User Manual

DA14585/586 SDK 6 Software Developer's Guide

UM-B-080

Abstract

This document describes the steps required to develop BLE applications on the SmartBond™ DA1458x Product Family software platform, specifically for the DA14585/586 devices, as supported by the new v6.x SDK series. It guides the developer through a number of pillar examples, acquainting in the developing of BLE applications on the DA14585/586 software architecture and APIs.

Contents

Figures

Tables

1 Terms and Definitions

AES	Advanced Encryption Standard
BLE	Bluetooth Low Energy
CPU	Central Processing Unit
DA1458x	DA1458x SoC Platform of Product Family of devices, for this document specifically referring to the DA14585/586 devices
DISS	Device Information Service Server
DK	Development Kit
GAP	Generic Access Profile
GTL	Generic Transport Layer
HCI	Host Controller Interface
HW	Hardware
MITM	Man In The Middle
NVDS	Non-Volatile Data Storage
OTA	Over The Air
OTP	One Time Programmable (memory)
SDK	Software Development Kit
SoC	System on Chip
SUOTAR	Software Update Over The Air Receiver
SUOTA	Software Update Over The Air
UUID	Universally Unique IDentifier

2 References

[1] UM-B-048, Getting Started with DA1458x Development Kits - Basic, User Manual, Dialog Semiconductor.

[2] UM-B-049, Getting Started with DA1458x Development Kits - Pro, User Manual, Dialog Semiconductor.

[3] DA14585 Data sheet, Dialog Semiconductor.

[4] DA14586 Data sheet, Dialog Semiconductor.

[5] UM-B-079, DA14585/586 Software Platform Reference, User Manual, Dialog Semiconductor.

[6] UM-B-012, DA1458x Creation of a secondary boot loader, User manual, Dialog Semiconductor

[7] AN-B-023, DA1458x Interfacing with external memory, Application note, Dialog Semiconductor

[8] UM-B-019, DA1458x Beacon reference design User Manual, Dialog Semiconductor

[9] UM-B-018, DA1458x SmartTag reference application, User manual, Dialog Semiconductor

[10] UM-B-025, DA1458x Basic Development kit, User manual, Dialog Semiconductor

[11] Bluetooth Specification version 4.2

[12] UM-B-013, DA14580 External processor interface over SPI, User manual, Dialog Semiconductor

3 Introduction

This document aims to serve as a guide to the embedded software developer by providing a step by step practical understanding on how to develop Bluetooth Low Energy standard applications, when using the system architecture of the DA14585/586 System on Chip (SoC) family of integrated circuit (IC) devices, through its development environment and tool chain.

3.1 Target Audience

This is a document for embedded software developers, also called embedded firmware engineers that are working on developing applications on any of the SmartBond™ DA14585/586 Product Family of devices which are based on the DA14585/586 System on Chip (SoC) platform.

Developers that are new to the DA14585/586 System on Chip (SoC) platform are advised to first read Ref. [5], especially the first chapters, and then scan through the rest of the reference documents, both to get familiar with the software platform and to learn where to find specific information as needed. Then spend some time reading through the sections of this guide.

Experienced embedded firmware engineers after going through the contents of Ref. [5], can focus on the pillar examples as provided in this document, and then take a deep dive into the SDK and deeper detailed technical documentation. This should allow to get a clear idea of how applications can be developed and are executed on Dialog's DA14585/586 Bluetooth® Low Energy devices as well as on how to best utilize the capabilities offered by Dialog's DA14585/586 SoC platform.

3.2 How to Use This Manual

This document describes the development steps through which a developer can develop BLE applications on the DA14585/586 software architecture, utilizing SDK 6.x and its supporting tool chain; to this extend, this document guides the developer, to check up the correctness of the setup of his/her development environment, how to use the supported tool chain to produce, run, debug and test his/her first build BLE example application, then guides him/her through the pillar BLE examples, through which he/she gets acquainted in developing complete BLE applications that use the DA14585/586 software architecture and APIs. It also explains through examples how one can use the peripherals that the DA14585/586 SoC supports as well as how to create a new project.

4 Getting Started

To get started it is important to check that we have in place the development environment. Therefore it is assumed in this document that the developer is familiar with the DA14585/586 development kits and the Keil uVision software development environment. Depending on the development kit variant, the developer should refer to UM-B-048, Getting Started with DA14585/586 Development Kits - Basic, User Manual [1], for the basic kits, and UM-B-049, Getting Started with DA14585/586 Development Kits - Pro, User Manual [2], for the professional kits.

Download instructions and installation steps on how one can download and install the development environment including drivers and tools are provided in the above mentioned Getting Started documents, [1] for the basic kits, [2] for the professional kits.

Therefore to accompany the development kit of his choice, the developer should also have already installed on his personal computer the following software applications.

4.1 Development Environment

The DA14585/586 development environment consists of:

- **ARM Keil µVision IDE/Debugger**, **ARM C/C++ Compiler**, and its essential middleware components, **Keil IDE** and the **Keil build** tools.
- **Segger ARM JTAG** software that is fully supported by the Keil environment.

4.2 Software Development Kit (SDK)

The DA14585/586 Software Development Kit (SDK) in its latest v6.x release as downloaded from the customer support web page: http://www.dialog-semiconductor.com/support.

4.3 Tools

The development environment is also supported by a number of other utilities and tools such as the **SmartSnippets Toolbox** and **Connection Manager** which are downloaded from the customer support web page: http://www.dialog-semiconductor.com/support under the "Software & Tools" menu.

4.4 SmartSnippets Toolbox

SmartSnippets is a framework of PC based tools to control DA14585/586 development kit, consisting of:

- **Booter** is used for downloading for downloading hex files to DA14585/586 SRAM over UART and for resetting the chip to execute from there.
- **UART Terminal** is available only for connection over UART. After successfully downloading the selected file to the DA14585/586 chip, the 'Start Terminal' button is activated and the user can press it in order to receive data from UART.
- **Power Profiler** is used for plotting the current (and associated charge) drawn by the DA14585/586 on the DK in real time over USB.
- **Sleep Mode Advisor** is used to help users understand how much power their application dissipates Sleep modes and what is its impact in battery lifetime duration.
- **OTP Programmer** tool is used for burning the OTP Memory and OTP Header.
- **SPI Flash Programmer** is used for downloading an image file to the SPI Flash Memory (DA14585/586).

For more details on how to use the above tools refer to the "User Guide" html document which can be found under the "help" drop down menu of the SmartSnippets Toolbox application.

