Tutorial

Flash Adapters

For the DA1468x SoC

Abstract

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

Contents

Figures

Tables

Terms and Definitions

CAT	Container Allocation Table
CLI	Command Line Interface
ID	Identifier
LLD	Low Level Drivers
SDK	Software Development Kit
SUOTA	Software Update Over the Air
VES	Virtual EEPROM

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 settings of Flash Adapters
- Give a complete sample project demonstrating the usage of Flash Adapters

1.2 Adapters Concept

The DA1468x family of devices supports several peripherals on different interfaces. To support them, the SmartSnippets ™ DA1468x SDK provides Low Level Drivers (LLDs) as well as adapters for each of the available peripherals. Both of these elements add an abstraction layer on top of the hardware functions, allowing a more intuitive approach for using the hardware peripheral. This makes software porting easier.

Low Level Drivers (LLDs)

- Provide simple HW abstraction (register addresses, access functions, and so on)
- Are not aware of operating system
- Add zero overhead to the application
- Avoid race conditions when using HW

Adapters

- Applications consist of more than one task
- Two or more tasks will need the same resource
- Adapters make sure access to the resources is managed

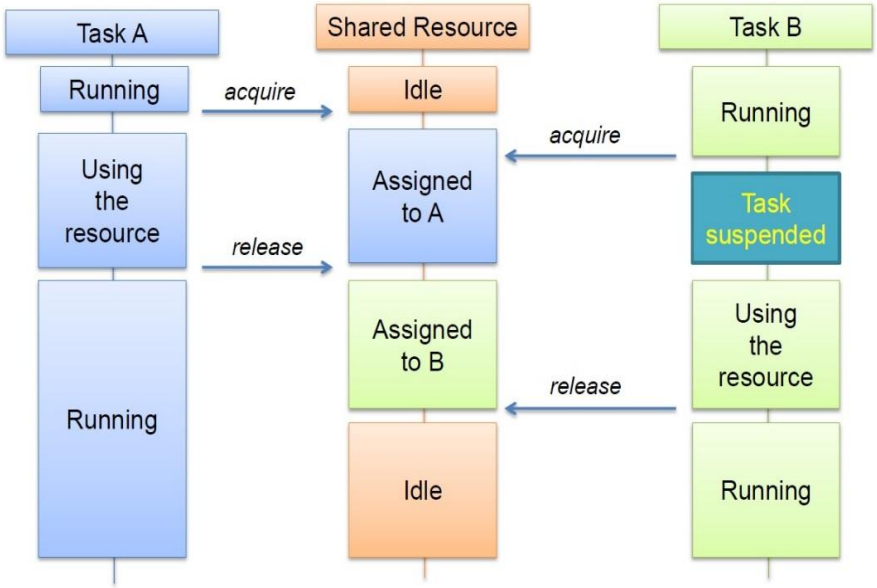

Figure 1: Adapters Communication

Adapters provided by SmartSnippets facilitate requests for a specific driver or resource. To achieve this, they utilize mechanisms offered by the OS such as semaphores and events, as well as the resource management API that facilitates multiple simultaneous resource acquisitions/releases. We do not need to worry about the internal structure of the adapters since this is out of the scope of this tutorial. Other advantages of adapters are that:

- 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, thus utilizing processor time more efficiently. After the transfer finishes, the calling freeRTOS task is released and resumes its execution.
- They ensure that system does not interrupt any current activity. Using adapters, the Power Manager (PM) is aware of the specific peripheral/resource (for example, UART or SPI) and before the system enters sleep, the PM checks whether there is activity on that specific peripheral or not.

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.

2 Flash Adapter Concept

This section explains the main features of Non-Volatile memory adapters as well as the steps to enable and correctly configure the peripheral adapters for accessing a memory through the QSPI controller. The procedure is a two-step process as illustrated in Figure 2

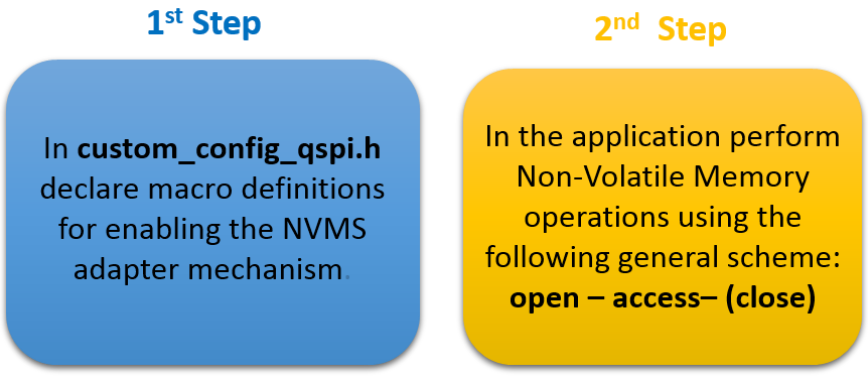

Figure 2: The Two-Step Process for Setting the Flash Adapter Mechanism

In contrast with the serial peripheral and GPADC adapters, configuring the non-volatile memory storage (NVMS) adapters is a two-step process. The key differences with the aforementioned adapters are:

- The signals for accessing a flash memory through the QSPI controller are mapped on dedicated pins on the DA1468x SoC.
- There is no need to declare device parameters as it is assumed that only one device/flash is connected on the QSPI bus.

In both cases, the NVMS mechanism automatically configures the correct signals and bus parameters during NVMS operations.

2.1 Header Files

All 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 external Non-Volatile Flash Memory. Table 1 briefly explains the header files related to flash adapters (red indicates the path under which the files are stored, and green indicates which ones are used for flash operations).

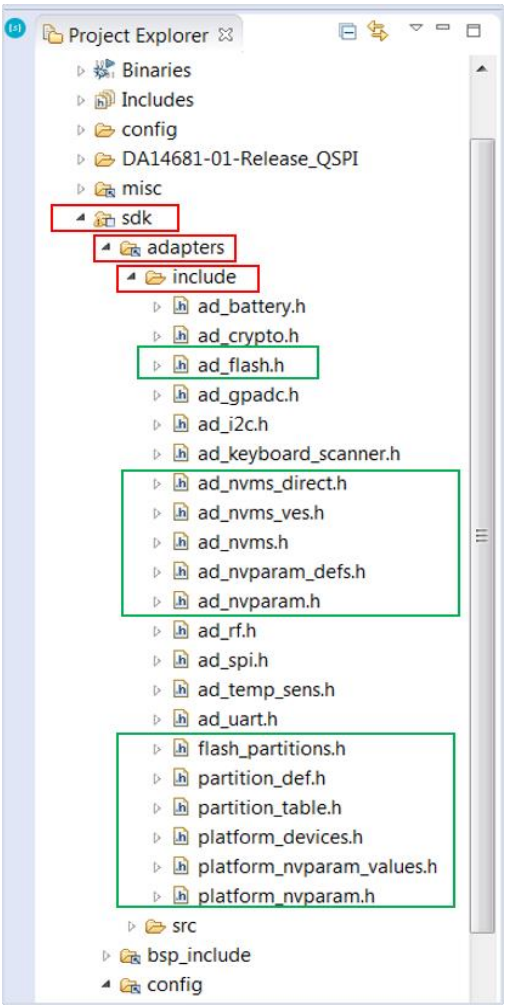

Figure 3: Headers for Flash Adapters.

Table 1: Header Files used by Flash Adapters

Filename	Description
ad_flash.h	This file contains APIs for performing the actual flash operations.
ad_nvms.h	This file contains APIs for handling all entries of a partition table. Use these APIs for accessing a region in the flash and performing read/write/erase operations.
ad_nvparam.h	This file contains APIs for handling the NVMS_PARAM_PART partition entry. Use these APIs when accessing that specific region in the flash. Alternatively, use APIs from the previously described header file.
ad_nvparam_defs.h	This file contains macros for declaring and configuring the area in the NVMS_PARAM_PART partition entry. Use these macros for declaring a custom structure within the NVMS_PARAM_PART partition entry.
ad_nvms_direct.h	This file contains macros and APIs for handling flash in Direct mode.

ad_nvms_ves.h	This file contains macros for handling flash in VES mode (Virtual EEPROM).
partition_def.h	This file contains macros as well as structures with respect to the partition table.
partition_table.h	This file selects the partition table used in the application, depending on the developer's configurations.
platform_nvparam.h	This file contains the default structure of the NVMS_PARAM_PART partition entry.
platform_nvparam_values.h	This file contains field values for the NVMS_PARAM_PART partition entry. Use this file to write the required field values. Please note that this file is not used in a regular build.

2.2 Preparing an NVMS Operation

1. As illustrated in Figure 4, to configure the NVMS adapter mechanism you need to enable it by defining the following macros in the project's /config/custom_config_qspi.h header file:

```
/*
 * Macros for enabling the NVMS adapter mechanism
 */
#define dg_configFLASH_ADAPTER          (1)
#define dg_configNVMS_ADAPTER           (1)

/*
 * Additional mechanism for accessing the [NVMS_PARAM_PART] partition entry.
 */
#define dg_configNVPARAM_ADAPTER        (1)

/*
 * This feature is only applied in [NVMS_GENERIC_PART] partition entry
 * and is mandatory to enable it when accessing that specific area!
 */
#define dg_configNVMS_VES               (0)
```

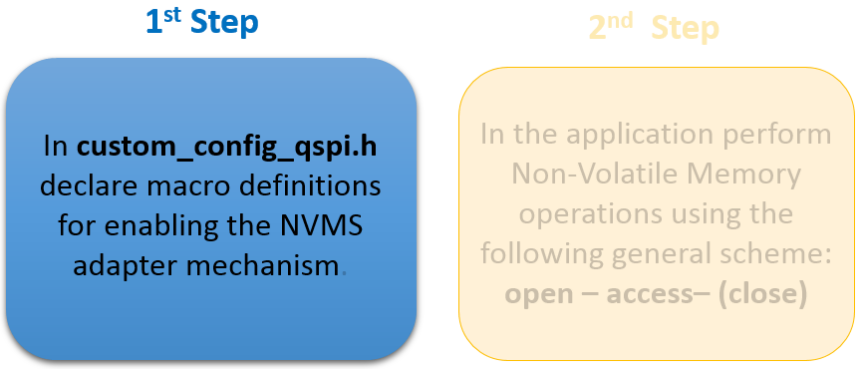

Figure 4: First Step for Configuring the NVMS Adapter Mechanism

The overall adapter implementation with all its integrated functions is now available.

