        ad_spi_complex_transact(spi_dev, &spi_transfer_cfg, 1);

        /* Close the already opened device */
        ad_spi_close(spi_dev);

        /*
         * Check whether the read data matches written data. If there is a match
         * [strncmp] returns [0].
         */
        if (!strncmp((char *)spi_transfer_cfg.rbuf, (char *)spi_transfer_cfg.wbuf,
                                                                BUFFER_SIZE)) {
                printf("\n\rRead data matches written data!\n\r\n\r");
        } else {
                printf("\n\rRead data does not match written data!\n\r\n\r");
        }

        /* Print out the received data */
        for (int i = 0; i < 5; i++) {
                printf("\n\rRBUF[%d] = 0x%X\n\r", i,
                                *( ((uint8_t *)spi_transfer_cfg.rbuf) + i) );
        }
        fflush(stdout);


} // 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 SPI_ECHO
#define SPI_MCP_4822

/*
 * Macros for enabling SPI operations using Adapters
 */
#define dg_configUSE_HW_SPI             (1)
#define dg_configSPI_ADAPTER            (1)
```

5.8 SPI Bus Configuration Macros

In the newly created platform_devices.h, add the following device configurations between SPI_BUS(SPI1) and SPI_BUS_END:

```
#ifdef SPI_ECHO
    SPI_SLAVE_DEVICE(SPI1, ECHO_LOOP, HW_GPIO_PORT_3, HW_GPIO_PIN_3,
                HW_SPI_WORD_8BIT, HW_SPI_POL_LOW, HW_SPI_PHA_MODE_0,
                    HW_SPI_FREQ_DIV_2, -1);
#endif

#ifdef SPI_MCP_4822
    SPI_SLAVE_DEVICE(SPI1, MCP_4822, HW_GPIO_PORT_3, HW_GPIO_PIN_4,
            HW_SPI_WORD_16BIT, HW_SPI_POL_LOW, HW_SPI_PHA_MODE_0,
                        HW_SPI_FREQ_DIV_2, -1);
#endif
```

Note: By default, the SDK comes with a few predefined device configurations in the platform_devices.h header file. 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 *prvSPITask_MCP_4822* 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:
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