4.5 Connection Manager

Connection Manager is a tool to control the link layer of the DA14585/586, with the following capabilities:

- Functional in Peripheral and Central role
- Set advertising parameters
- Set connection parameters
- Reading from Attribute database
- Perform production test commands

For more details on how to use the tool, please refer to the "Help Document" which can be found under the "Help" drop down menu of the Connection Manager application.

5 Blinky: Your First DA14585/586 Application

The Getting Started guides for the development kits provide in their "Using the Development Kit" section an example application called Blinky. It demonstrates step-by-step how one can load the Blinky example as a project in the Keil environment, how to set up and build, and lastly how to execute via the debug environment on any of the DA14585/586 devices, depending on the development kit and the exact device that the developer uses.

5.1 Application Description

Blinky is a simple application example which demonstrates basic initialization of DA14585/586 and LED blinking.

The project is located in the `<sdk_root_directory>\projects\target_apps\peripheral_examples\blinky` SDK directory. The Keil v5 project file is the:

`<sdk_root_directory>\projects\target_apps\peripheral_examples\blinky\Keil_5\blinky.uvprojx`

5.2 Hardware Configuration

The common UART terminal configuration described in section 7.4 is required.

Table 1: Blinky Example Jumper Configuration

GPIO	Function	DA14585/586 DK-Basic	DA14585/586 DK-Pro
P0_4	UART2 TX	Connect J4.11 - J4.12	Connect J5.11 - J5.12
P0_5	UART2 RX	Connect J4.13 - J4.14	Connect J5.13 - J5.14
P1_0	LED	Connect J9.1 – J9.2	Connect J9.1 – J9.2

5.3 Running the Example

Please follow the step-by-step instructions as described in the Getting Started guides for the development kits in their "Using the Development Kit" section.

After the Blinky example has been built and downloaded to the DK the LED will start to blink and the following output from program will be visible in the console.

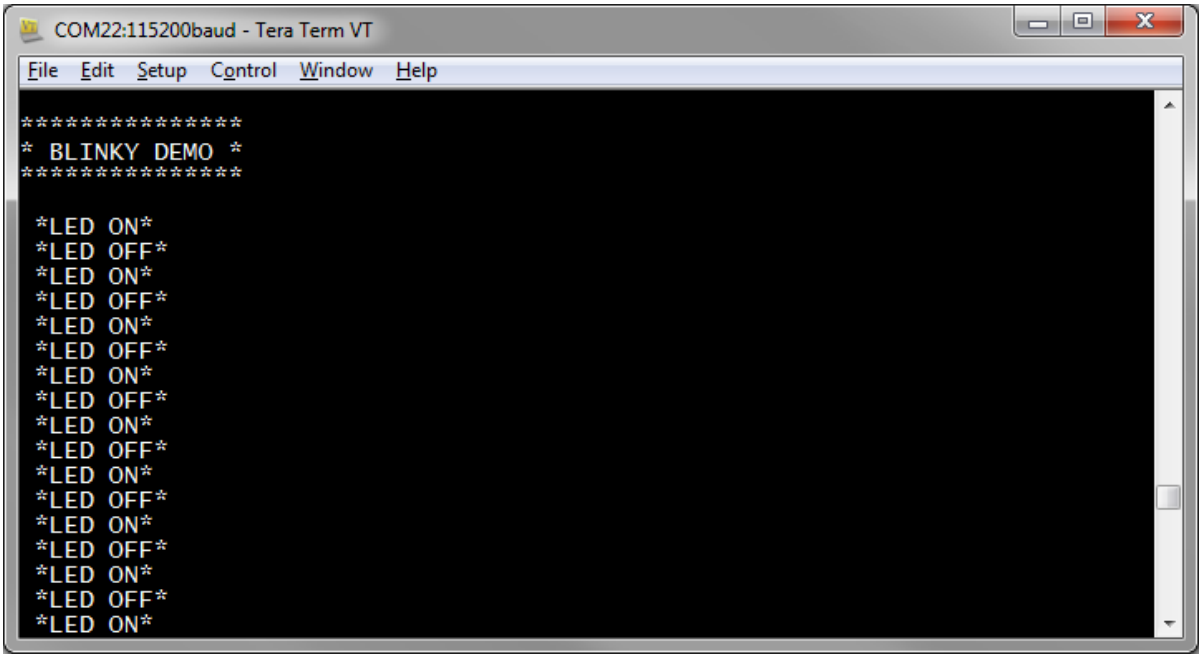


Figure 1: Blinky Example Output Console

The source code for this example is located in function `blinky_test()` inside:

`projects\target_apps\peripheral_examples\blinky\src\main.c`

6 Proximity Reporter: Your First Bluetooth Low Energy Application

6.1 Application Description

The Proximity profile defines the behavior of a Bluetooth device when this moves away from a peer device so that the connection is dropped or the path loss increases above a predefined level, causing an immediate alert. This alert can be used to notify the user that the devices have become separated.

The Proximity profile can also be used to define the behavior of two devices coming closer together such that a connection is made or the path loss decreases below a predefined level.

The Proximity profile defines two roles:

- **Proximity Monitor (PM)**. The Proximity Monitor shall be a GATT client.
- **Proximity Reporter (PR)**. The Proximity Reporter shall be a GATT server.

This section describes only the Proximity Reporter application.

6.2 Basic Operation

The Proximity Reporter application supports the following services.

- Immediate Alert service (UUID 0x1802).
- Link Loss service (UUID 0x1803).
- Tx Power service (UUID 0x1804).
- Device Information service (UUID 0x180A).
- Battery service (UUID 0x180F).
- Software Update Over The Air Receiver (SUOTAR) service (UUID 0xFEF5).

The Proximity Reporter application has the following features:

- Two levels of Alert Indications. Mild/High -> Slow/Fast green LED blinking.
- 500 ms advertising interval.
- Extended sleep without OTP copy mode after 3 minutes of inactivity.
- Push button.
- Stop Alert indications.
- Exit Extended sleep without OTP copy mode.
- Pairing / bonding / encryption.
- Supports Extended Sleep mode.

The Proximity Reporter operation is implemented in C source file `user_proxr.c`.

6.3 User Interface

The application will notify the user when an alert indication, link loss and immediate alert are triggered. The Alert Notification will be:

- **High level alert:** A fast (500 ms) LED blinking.
- **Mild level alert:** A slow (1500 ms) LED blinking.

The user can stop alert notification by pressing a push button.