2. As the NVMS adapter mechanism is enabled, the developer can use the available APIs to perform NVMS operations. Carry out the following sequence of APIs in an application to successfully perform read/write flash memory operations. The NVMS related APIs can be found in /sdk/adapters/include/ad_nvms.h.

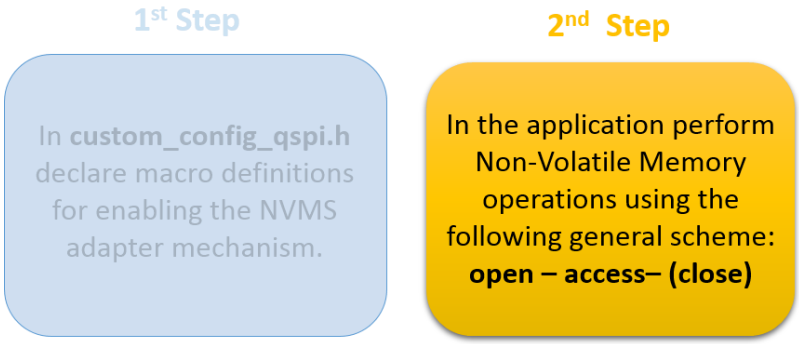

Figure 5: Second Step for Configuring the NVMS Adapter Mechanism

a. Call ad_nvms_init() once (for instance, at platform start in system_init()) to perform the necessary initialization routines, including discovering all the underlying storage partitions.

b. Call ad_nvms_open() to open a partition entry. This must be done before any read/write/delete activity can be performed. Valid partition entries are those found in the nvms_partition_id_t enumeration.

c. Call the appropriate APIs to write/read/read data in the flash memory.

Note: If several partitions are stored in one physical device (QSPI Flash), opening one partition will limit reads and writes to this partition only, making all addressing relative to the beginning of this particular partition and not to the whole flash.

The SDK for the DA1468x family of devices provides an additional layer for accessing the **NVMS_PARAM_PART** partition entry. To successfully access the aforementioned partition entry, carry out the following sequence of APIs in an application. All the related APIs can be found in /sdk/adapters/include/ad_nvparam.h.

1. Call ad_nvparam_open() to open the NVMS_PARAM_PART partition entry. This must be done before any read or write activity can be performed.
2. Call the appropriate APIs to access the specific partition entry.
3. Call ad_nvparam_close() to release and free any resource previously allocated by the ad_nvparam_open() API.

Note: The default name associated with this partition entry is ble_platform and is declared in platform_nvparam.h header file, that is NVPARAM_AREA(ble_platform, NVPARAM_PART, 0x0000).

2.3 Handling NVMS Operations

This section briefly explains how NVMS adapters are managed by the SDK.

The macro **dg_configDISABLE_BACKGROUND_FLASH_OPS** is used to define the way flash operations are handled by the SDK. This macro is located in /sdk/config/bsp_defaults.h. When the macro is set to zero (default value), all flash operations take place when the system is idle, that is, when there is no task in progress. This demonstrates the importance of giving the system the time to perform flash operations (when requested). The reasons for this approach are:

- In general, flash operations are a slow process compared with other peripheral operations. In addition, when a write operation is attempted and the flash is not clear (that is, its content is not 0xFF), the application code must erase the entire sector where the data dwells. For example, the Winbond 8-Mbit Flash, which is the default flash, contains sectors of 4 KBytes each. This means that even if only one byte is 'dirty' and a write operation is performed in the area where that byte resides, the whole sector must be erased. Specifically, when an erase operation is issued due to a dirty region, the application code does the following:
    1. The whole sector is copied into a buffer in RAM. The three settings for controlling this buffer are located in the ad_nvms_direct.h header file. By default, a static buffer is used to ensure that the memory is always available.
    2. A write operation is performed. This writes the requested data to the buffer at the point it would be written if it were in flash.
    3. The sector is erased.
    4. The sector is written with the contents of the previously written buffer.
- When performing flash operations, the SDK disables all the interrupts of the system to prevent another task from accessing the flash while a flash operation is in progress. By default, the SDK has been configured to write 128 bytes at a time to the flash buffer (Winbond 8-Mbit has a buffer of 256 bytes). The smaller this value is, the more time it takes for a flash operation to finish. On the other hand, the smaller the value, the more frequently the system suspends flash operations to serve pending interrupts (issued while a flash operation was in progress).

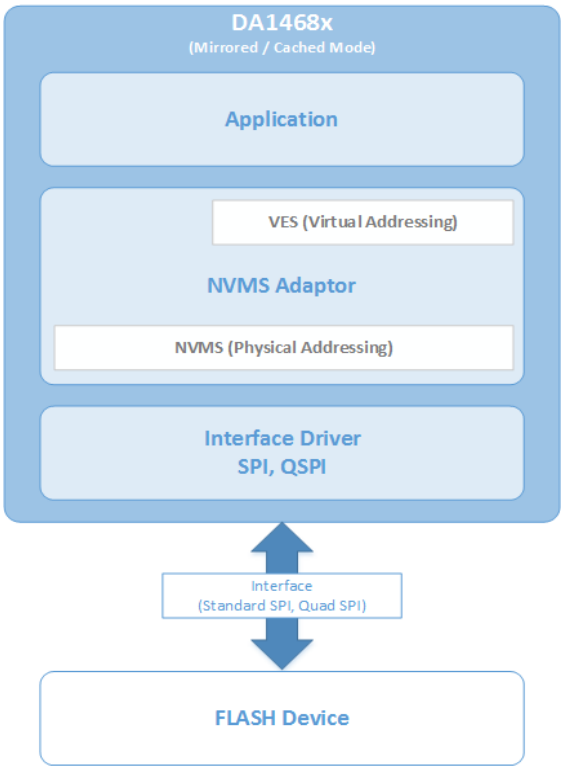

Figure 6: NVMS Overview

2.4 Modes of Operation

Before discussing the supported modes of operation, we need to mention some limitations introduced by flash memories:

1. Data cannot always be written to flash without performing an erase cycle first.
2. An erase cycle is limited to a full sector, that is, a whole sector must be erased (for the used Winbond, each sector is 4 kBytes in size which is a relatively big chunk of data).
3. A sector can be erased a limited number of times only (guaranteed by manufacturer). After that number of erase cycles, data storage is unreliable.

The SDK currently supports two different modes: Direct Access and VES (Virtual EEPROM).

2.4.1 Direct Access

Direct access drivers use relative addresses, from the beginning of a partition entry, without performing any address translation. This means that all writes are performed exactly at the requested address. If a write will not change data (that is, the same data is written) it will not be performed at all. If a write cannot be performed without an erase, then an erase operation is also initiated.

Disadvantages:

- Power failure during a write or erase operation will result in data loss, including data that was not touched by the last write.
- Writing small amounts of data at the same location many times, will result in wearing the flash (the number of write/erase cycles are device-specific, see the corresponding Flash datasheet).

2.4.2 VES

VES drivers provide access to a partition entry with power failure and wearing protection. To achieve this, VES drivers write data to random locations within the flash without needing to erase a whole sector when the same location is modified. This is accomplished by writing to different flash locations for the same user provided address. The VES driver provides virtual addressing, that is, the user specified address is translated to a real flash location before a read or write operation. For this to work, the flash size must be bigger than the addressing space visible to the user. A common rule of thumb is 8x the virtual EEPROM size needed.

This rule is employed in SDK using the **AD_NVMS_VES_MULTIPLIER** macro, found in the ad_nvms_ves.h header file. In particular, for the 1 MByte flash model the generic partition is 128 kBytes (0x20000) hence the virtual address space is around:

partition_entry_size / AD_NVMS_VES_MULTIPLIER = 128 / 8 = 16 kBytes.

In addition, the flash sectors are divided into a number of containers, where each container holds the data for a range of virtual EEPROM addresses. The size of a container is compile time configurable in 2^n bytes and by default has been configured to 64 Bytes using the **AD_NVMS_VES_CONTAINER_SIZE** macro in ad_nvms_ves.h.

A Container Allocation Table (CAT) stored in RAM, is used for tracking where valid containers are located and a Sector Allocation Table (SAT) holds the status (%dirty, free) of each sector. The selected size for a container is a trade-off between the amount of RAM needed for the CAT and the potential number of erase cycles per sector. The smaller the size of a container, the more RAM is occupied. In particular, each entry in the CAT consists of 2 bytes. Hence, in our case for the 12 kBytes of virtual EEPROM, the formula for calculating the occupied RAM is: **12288 / (64- 4) = 205 CAT entries = 205 * 2 = 410 bytes**.