The selected LED and push button (port and pin number of the DA14585/586) are defined by the user configuration depending on the underlying hardware (Development Kit). The user file `user_periph_setup.h` holds the peripheral configuration settings of the LED and push button.

6.4 Loading the Project

The Proximity Reporter application is developed under the Keil v5 tool. The respective Keil project file is the `prox_reporter.uvprojx`.

Figure 2 shows the Keil project layout with emphasis on the user related files, included in the Keil project folders `user_config`, `user_platform` and `user_app`. These folders contain the user configuration files of the Proximity Reporter application.

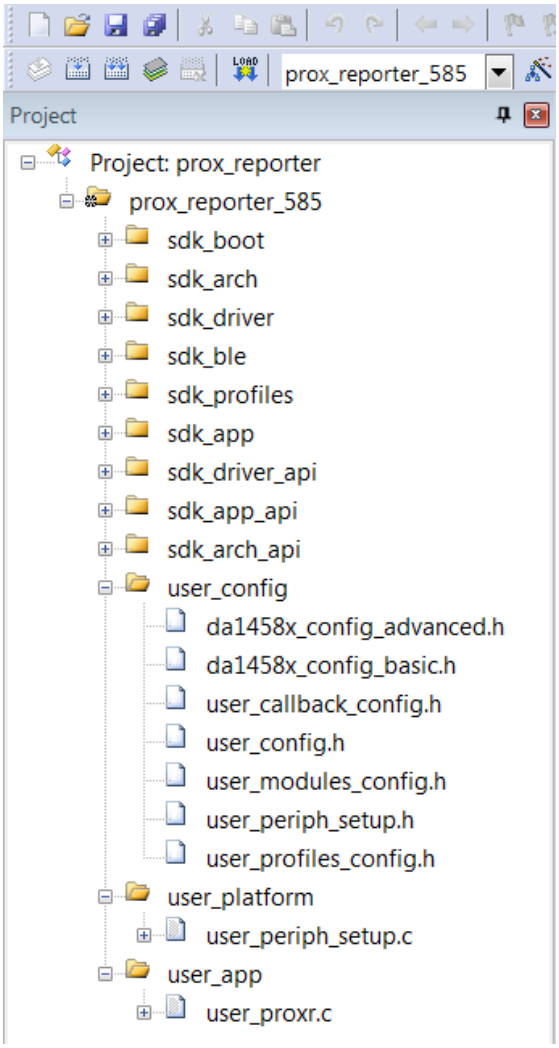


Figure 2: Proximity Reporter Keil Project Layout

6.5 Going Through the Code
6.6 Initialization

The aforementioned Keil project folders (user_config, user_platform and user_app), contain the files that initialize and configure the Proximity Reporter application.

- da1458x_config_advanced.h, holds DA14585/586 advanced configuration settings.
- da1458x_config_basic.h, holds DA14585/586 basic configuration settings.
- user_callback_config.h, callback functions that handle various events or operations.
- user_config.h, holds advertising parameters, connection parameters, etc.
- user_modules_config.h, defines which application modules are included or excluded from the user's application. For example:
  - #define EXCLUDE_DLG_DISS (0), the Device information application profile is included. The SDK takes care of the Device information application profile message handling.
  - #define EXCLUDE_DLG_DISS (1), the Device information application profile is excluded. The user application has to take care of the Device information application profile message handling.
- user_profiles_config.h, defines which BLE profiles (Bluetooth SIG adopted or custom ones) will be included in user's application. The BLE profiles that are included in the user_profile_config.h file are:
  - CFG_PRF_DISS, includes the Immediate Alert, Link Loss and Tx Power services.
  - CFG_PRF_BASS, includes the Device Information service.
  - CFG_PRF_PXPR, includes the Battery service.
- user_periph_setup.h, holds hardware related settings relative to the used Development Kit.
- user_periph_setup.c, source code file that handles peripheral (GPIO, UART, etc.) configuration and initialization relative to the selected Development Kit.

6.7 Events Processing and Callbacks

Several events can occur during the lifetime of the BLE application and these events need to be handled in a specific manner. Also, operations need to be served depending on the application scenario. It depends on the application itself to define which events and operations are handled and how. The SDK is flexible enough to either call a default handler or call the user's defined event or operation handler.

The SDK mechanism, which is provided to the user in order to take care of the above, is the registration of callback functions for every event or operation. The C header file user_callback_config.h, which resides in user space, contains the registration of callback functions.

The Proximity Reporter application registers the following callback functions:

- General BLE events:

```
static const struct app_callbacks user_app_callbacks = {
    .app_on_connection              = default_app_on_connection,
    .app_on_disconnect              = user_app_on_disconnect,
    .app_on_update_params_rejected  = NULL,
    .app_on_update_params_complete  = NULL,
    .app_on_set_dev_config_complete = default_app_on_set_dev_config_complete,
    .app_on_adv_nonconn_complete    = NULL,
    .app_on_adv_undirect_complete   = app_advertise_complete,
    .app_on_adv_direct_complete     = NULL,
    .app_on_db_init_complete        = default_app_on_db_init_complete,
    .app_on_scanning_completed      = NULL,
    .app_on_adv_report_ind          = NULL,
    .app_on_get_dev_appearance      = default_app_on_get_dev_appearance,
```



```
    .app_on_get_dev_slv_pref_params = default_app_on_get_dev_slv_pref_params,
    .app_on_set_dev_info            = default_app_on_set_dev_info,
    .app_on_data_length_change      = NULL,
    .app_on_update_params_request   = default_app_update_params_request,
#if (BLE_APP_SEC)
    .app_on_pairing_request         = default_app_on_pairing_request,
    .app_on_tk_exch_nomitm          = default_app_on_tk_exch_nomitm,
    .app_on_irk_exch                = NULL,
    .app_on_csrk_exch               = default_app_on_csrk_exch,
    .app_on_ltk_exch                = default_app_on_ltk_exch,
    .app_on_pairing_succeded        = NULL,
    .app_on_encrypt_ind             = NULL,
    .app_on_mitm_passcode_req       = NULL,
    .app_on_encrypt_req_ind         = default_app_on_encrypt_req_ind,
    .app_on_security_req_ind        = NULL,
#endif // (BLE_APP_SEC)
};
```

- The above structure defines that a certain event will be processed by a default handler or by a user defined handler or it will not be processed at all (NULL entries). The user defined handlers (e.g. app_advertise_complete()) are defined in C source file `user_proxr.c`.
- System specific events:

```
static const  struct arch_main_loop_callbacks user_app_main_loop_callbacks = {
    .app_on_init            = default_app_on_init,
    .app_on_ble_powered     = NULL,
    .app_on_sytem_powered   = NULL,
    .app_before_sleep       = NULL,
    .app_validate_sleep     = NULL,
    .app_going_to_sleep     = NULL,
    .app_resume_from_sleep  = NULL,
};
```

- The above structure defines that a certain event will be processed by a default handler or by a user defined handler or it will not be processed at all (NULL entries).
- BLE operations:

```
static const struct default_app_operations user_default_app_operations = {
    .default_operation_adv = default_advertise_operation,
};
```

- The above structure defines that a certain operation will be processed by a default handler or by a user defined handler or it will not be processed at all (NULL entries).

6.8 BLE Application Abstract Code Flow

Figure 3 shows the abstract code flow diagram of the Proximity Reporter application. The diagram depicts the SDK interaction with the callback functions registered in `user_callback_config.h` and the functions implemented in `user_proxr.c`..

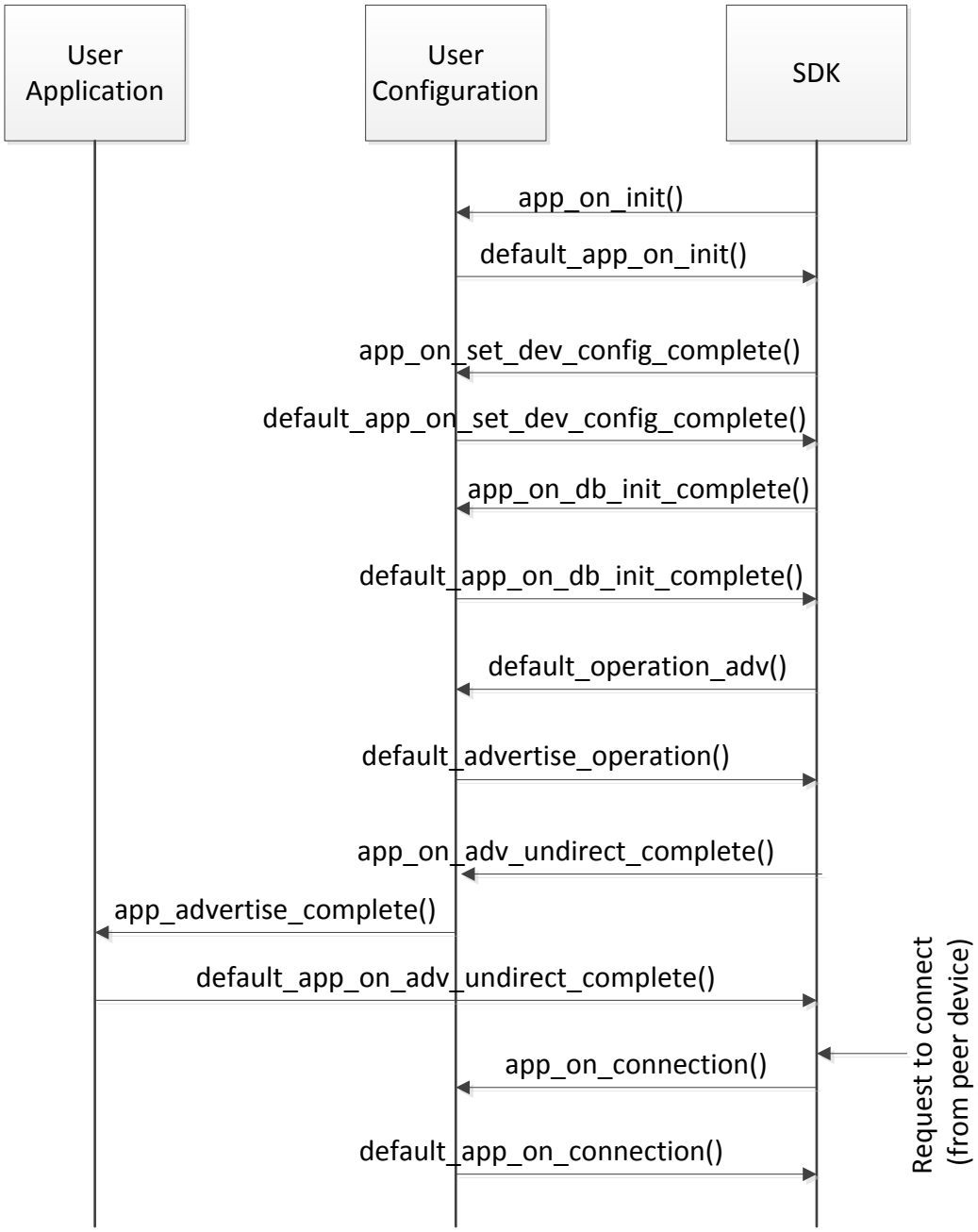


Figure 3: Proximity Reporter - User Application Code Flow

The default advertising function is the `default_advertise_operation()`, as shown in Figure 4, and it is located in `<sdk_default_directory>\sdk\app_modules\src\app_default_hnd`.

```
void default_advertise_operation(void)
{
    if (user_default_hnd_conf.adv_scenario == DEF_ADV_FOREVER)
    {
        app_easy_gap_undirected_advertise_start();
    }
    else if (user_default_hnd_conf.adv_scenario == DEF_ADV_WITH_TIMEOUT)
    {

app_easy_gap_undirected_advertise_with_timeout_start(user_default_hnd_conf.advertise_p
eriod, NULL);
    }
}
```

Figure 4: The default advertise function

The parameters of the `default_advertise_operation()` can be configured in the *user_config.h* file located in `<sdk_default_directory>\projects\target_apps\ble_examples\prox_reporter\src\config` as shown in the following code snippet (Figure 5).

```
static const struct default_handlers_configuration  user_default_hnd_conf = {
    //Configure the advertise operation used by the default handlers
    //Possible values:
    //  - DEF_ADV_FOREVER
    //  - DEF_ADV_WITH_TIMEOUT
    #if (USE_PUSH_BUTTON)
        .adv_scenario = DEF_ADV_WITH_TIMEOUT,
    #else
        .adv_scenario = DEF_ADV_FOREVER,
    #endif

    //Configure the advertise period in case of DEF_ADV_WITH_TIMEOUT.
    //It is measured in timer units (3 min). Use MS_TO_TIMERUNITS macro to convert
    //from milliseconds (ms) to timer units.
    .advertise_period = MS_TO_TIMERUNITS(180000),

    //Configure the security start operation of the default handlers
    //if the security is enabled (CFG_APP_SECURITY)
    .security_request_scenario = DEF_SEC_REQ_NEVER
};}
```

Figure 5: Advertise parameters configuration

6.9 Building the Project for Different Targets and Development Kits

The Proximity Reporter application can be built for two different target processors, DA14585 and DA14586.