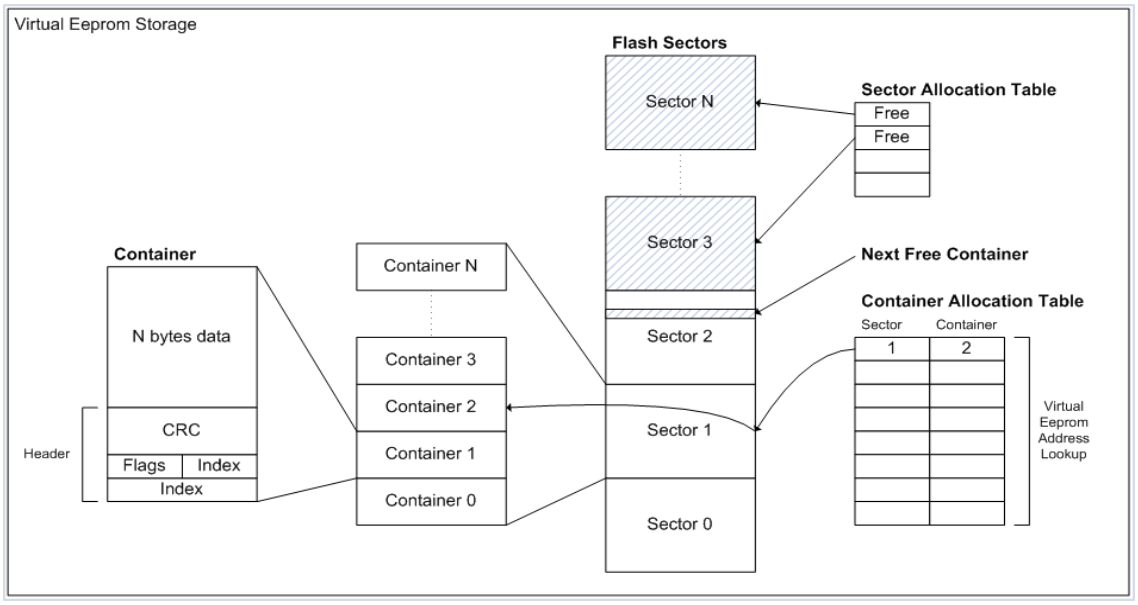

Figure 7: Virtual EEPROM Storage Model

Note: The VES feature should only be used when a small chunk of data is written/modified frequently in flash. As mentioned earlier, the only area marked as VES is the NVMS_GENERIC_PART partition entry. The BLE persistent storage mechanism provided by Dialog uses this partition entry and thus, the VES feature must be enabled. Otherwise, all the related operations will fail.

3 Non-Volatile Memory Storage

This section analyzes the way the SDK handles the Non-Volatile Memory Storage. The DA1468x SDK defines a set of storage classification rules that allow proper storage handling and budget estimation. For each storage type, a dedicated region is mapped in the flash memory which can be identified by a unique ID. Table 2 explains the available partition IDs, defined in /sdk/adapters/partition_def.h. The exact memory mapping depends on the flash memory model (size, sector size) and needs to be specified at compile time. The SDK provides a few ready-to-use partition tables that can be found under /sdk/config. By default, the **1M** model is enabled and perfectly fits in the ProDK QSPI Winbond W25Q80EW 8-Mbit flash memory (1 Mbyte with sectors of 4 kB). For more information on the used Flash, read its Datasheet. Other partition tables for different sized flash memories (on a custom board) can be selected by declaring the appropriate macro in /config/custom_config_qspi.h.

Table 2: NVMS Partition IDs

Tag ID	Description
NVMS_FIRMWARE_PART	This entry is used during a non-SUOTA/SUOTA enabled application. When in a non-SUOTA application, this entry contains the current application firmware version, whereas in a SUOTA application, it contains the bootloader that manages the firmware update process.
NVMS_PARAM_PART	This entry is used during a non-SUOTA/SUOTA enabled application for storing BLE related information (for example, the BD address of the device).
NVMS_BIN_PART	This entry is used during a non-SUOTA enabled application for storing binaries.
NVMS_LOG_PART	This entry is used during a non-SUOTA/SUOTA enabled application for logging events or values.
NVMS_GENERIC_PART	This entry is used during a non-SUOTA/SUOTA enabled application for storing generic data such as bonding data. This is the only area marked as VES (Virtual EEPROM)
NVMS_PLATFORM_PARAMS_PART	This entry is used during a SUOTA enabled application for storing platform-specific information.
NVMS_PARTITION_TABLE	This entry is used during a non-SUOTA/SUOTA enabled application and contained information on the partition table used.
NVMS_FW_EXEC_PART	This entry is used during a SUOTA enable application and contains the current application firmware version.
NVMS_FW_UPDATE_PART	This entry is used during a SUOTA enable application and contains the new updated firmware version.
NVMS_PRODUCT_HEADER_PART	This entry is used during a SUOTA enable application and contains information on the target device.
NVMS_IMAGE_HEADER_PART	This entry is used during a SUOTA enable application and contains information on the software version.

The partition layout significantly differs between a SUOTA enabled build and a non-SUOTA enabled build, as depicted in Figure 8.

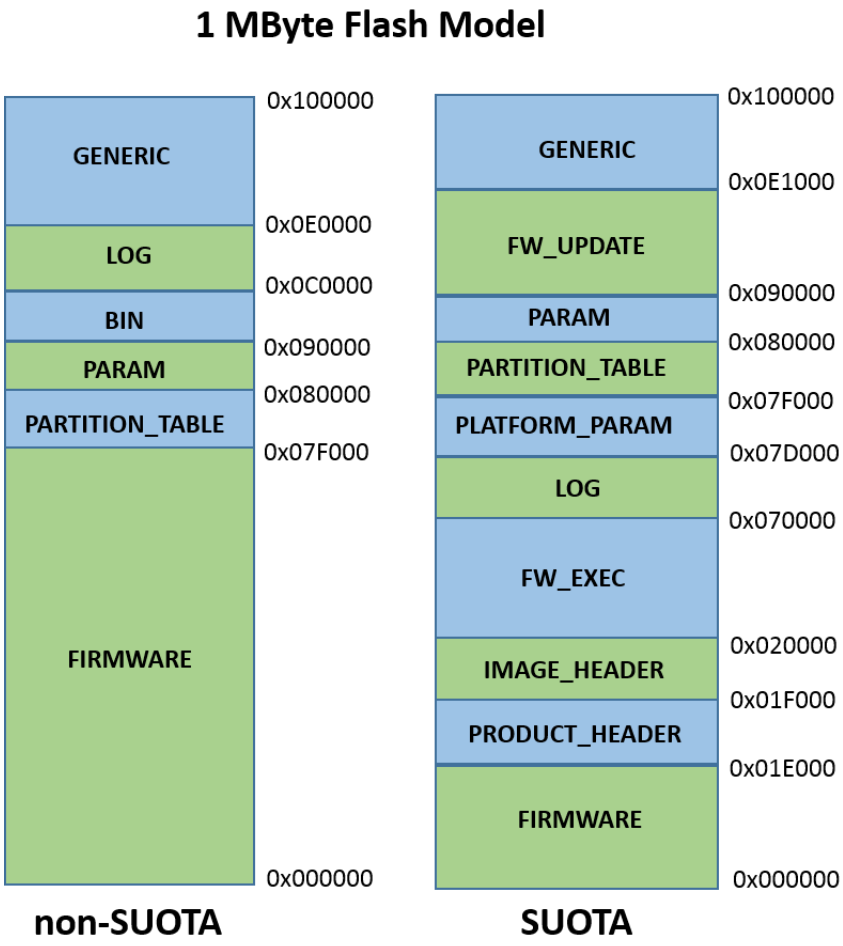

Figure 8: Partition Layouts for a non-SUOTA (left) and SUOTA (right) Enabled Application

3.1 Creating Custom Partition Tables

This section describes the steps required to successfully alter a partition table. It utilizes the default **1M** flash memory model for a non-SUOTA enabled application and slightly modifies it. It splits the default **NVMS_BIN_PART** into three equal-sized areas. To do this, the SDK provides some macros in /sdk/adapters/include/flash_partitions.h.

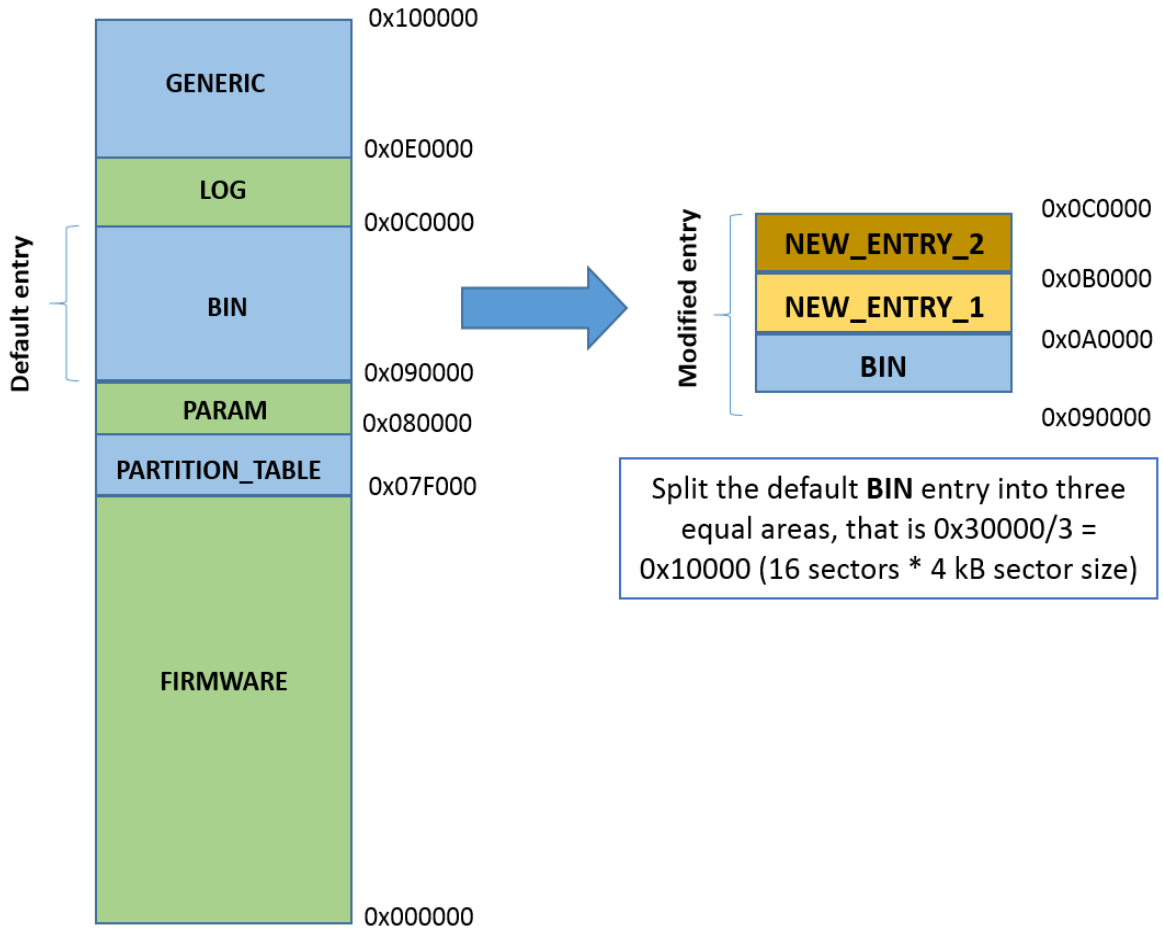

Figure 9: Modifying the 1M Model Partition Table

Warning:

1. The size of a partition entry should be multiple of sector size, 4 kB in our case (this is device-specific information).
2. The first declared entry should always be the place where the firmware dwells. The minimum allowable size is **128 kB** and it should be **64 kB** aligned. Keep in mind that, the DA1468x platforms has a read-only cache controller. That is, cache will not be updated when a cacheable area is re-written.
3. It is recommended not to change the default location of the **NVMS_PARTITION_TABLE**. However, if you do change the location, the starting address should be declared in /sdk/adapters/include/flash_partitions.h., using the **PARTITION_TABLE_ADDR** macro.

1. Establish a connection between the target device and your PC through the **USB2(DBG)** port of the motherboard.
2. Import a non-SUOTA demonstration example, for example, the **freertos_retarget** found in the SDK of the DA1468x family of devices.

Note: It is essential to import the folder named scripts to perform various operations such as building, debugging, downloading.

3. In the target application, create a new folder as well as a header file under /sdk/config/ directive. It should look like this:

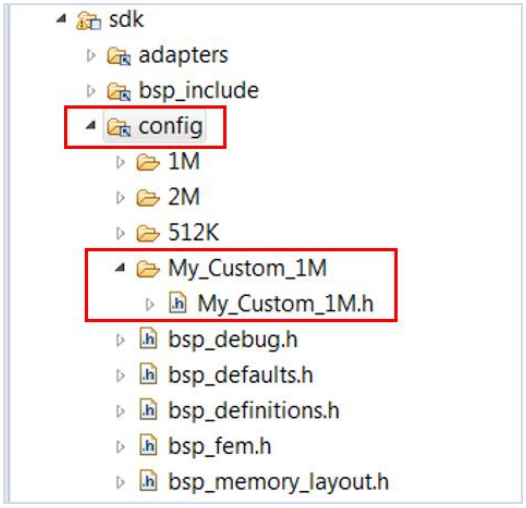

Figure 10: Creating a Custom Partition Table

4. In the newly created header file (My_Custom_1M.h) add the following code to define the new partition scheme:

```
/*
 * General form of the PARTITION2 (start, size, id, flags)
 *
 * \[start] The physical start address of the partition entry in Flash
 * \[size]  The size of the partition entry in bytes. Since the underlying flash
 *          consists of 4K sectors, a partition entry should be multiple of 4 kBytes
 *          (0x1000)
 *
 *
 * \[id]     A value from the [nvms_partition_id_t] enumerator or a custom one.
 * \[flags]  Indicates the permission attributes. Valid values are:
 *
 *           0,  PARTITION_FLAG_READ_ONLY,  PARTITION_FLAG_VES
 */

PARTITION2( 0x000000 , 0x07F000 , NVMS_FIRMWARE_PART                          , 0 )
PARTITION2( 0x07F000 , 0x001000 , NVMS_PARTITION_TABLE ,PARTITION_FLAG_READ_ONLY )
PARTITION2( 0x080000 , 0x010000 , NVMS_PARAM_PART                             , 0 )
PARTITION2( 0x090000 , 0x010000 , NVMS_BIN_PART                               , 0 )
PARTITION2( 0x0A0000 , 0x010000 , NVMS_CUSTOM_ENTRY_ONE                       , 0 )
PARTITION2( 0x0B0000 , 0x010000 , NVMS_CUSTOM_ENTRY_TWO                       , 0 )
PARTITION2( 0x0C0000 , 0x020000 , NVMS_LOG_PART                               , 0 )
PARTITION2( 0x0E0000 , 0x020000 , NVMS_GENERIC_PART,          PARTITION_FLAG_VES )
```

5. Modify the nvms_partition_id_t enumerator, located in sdk/adapters/include/partition_def.h to add new IDs for the newly defined entries. A possible modification is illustrated below:

```
/**
 * \brief NVMS Partition IDs
 */
typedef enum {
        NVMS_FIRMWARE_PART                = 1,
        NVMS_PARAM_PART                   = 2,
        NVMS_BIN_PART                     = 3,
        NVMS_LOG_PART                     = 4,
        NVMS_GENERIC_PART                 = 5,
        NVMS_PLATFORM_PARAMS_PART         = 15,
        NVMS_PARTITION_TABLE              = 16,
        NVMS_FW_EXEC_PART                 = 17,
        NVMS_FW_UPDATE_PART               = 18,
        NVMS_PRODUCT_HEADER_PART          = 19,
        NVMS_IMAGE_HEADER_PART            = 20,

        /*
         * New IDs for the newly defined entries!
         */
        NVMS_CUSTOM_ENTRY_ONE             = 21,
        NVMS_CUSTOM_ENTRY_TWO             = 22,
} nvms_partition_id_t;
```

6. Modify the /sdk/adapters/include/partition_table.h header file to include another condition for selecting the new partition scheme. It should look like this:

```
#if defined(USE_PARTITION_TABLE_2MB)
#include <2M/partition_table.h>
#elif defined(USE_PARTITION_TABLE_2MB_WITH_SUOTA)
#include <2M/suota/partition_table.h>
#elif defined(USE_PARTITION_TABLE_512K)
#include <512K/partition_table.h>
#elif defined(USE_PARTITION_TABLE_512K_WITH_SUOTA)
#include <512K/suota/partition_table.h>
#elif defined(USE_PARTITION_TABLE_1MB_WITH_SUOTA)
#include <1M/suota/partition_table.h>
#elif defined(USE_MY_CUSTOM_PARTITION_TABLE)
#include <My_Custom_1M/My_Custom_1M.h>
#else
#include <1M/partition_table.h>
#endif
```

7. In custom_config_qspi.h header file add the macro for selecting the new partition scheme.

```
#define USE_MY_CUSTOM_PARTITION_TABLE
```

8. Erase the whole flash memory contents either via the serial port or jtag interface. The following uses the second option.
    a. Run the script to erase the flash through jtag.

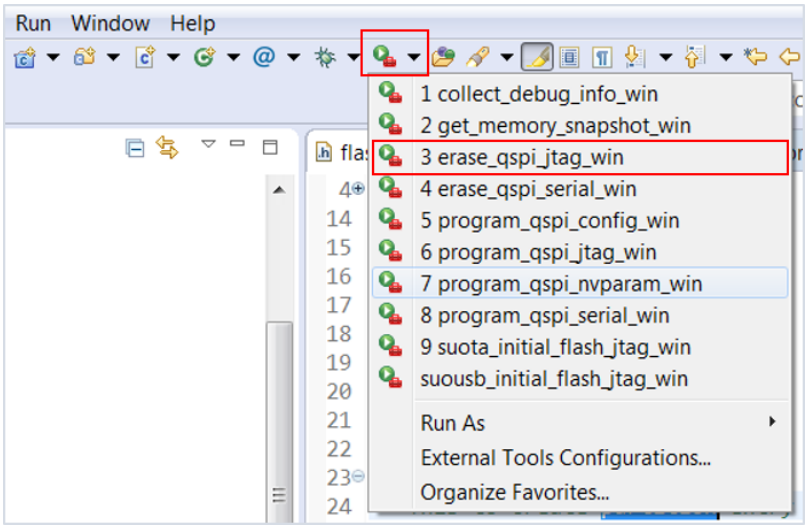

Figure 11: Select the Flash Erase Script

b. In the **Console** window at the bottom of the IDE, enter 'y' and then press **Enter**. Wait for the process to complete.

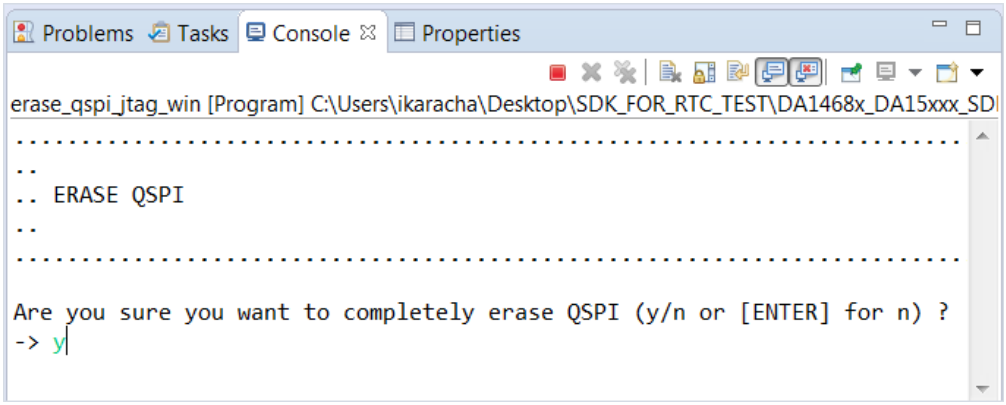

Figure 12: Erase the Flash

Warning: When changing a partition table, it is essential to erase the old one in order for the new one to be taken into consideration.