The selection is done via the Keil tool as depicted in Figure 6.

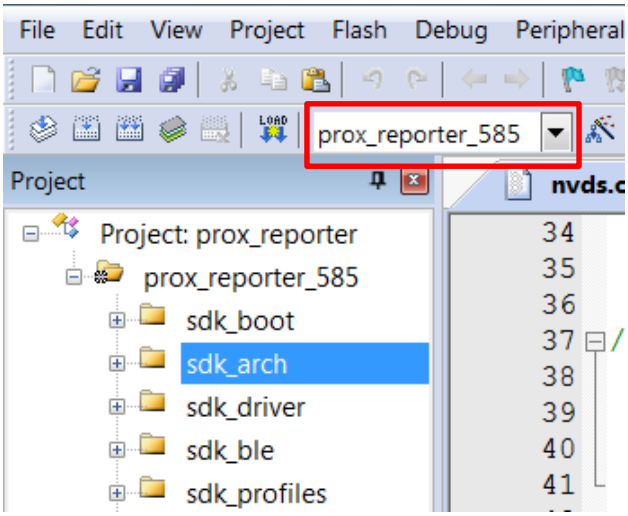


Figure 6: Building the Project for Different Targets

The user also has to select the correct Development Kit in order to build and run the application. This selection is done via the Configuration Wizard of the `user_periph_setup.h` file. See Figure 7.

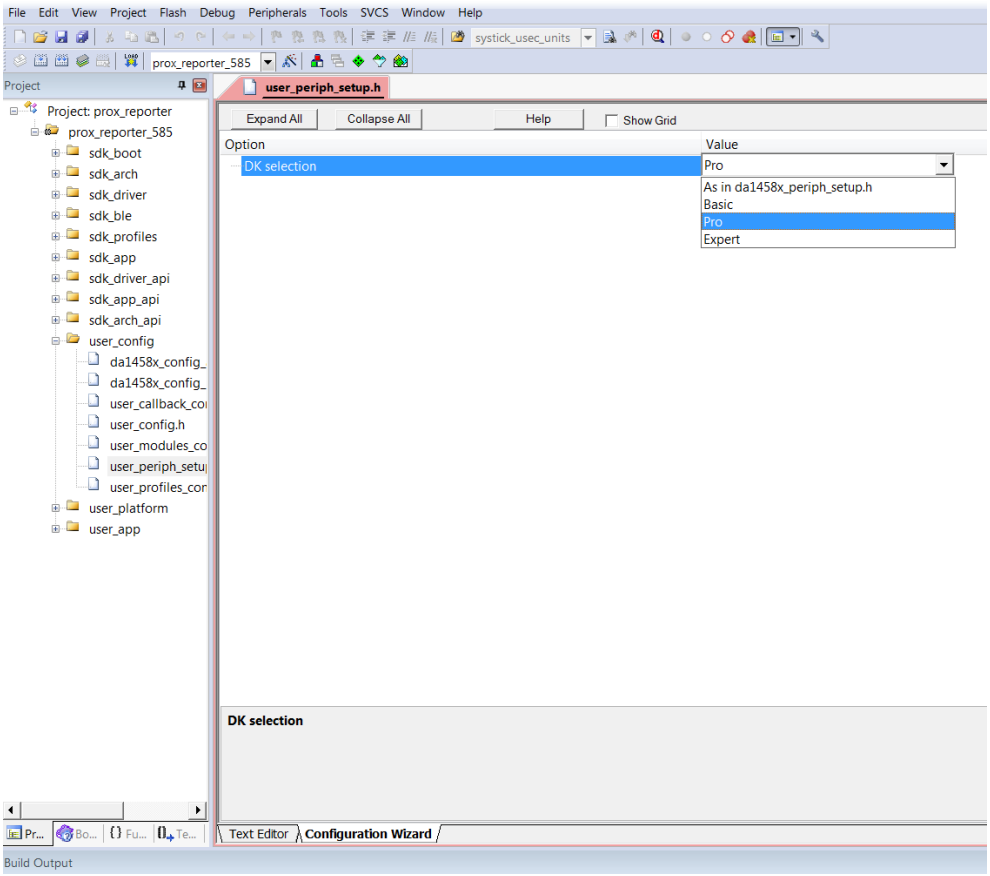


Figure 7: Development Kit Selection for Proximity Reporter Application

After the proper selection of the target processor and development kit, the application is ready to be built.

6.10 Interacting with BLE Application

6.11 LightBlue iOS Application

The LightBlue iOS application can be used to connect an iPad/iPod/iPhone device to the application. In such a case the iPad/iPod/iPhone acts as a BLE Central and the application as a BLE Peripheral. Figure 8 shows the result when the iPad/iPod/iPhone device manages to connect to the DA14585/586 (the application's advertising device name is **DIALOG-PRXR**).

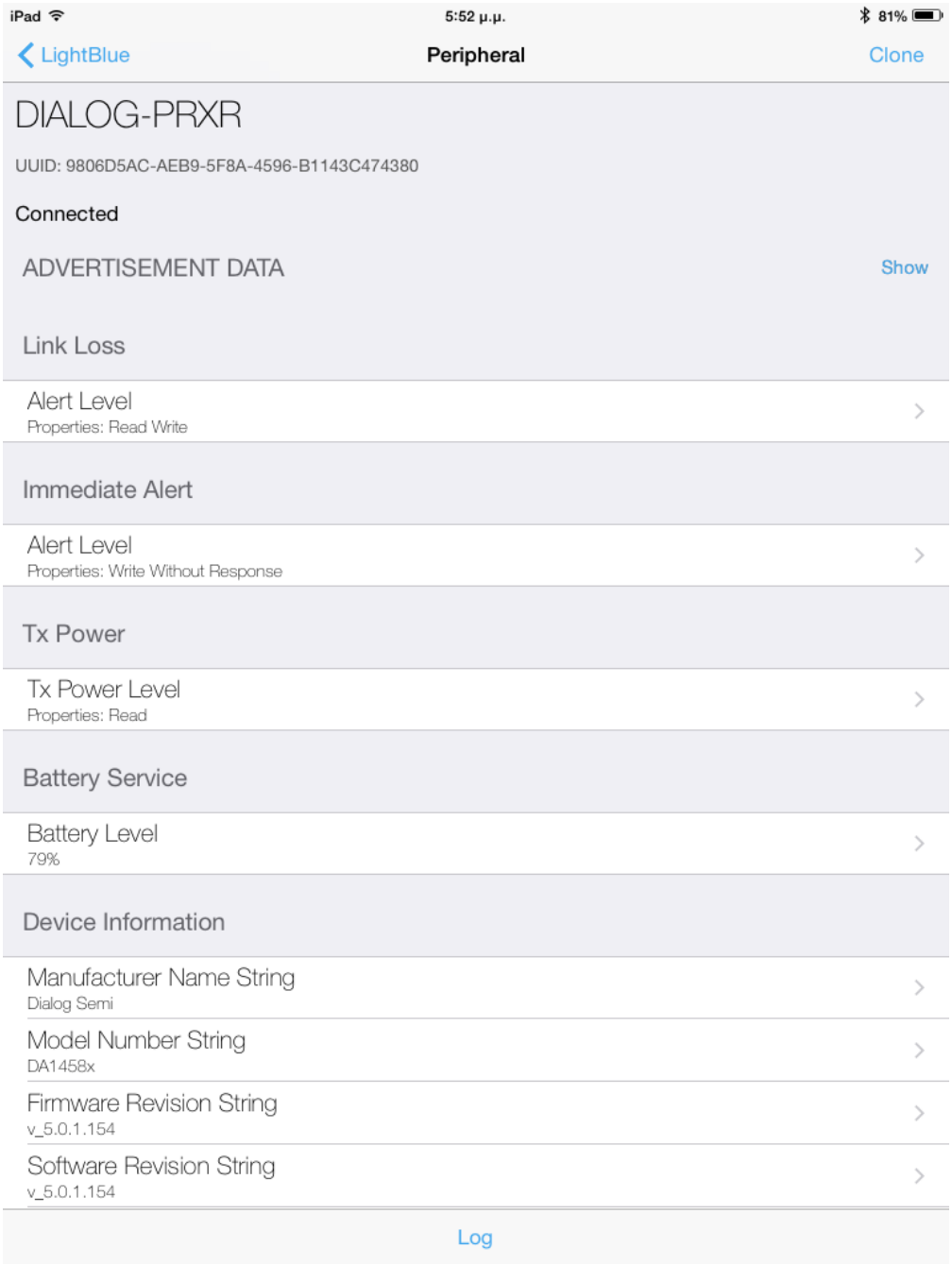


Figure 8: LightBlue Application Connected to Proximity Reporter Application

7 Peripheral Example Applications

The DA14585/586 Software Development Kit (SDK) includes a set of engineering examples which demonstrate the use of the drivers provided with the SDK for accessing the main peripheral devices of DA14585/586.

7.1 Introduction

The peripheral example applications demonstrate the DA14585/586's peripheral connectivity capabilities such as interfacing to SPI Flash and I2C EEPROM memories as well as using on chip peripherals such as the timers, the quadrature decoder, and the ADC. The user interaction, when applicable, is done via a UART terminal.

The following examples are provided:

- UART Print String Example: How to configure, initiate, and send to the UART interface.
- UART2 Asynchronous Example: How to perform IRQ based IO operations using the UART2 interface.
- SPI Flash Memory Example: How to initiate, read, write and erase an SPI Flash memory.
- I2C EEPROM Example: How to initiate, read, write and erase an EEPROM memory.
- Quadrature Encoder Example: How to configure and read from the quadrature decoder peripheral. The Wakeup Timer setup for responding to GPIO activity is also demonstrated in this example.
- Systick Example: How to use Systick timer to generate an interrupt periodically. A LED is changing its state upon each interrupt.
- TIMER0 (PWM0, PWM1) Example: How to configure TIMER0 to produce PWM signals. A melody is produced on an externally connected buzzer.
- TIMER0 general Example: How to configure TIMER0 to count a specified amount of time and generate an interrupt. LED is blinking every interrupt
- TIMER2 (PWM2, PWM3, PWM4) Example: How to configure TIMER2 to produce PWM signals. LEDs are changing light brightness in this example.
- Battery Example: How to read the battery indication level, using the ADC.

7.2 Software Description

The peripheral example applications are located in the `target_apps\peripheral_examples\` directory of the DA14585/586 Software Development Kit.

The implementation of the drivers is located in the `sdk\platform\driver` directory. To use the DA14585/586 peripheral drivers, one should:

- Add the driver's source code file (e.g. `spi\spi.c`) to the project.
- Include the driver's header file (e.g. `spi\spi.h`) whenever the driver's API is needed.
- Add the driver folder path to the Include Paths (C/C++ tab of Keil Target Options).

Each of the example projects includes the following files:

`main.c`**:** Includes both the main and test functions.

`user_periph_setup.c`**:** Includes the system initialization and GPIO configuration functions for peripheral that is about to be presented.

`user_periph_setup.h`**:** Defines the hardware configuration, such as GPIO assignment for peripheral that is about to be presented.

`common_uart.c, .h`**:** Introduces functions to printing of byte, word, double word and string variables. It calls functions from the GPIO driver.

7.3 Getting Started

It is assumed that the user is familiar with the use of the DA14585/586 Development Kits as described in Ref. [1] and Ref. [2].

Prior to running a desired peripheral example, the user has to configure the DA14585/586 development board accordingly, depending on the required hardware configuration.

Each example contains a subsection which describes the GPIO assignment of the DA14585/586 development board.

7.4 Configuring the UART Interface on a DA1458x DK

The DA14585/586 UART2 interface will be used for the UART interaction terminal.

7.4.1 DA1458x DK-Basic

On a DA14585/586 DK-Basic the user has to connect PIN11 to PIN12 and PIN13 to PIN14 on the J4 connector to enable interfacing the UART2 TX/RX to the Segger chip. No UART2 CTS/RTS functionality is required.

7.4.2 DA14585/586 DK-Pro

On a DA14585/586 DK-Pro the user has to connect PIN11 to PIN12 and PIN13 to PIN14 on the J5 connector to enable interfacing the UART2 TX/RX to the FT2232HL chip. No UART2 CTS/RTS functionality is required.

7.5 Using a Serial Port Terminal with a DA14585/586 DK

A serial port terminal application (e.g. *Tera Term*) should be used for user interaction with each peripheral example project. All examples use the following COM port settings:

```
Baud rate: 115200
Data: 8-bit
Parity: none
Stop: 1 bit
Flow control: none
```

The procedure of discovering the COM port number of a DA1458x DK is described in subsequent paragraphs.