9. Build the project either in **Debug_QSPI** or **Release_QSPI** mode and burn the generated image to the chip.

3.1.1 Verifying with the SmartSnippets Toolbox

1. Open a new instance of the SmartSnippets Toolbox and switch to the **QSPI Partition Table** window (1).

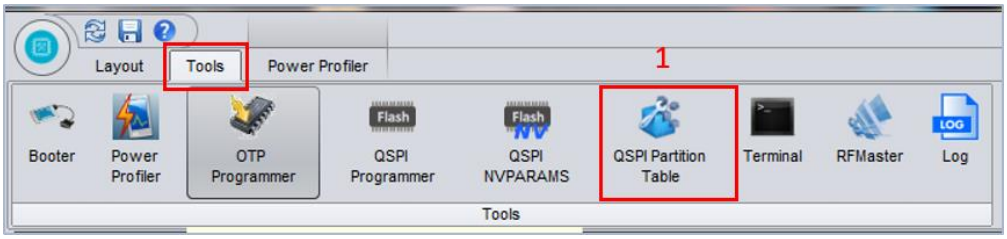

Figure 13: SmartSnippets Toolbox - Display the Partition Table Area

2. In the **Partition Table** area, click **Connect** (2). A rotating cursor is displayed waiting for the connected device to reset.
3. Press button **K2** on the DevKit to reset the device.
4. Wait for the cursor to stop rotating and click **Read** (3).

   All the partition entries are displayed. The custom defined entries will be displayed as **Unknown** areas since their corresponding IDs are not recognized by the SmartSnippets Toolbox.

Note: If the new partition table is not shown or updated, unplug and then plug the USB cable from the USB2(DBG) port of the motherboard, wait for the device to connect and then execute the steps 1 – 4 again.

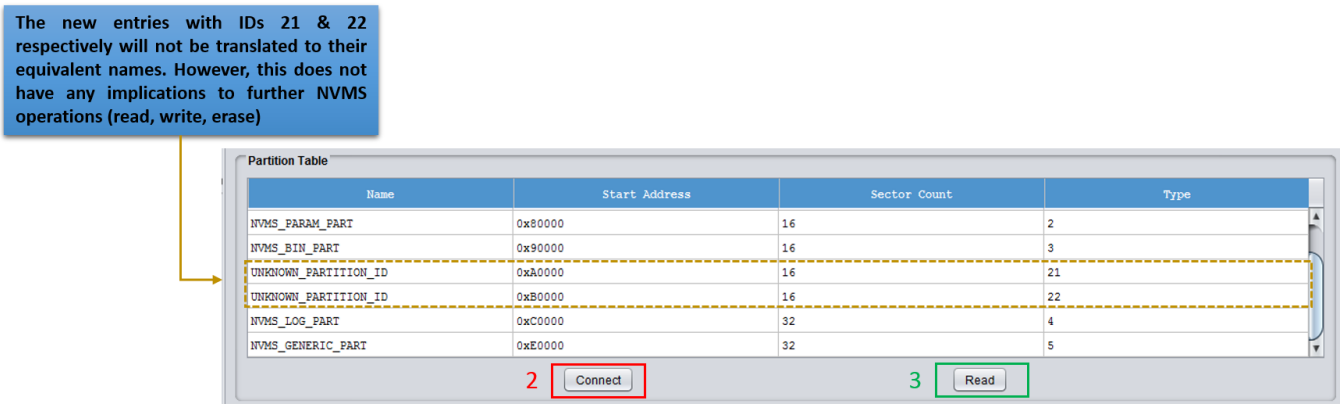

Figure 14: SmartSnippets Toolbox, Partition Table Area

3.2 NVPARAM Flash Partition Entry

The SDK provides an additional layer for accessing the **NVMS_PARAM_PART** partition entry. The internal structure is declared in /sdk/adapters/include/platform_nvparam.h and consists of thirteen fields (all starting with **TAG_BLE_PLATFORM_**). Each field value includes an additional byte which indicates its validity. This value must be set to 0x00 to be considered valid. The SmartSnippets SDK also provides a script for burning the predefined structure in flash memory. All the preferred values should be declared in platform_nvparam_values.h. This file is not used in a regular build. Instead, it will be used when the program_qspi_nvparam_win script is explicitly used.

Flash Adapters

1. Using the previous non-SUOTA sample code, in /sdk/adapters/include/platform_nvparam_values.h, declare the preferred values including their validity flags. For this demonstration, let's modify the value for the BD address field:

```
//                                                  ,-- parameter value
//                                                  |                                   ,-- 'validity' flag
//                                                  V                                   V
NVPARAM_PARAM_VALUE( TAG_BLE_PLATFORM_BD_ADDRESS,        uint8_t,  0x01, 0x02, 0x03, 0x04, 0x05, 0x06,          0x00)
```

Note: During a BLE application, if this field is valid, the BLE manager will bypass any default or user-defined BD address with this one.

2. Run the script to populate the NVMS_PARAM_PART partition entry with the defined structure.

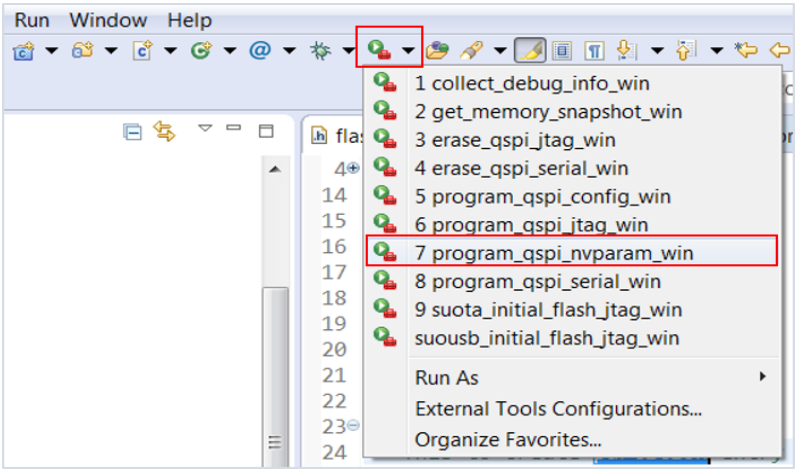

Figure 15: Script for Writing the NVMS_PARAM_PART Partition Entry

3. Open a new instance of the SmartSnippets Toolbox and switch to the **QSPI NVPARAMS** window (1).

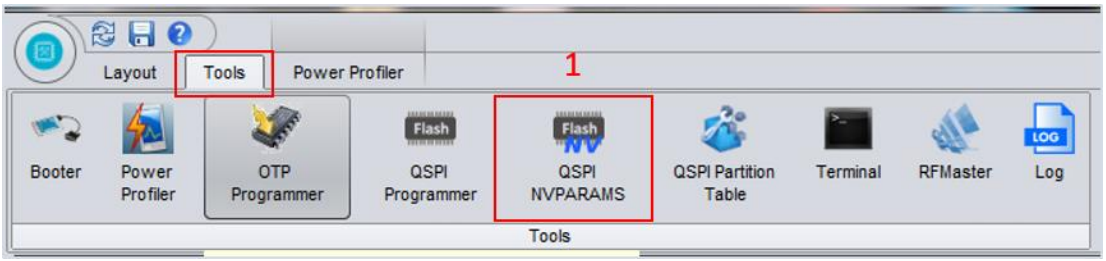

Figure 16: SmartSnippets Toolbox - Display the NVPARAMS Area

a. In the **QSPI NVPARAMS** area, click **Connect** (located at the bottom). A rotating cursor is displayed waiting for the connected device to reset.

b. Press button **K2** on the DevKit to reset the device.

c. When the cursor stops rotating, click **Read** (located at the bottom).

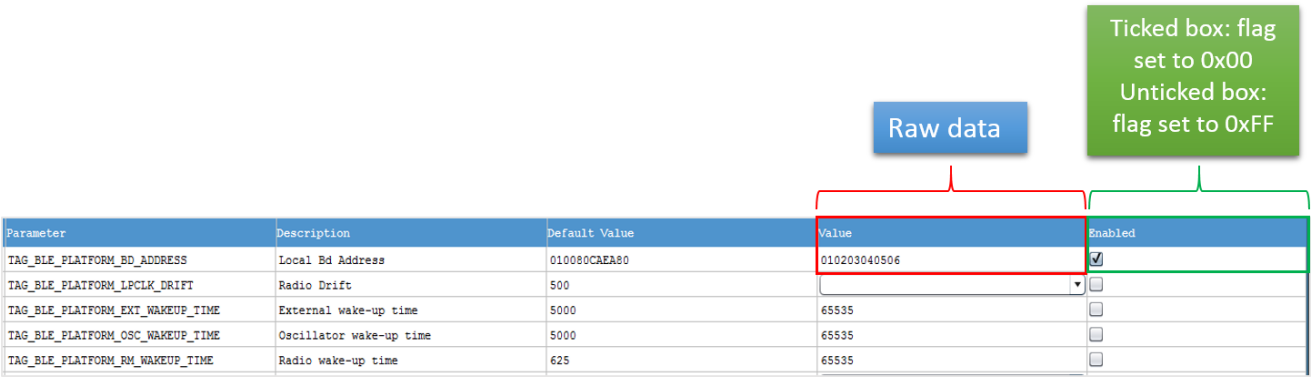

Figure 17: SmartSnippets Toolbox, NVPARAMS Area

4 Analyzing The Demonstration Example

This section analyzes an application example which demonstrates using the flash 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 various flash operations and enables the wake-up timer for handling external events. Both the NVMS and NVMS_PARAM APIs are demonstrated.

4.1 Application Structure

The key goal of this demonstration is for the device to perform a few flash operations upon an event. The button **K1** 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 called flash_state is toggled. It can take two different values which are interpreted as follows:

- **flash_state = 1**
  - A write access is attempted in TAG_BLE_PLATFORM_BD_ADDRESS field of the NVMS_PARAM_PART partition entry. The first byte is set to 0x00, thus validating the stored BD address.
  - A write access is attempted in NVMS_LOG_PART partition entry. A text message is stored indicating that the BD address stored in flash is valid.
  - A read access is attempted in NVMS_LOG_PART partition entry to read the status of the BD address. The read data is printed out on the serial console.

- **flash_state = 0**
  - A write operation is attempted in TAG_BLE_PLATFORM_BD_ADDRESS field of the NVMS_PARAM_PART partition entry. The first byte is set to 0xFF, thus invalidating the stored BD address.
  - A write operation is attempted in NVMS_LOG_PART partition entry. A text message is stored indicating that the BD address stored in flash is invalid.
  - A read access is attempted in NVMS_LOG_PART partition entry to read the status of the BD address. The read data is printed out on the serial console.

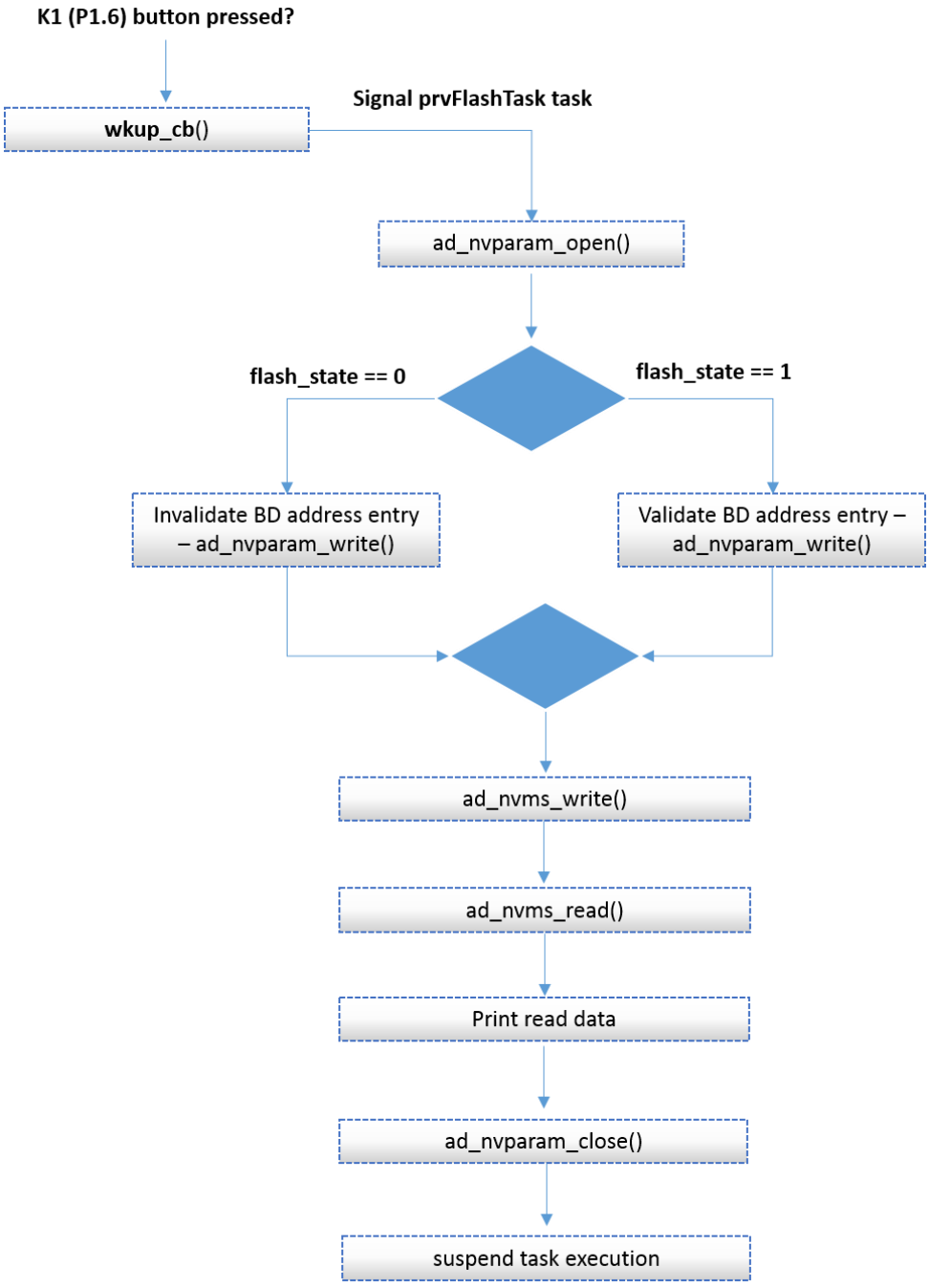

Figure 18: Flash read/write SW FSM – Main Execution Path

Note: It is essential for the system to enter the idle mode, that is all the OS tasks are either blocked or suspended. By default, the Background Options mechanism is enabled allowing the execution of flash memory related operations only when the system is idle.

2. For debugging purposes, LED D2 on Pro DevKit is used to indicate a flash operation is in progress. The LED blinks once at every **K1** button press, indicating that accesses to flash memory have been successfully executed.

To further safeguard the code, assertions are used. At the end of each read/write access to flash memory, the APIs return the actual number of read/written bytes. If a flash operation fails to be executed, the API returns a value equal to zero as none of the requested bytes were written/read. Hence, the condition in the assertion is false and the code execution gets stack at that point. Hence, allowing the developer to identify the point of interest during a debugging session.

```
/*
 * The function returns the actual number of written data.
 */

wd_log_bytes = ad_nvms_write(nvms_var, 0, (uint8_t *)log_value_1_wd,
                                                    sizeof(log_value_1_wd));

/*
 * If condition if false, the code will get stuck right here.
 */
OS_ASSERT(wd_log_bytes != 0);
```

5 Running The Demonstration Example

This section describes the steps required to prepare the Pro DevKit and other tools to successfully run the example code. A serial terminal, the SmartSnippets Toolbox (a tool delivered along with the SmartSnippets IDE), and the Command Line Interface (CLI) programmer are required for testing and verifying the code. If you are not familiar with the recommended process on how to clone a project or configure a serial terminal, read the Starting a Project tutorial.

There are two main methods to verify the correct behavior of the demonstrated code and inspect the contents of the flash memory. The first method is to use the SmartSnippets Toolbox and the second is to use the CLI programmer provided by the SDK. To facilitate the developer a third method, using a serial terminal, is also provided.

5.1 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 with the DA1468x SoC. For this tutorial, a Pro DevKit is used.

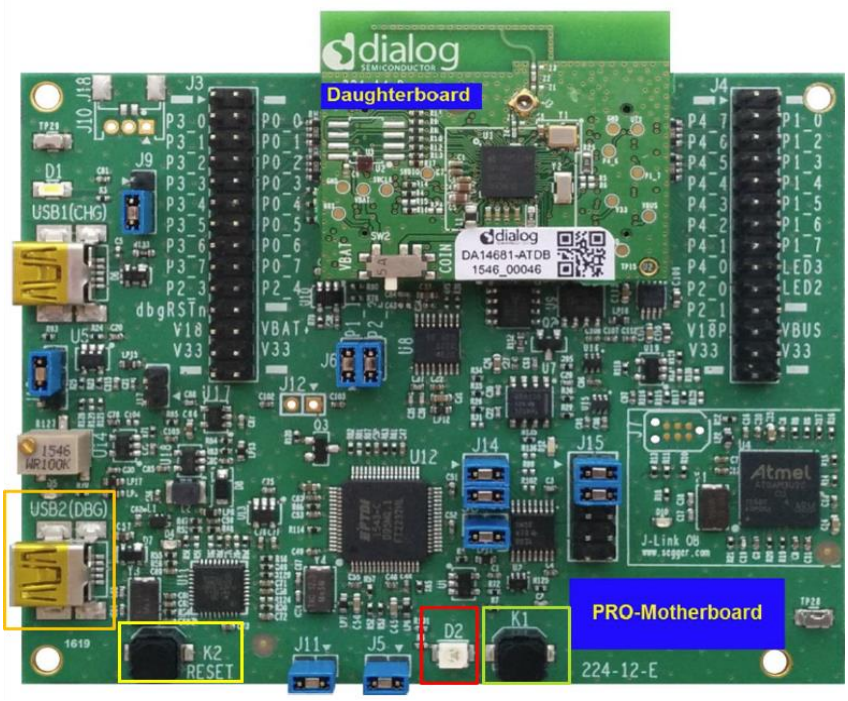

Figure 19: DA1468x Pro DevKit

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, downloading).

3. In the newly created project, add/modify the required code blocks as illustrated in the Code Overview section.

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

4. Build the project in either **Debug_QSPI** or **Release_QSPI** mode and burn the generated image to the chip (either via the serial or jtag port).
5. Press the **K2** button on Pro DevKit to start the chip executing its firmware.
6. 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 the validation of the BD address stored in the flash memory. (1)
7. Press the **K1** button on Pro DevKit again. A new debugging message is displayed on the console indicating the invalidation of the BD address. (2)

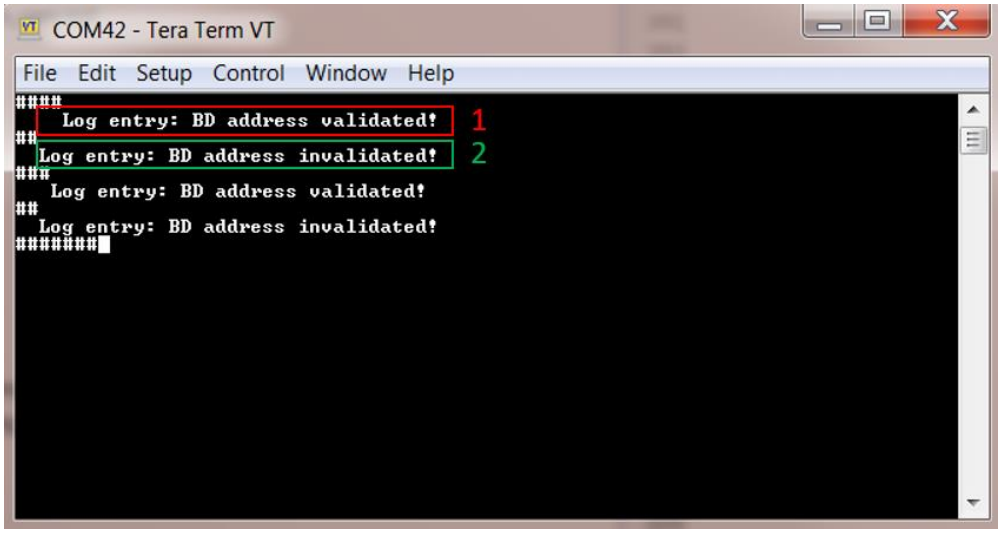

Figure 20: Debugging Messages for NVMS Operations

5.2 Verifying with the SmartSnippets Toolbox

1. With the system up and running, open the SmartSnippets Toolbox and execute the following steps:
    a. (Optional) Select **New** to create a new project (1). In the **New Project** window, enter a name for the project (2). This step is optional if a project has already been created.
    b. Choose an available project (4).
    c. Choose a communication interface (3) and a port (5).
    d. Select the family of devices to use (6).
    e. Open the selected project (7).