7.5.1 Connecting to a DA14585/586 DK-Basic

When the DA14585/586 DK-Basic [1] is connected via USB to a Windows machine (e.g. laptop), a J-Link device should be discovered in the Windows Devices and Printers. The JLink CDC UART Port which is displayed in the J-Link's properties window (Figure 9) must be used in the serial port terminal application.

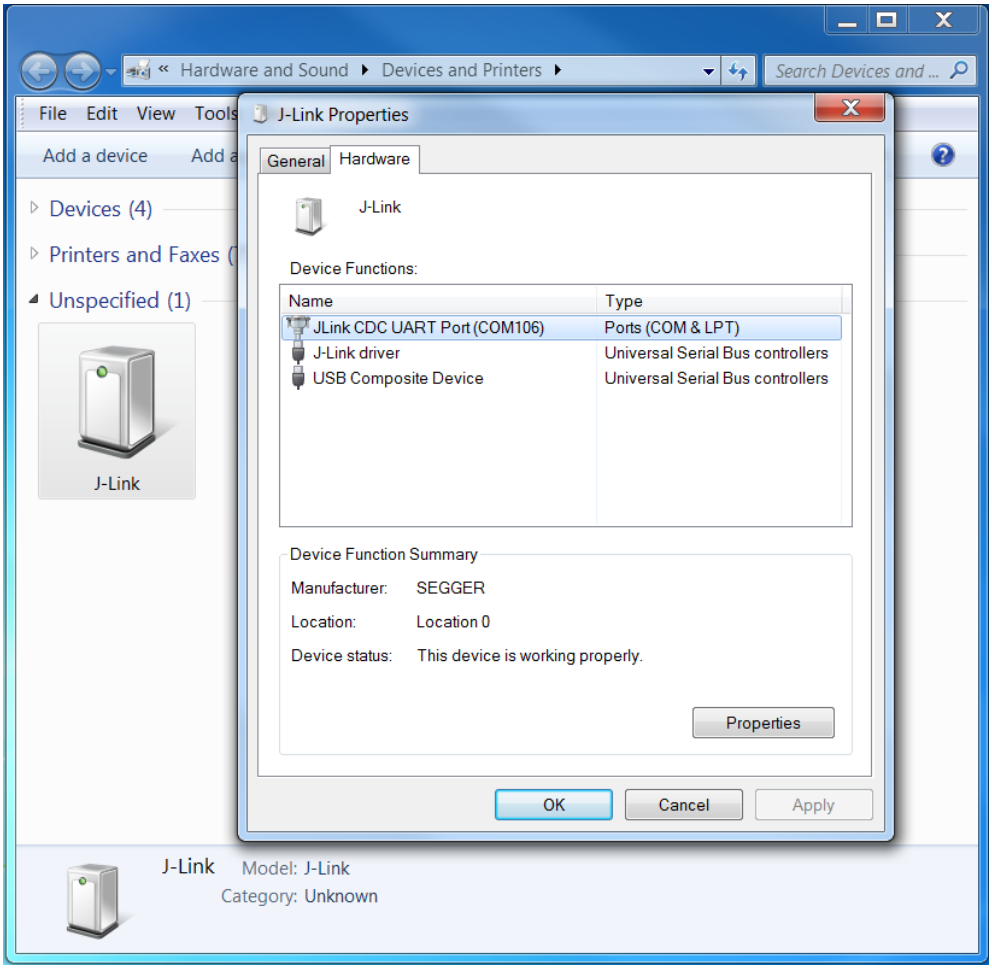


Figure 9: DA14585/586 DK - Basic Virtual COM Port

7.5.2 Connecting to a DA1458x DK-Pro

When the DA14585/586 DK [2] is connected via USB with a Windows machine (e.g. laptop) a Dual RS232-HS device should be discovered in the Windows Devices and Printers. In the Dual RS232-HS's properties window two USB Serial Ports are displayed. The user must select the serial port with the smaller number (Figure 10) to provide it to a terminal console application.

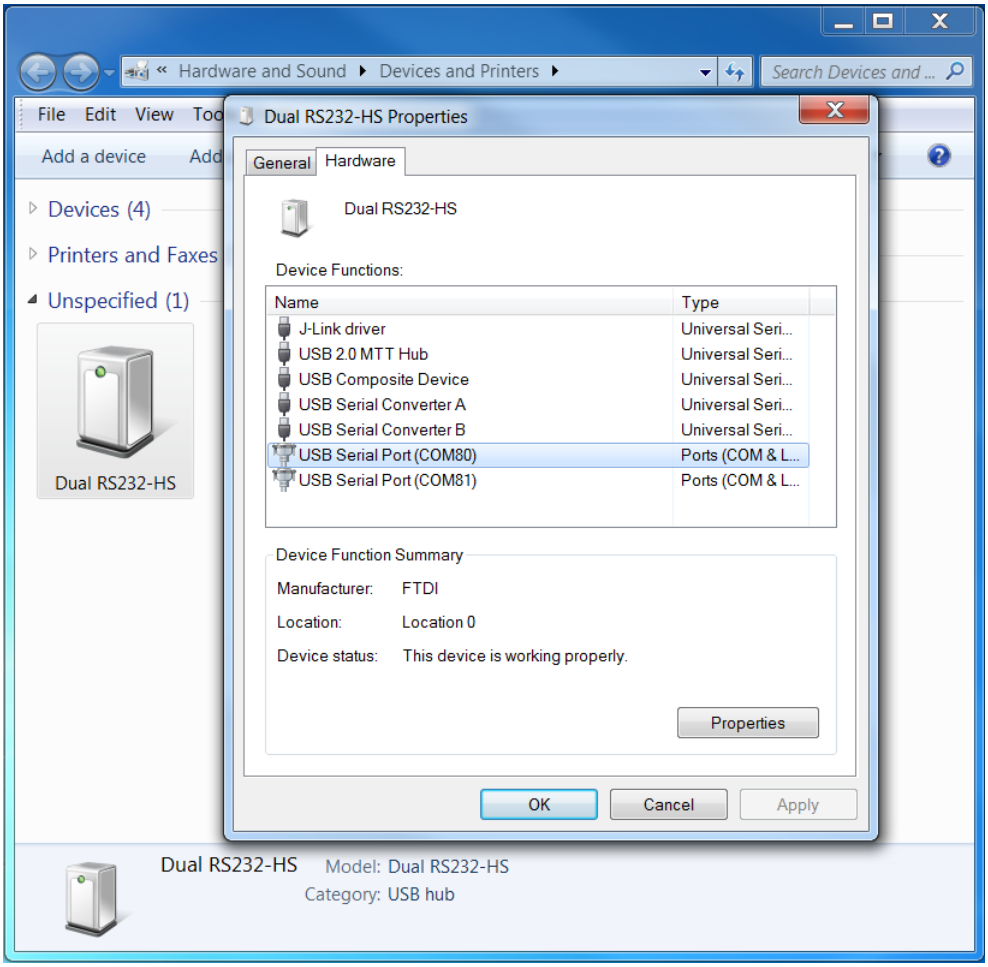


Figure 10: DA14585/586 DK-Pro Virtual COM Port

7.6 UART (Simple) Example

The simple UART example demonstrates how to configure, initiate, and send some characters synchronously to the UART interface.