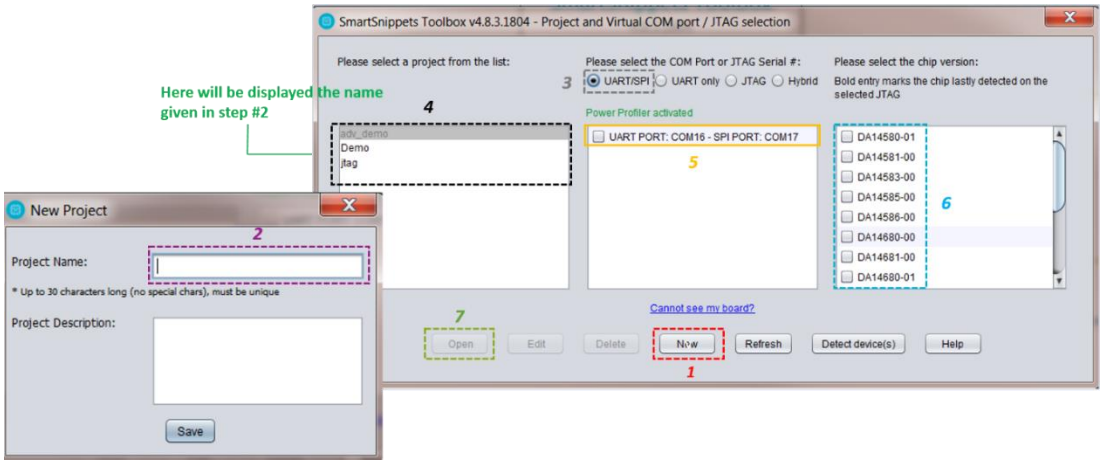

Figure 21: Opening a Project in the SmartSnippets Toolbox

2. Inspect the contents of the flash memory.
    a. Switch to the **QSPI Partition Table** window (1).

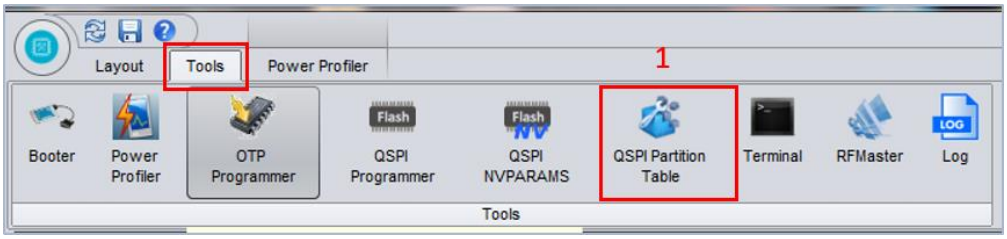

Figure 22: SmartSnippets Toolbox - Display the Partition Table Area

b. In the **Partition Table** area, click **Connect** (2). A rotating cursor is displayed waiting for the connected device to reset.

c. Press button **K2** on the DevKit to reset the device.

d. Wait for the cursor to stop rotating and click **Read** (3). All the partition entries are displayed.

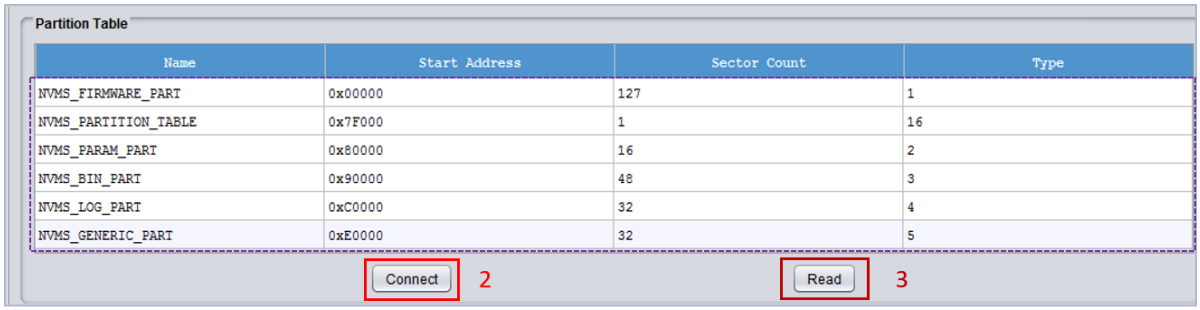

Figure 23: SmartSnippets Toolbox, Partition Table Area

3. In the **Partition Tasks** area, select the **NVMS_LOG_PART** partition entry (4).

4. Press **Read** (5) and, after a while, the contents of the selected partition entry are displayed (6).

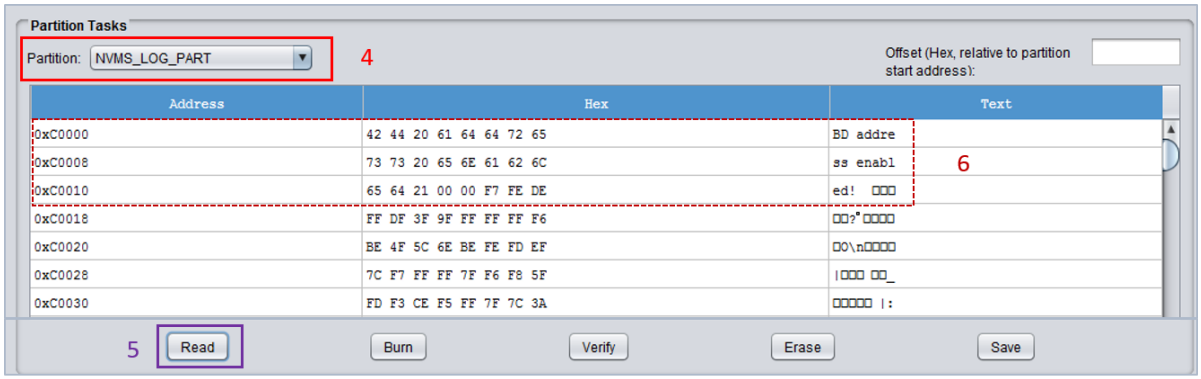

Figure 24: SmartSnippets Toolbox, Partition Tasks Area

5.3 Verifying with the Command Line Interface (CLI) Programmer

The CLI programmer is a command line tool for reading/writing the flash/OTP/RAM. The tool communicates with the target device over a UART port or JTAG interface. This section demonstrates the JTAG interface. For more information on using the CLI programmer, please read the UM-B-044 User Manual: DA1468x Software Platform Reference (Appendix B).

1. With the system up and running, start the gdbserver (J-link GDB Sever) using the configuration shown in Figure 26.

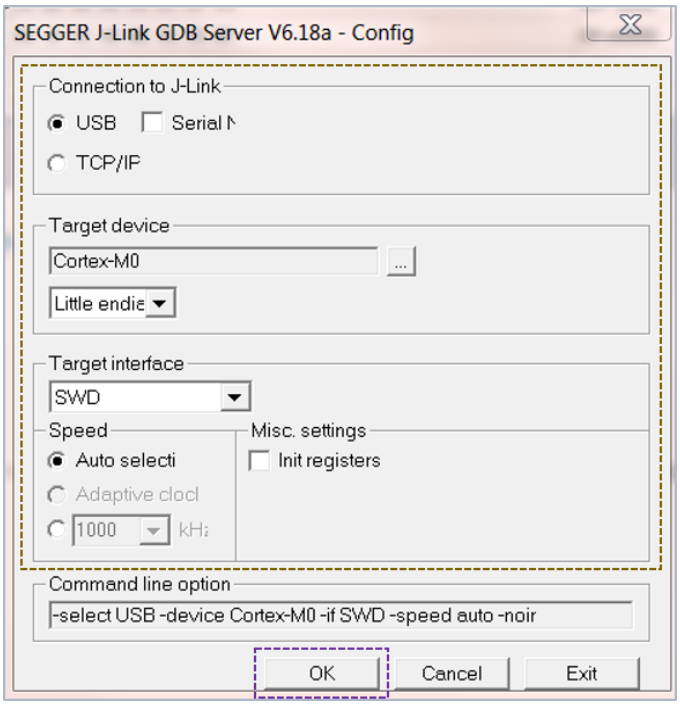

Figure 25: Configuration of the GDB Server

2. Open a command prompt window (cmd) under the path where the CLI programmer is located. By default this is the <sdk_root_directory>/binaries folder of the SDK.
3. Enter the command to read the flash memory over JTAG. The general form of this command is:

   cli_programmer gdbserver read_qspi <address_in_flash> -- <number of bytes to read>

   For example, to read the first 25 bytes from the LOG partition entry (starting from the physical memory address 0xC0000), enter:

   cli_programmer gdbserver read_qspi 0xC0000 – 25

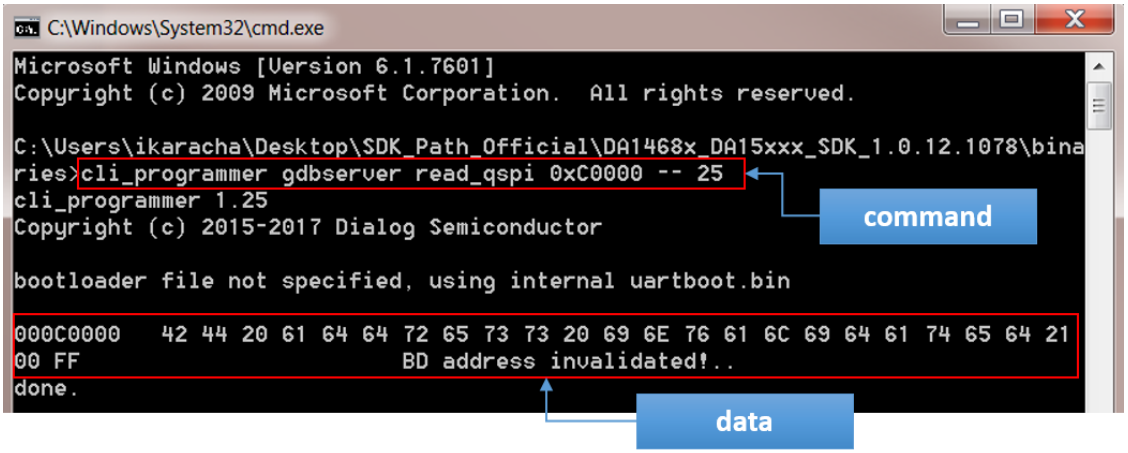

Figure 26: Reading Flash Contents with the CLI Programmer

6 Code Overview

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

6.1 Header Files

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

```
#include "hw_wkup.h"

#include "ad_nvparam.h"
#include "ad_nvms.h"
#include  <platform_nvparam.h>
```

6.2 System Init Code

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

```
/* OS signals used for synchronizing OS tasks */
static OS_EVENT signal_flash;

/* Flash memory application task – Function prototype */
static void prvFlashTask( void *pvParameters );

/* Flash memory application task priority */
#define mainFLASH_TASK_PRIORITY    ( OS_TASK_PRIORITY_NORMAL )

/* Flag for selecting Flash memory operations. */
volatile static bool flash_state = 0;


static void system_init( void *pvParameters )
{
      OS_TASK task_h  = NULL;
      OS_TASK flash_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 only be called
       * 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 signal. */
    OS_EVENT_CREATE(signal_flash);

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


    /* Start main task here */
    OS_TASK_CREATE( "Template",   /* The text name assigned to the task,
                                     for debug only; not used by the kernel. */

            prvTemplateTask,   /* The function that implements the task. */
            NULL,              /* The parameter passed to the task */

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

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

    /*
     * Task responsible for flash memory operations.
     */
    OS_TASK_CREATE( "FLash",
            prvFlashTask,      /* The function that implements the task. */

            (void *)flash_h,    /* The parameter passed to the task. */

            200 * OS_STACK_WORD_SIZE,
            mainFLASH_TASK_PRIORITY,   /* The priority assigned to the task. */

            flash_h );                 /* The task handle */
    OS_ASSERT(flash_h);


    /* The work of the SysInit task is complete */
    OS_TASK_DELETE( xHandle );
}
```

In **main()**, slightly modify the task creation of system_init(). The stack size should be increased, for instance to 500 bytes. The following code snippet shows how it should look like:

```
        status = OS_TASK_CREATE("SysInit",
                    system_init,
                    ( void * ) 0,
                    500,          /* Modified code line! */

                    OS_TASK_PRIORITY_HIGHEST,
                   xHandle );
```

6.3 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 flash status.
       */
      flash_state ^= 1;

      /*
       * Time for resuming [prvFlashTask] task has elapsed.
       */
      OS_EVENT_SIGNAL_FROM_ISR(signal_flash);
}

/*
 * 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);
}
```

6.4 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.

```
/**
 * @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);


    /* This pin drives the D2 Led on the Pro DevKit (for debugging purposes) */
    hw_gpio_set_pin_function(HW_GPIO_PORT_1, HW_GPIO_PIN_5,
                        HW_GPIO_MODE_OUTPUT, HW_GPIO_FUNC_GPIO);
}

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

6.5 Flash Task Code

Code snippet of the **prvFlashTask** task responsible for interacting with the externally connected flash memory of the system. In **main.c**, add the following code (for instance, after system_init()):

```
/*
 * Task responsible for interacting with the externally
 * connected flash memory of the system.
 */
static void prvFlashTask( void *pvParameters )
{

    uint16_t wd_param_bytes = 0;

    uint16_t wd_log_bytes = 0;
    uint16_t rd_log_bytes = 0;


    /*
     * Data to be written during a write access to
     * [NVMS_PARAM_PART] partition entry.
     */
     uint8_t bd_address_nvparam_wd[] = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x00};

    /*
     * Buffer used for storing data during a read access to
     * [NVMS_LOG_PART] partition entry.
     */
    char log_value_rd[25];

    /*
     * Data to be written during a write access to
```

```
 * [NVMS_LOG_PART] partition entry.
 */
char log_value_1_wd[] = "BD address validated!";
char log_value_2_wd[] = "BD address invalidated!";

/*
 * Handler for accessing [NVMS_PARAM_PART] partition entry.
 */
nvparam_t param;

/*
 * Handler for accessing [NVMS_LOG_PART]  partition entry.
 */
nvms_t nvms_var;

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

/*
 * Before accessing a partition entry, you should first open it.
 */
nvms_var = ad_nvms_open(NVMS_LOG_PART);

for (;;) {

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

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

        /*
         * Before accessing a partition entry, you should first open it.
         */
        param = ad_nvparam_open("ble_platform");

        // Validate the BD address
        if (flash_state == 1) {

            /*
             * Validate the validity flag of BD address entry
             * (7th byte)
             */
            memset(bd_address_nvparam_wd + 6, 0x00, 1);

            /*
```

```
        * Attempt a write access to [NVMS_PARAM_PART]
        * partition entry to write the BD address field value.
        *
        *  \note The function returns the actual number of written data.
        */
       wd_param_bytes = ad_nvparam_write(param,
                   TAG_BLE_PLATFORM_BD_ADDRESS,
                   sizeof(bd_address_nvparam_wd),
                   bd_address_nvparam_wd);

       OS_ASSERT(wd_param_bytes != 0);


       /*
        * Attempt a write access to [NVMS_LOG_PART] partition
        * entry to log the status of the previously written BD
        * address (validated/invalidated)
        *
        * \note The function returns the actual number of written data.
        */
       wd_log_bytes = ad_nvms_write(nvms_var, 0,
                  (uint8_t *)log_value_1_wd, sizeof(log_value_1_wd));

       OS_ASSERT(wd_log_bytes != 0);


       // ------------------ Read operations --------------------

       memset(log_value_rd, 0x20, sizeof(log_value_rd));

       /*
        * Attempt a read access to [NVMS_LOG_PART] partition
        * entry to read the current status of BD address
        * (validated/invalidated).
        *
        * \note The function returns the actual number of read data.
        */
       rd_log_bytes = ad_nvms_read(nvms_var, 0,
                  (uint8_t *)log_value_rd, sizeof(log_value_rd));

       OS_ASSERT(rd_log_bytes != 0);


       /*
        * Print the log status on the serial console.
        */
       printf("\nLog entry: %s\n\r", log_value_rd);

   // Invalidate the BD address
   } else if (flash_state == 0) {

       /*
        * Invalidate the validity flag of BD address entry
        * (7th byte)
        */
```

```
        memset(bd_address_nvparam_wd + 6, 0xFF, 1);

        /*
         * Attempt a write access to [NVMS_PARAM_PART]
         * partition entry to write the BD address field value.
         *
         * \note The function returns the actual number of written data.
         */
        wd_param_bytes = ad_nvparam_write(param,
                TAG_BLE_PLATFORM_BD_ADDRESS,
                        sizeof(bd_address_nvparam_wd),
                         bd_address_nvparam_wd);

        OS_ASSERT(wd_param_bytes != 0);

        /*
         * Attempt a write access to [NVMS_LOG_PART]
         * partition entry to log the status of the previously
         * written BD address (validated/invalidated)
         *
         * \note The function returns the actual number of written data.
         */
        wd_log_bytes = ad_nvms_write(nvms_var, 0,
                (uint8_t *)log_value_2_wd, sizeof(log_value_2_wd));

        OS_ASSERT(wd_log_bytes != 0);


        // ----------------- Read operations ------------------

        memset(log_value_rd, 0x20, sizeof(log_value_rd));

        /*
         * Attempt a read access to [NVMS_LOG_PART] partition
         * entry to read the current status of BD address
         * (validated/invalidated).
         *
         * \note The function returns the actual number of read data.
         */
        rd_log_bytes = ad_nvms_read(nvms_var, 0,
                    (uint8_t *)log_value_rd, sizeof(log_value_rd));

        OS_ASSERT(rd_log_bytes != 0);

        /*
         * Print the log status on the serial console
         */
        printf("\nLog entry: %s\n\r", log_value_rd);

    }


/* Close the already opened adapter */
ad_nvparam_close(param);
```

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


        } // end of for loop
} // end of task
```

6.6 Macro Definitions

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

```
#define dg_configFLASH_ADAPTER          (1)
#define dg_configNVMS_ADAPTER           (1)

/*
 * Additional mechanism for accessing the [NVMS_PARAM_PART] partition entry.
 */
#define dg_configNVPARAM_ADAPTER        (1)
```

Revision History

Revision	Date	Description
1.0	26-Feb-2018	First released version
1.1	05-Mar-2018	Table of contents, Missing header files in App. code, Minor typos
2.0	24-July-2018	More descriptive steps to follow, figures and examples.
2.1	17-Sept-2018	Updated figures in ‘Analyzing The *Demonstration Example’* section, Minor improvement in *prvFlashTask application* 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