The project is located in the `<sdk_root_directory>\projects\target_apps\peripheral_examples\uart` SDK directory.

The UART example is developed under the Keil v5 tool. The Keil project file is the:

`<sdk_root_directory>\projects\target_apps\peripheral_examples\uart\Keil_5\uart.uvprojx`

7.6.1 Hardware Configuration

The common UART terminal configuration described in section 7.4 is required.

Table 2: UART Example Jumper Configuration

GPIO	Function	DA14585/586 DK-Basic	DA14585/586 DK-Pro
P0_4	UART2 TX	Connect J4.11 - J4.12	Connect J5.11 - J5.12
P0_5	UART2 RX	Connect J4.13 - J4.14	Connect J5.13 - J5.14

7.6.2 Running the Example

Once the user has built and loaded the example project to the DK, the console will display the message shown in Figure 11. The uart_test function uses printf_byte and printf_string functions to print the message on the console.

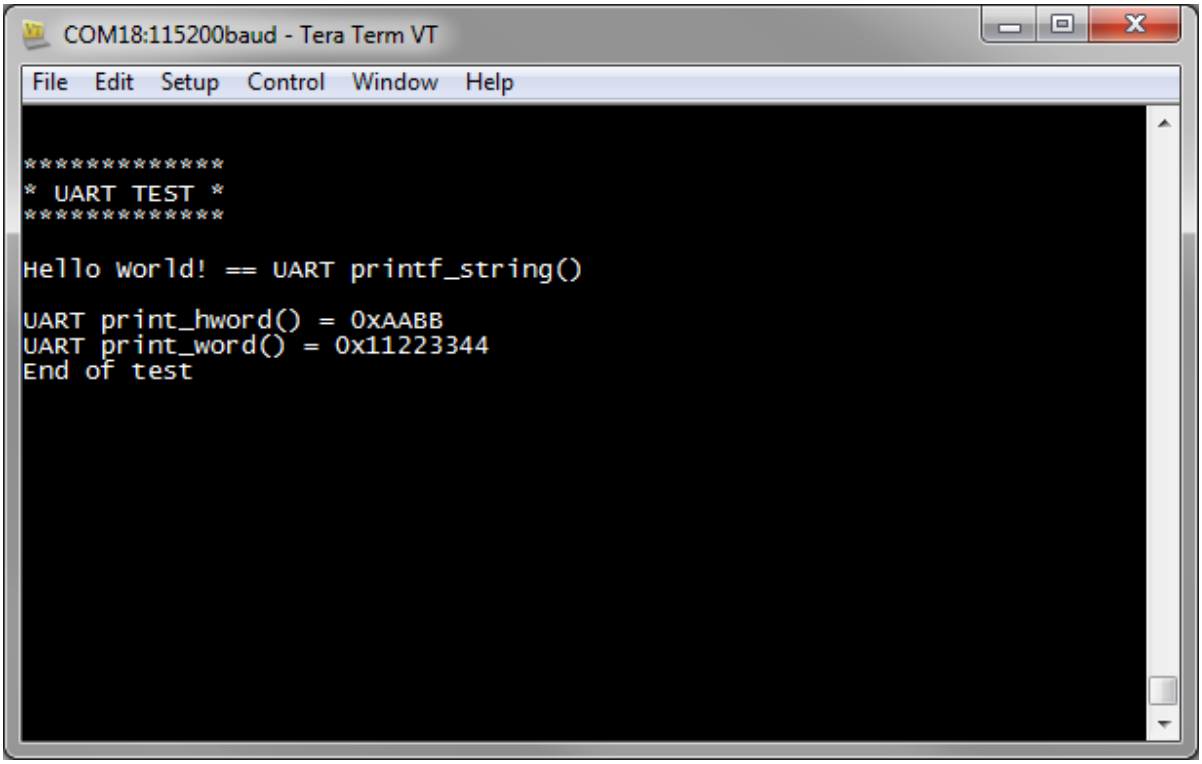


Figure 11: UART Simple Example

The user can select the UART settings in the header file:

<sdk_root_directory>\projects\target_apps\peripheral_examples\uart\include\user_periph_setup.h

The predefined settings are the following:

```
// Select UART settings
#define UART2_BAUDRATE      UART_BAUDRATE_115K2        // Baudrate in bits/s:
                                                        { 9K6, 14K4, 19K2, 28K8,
                                                         38K4, 57K6, 115K2}
#define UART2_FRAC_BAUDRATE UART_FRAC_BAUDRATE_115K2 // Baudrate fractional part:
                                                          {9K6, 14K4, 19K2, 28K8, 38K4,
                                                           57K6, 115K2}
#define UART2_DATALENGTH    UART_CHARFORMAT_8         // Datalength in bits:
                                                          {5, 6, 7, 8}
#define UART2_PARITY        UART_PARITY_NONE           // Parity: {UART_PARITY_NONE,
                                                                    UART_PARITY_EVEN,
                                                                    UART_PARITY_ODD}
#define UART2_STOPBITS      UART_STOPBITS_1            // Stop bits: {UART_STOPBITS_1,
                                                                      UART_STOPBITS_2}
#define UART2_FLOWCONTROL   UART_FLOWCONTROL_DISABLED // Flow control:
                                                        {UART_FLOWCONTROL_DISABLED,
                                                         UART_FLOWCONTROL_ENABLED}
```

The source code for this example can be found in function uart_test in:

projects\target_apps\peripheral_examples\uart\src\main.c.

7.7 UART2 Asynchronous Example

The UART2 asynchronous example demonstrates how to perform IRQ based IO operations using the UART2 driver (`<sdk_root_directory>\sdk\platform\driver\uart\uart2.c`).

The project is located in the `<sdk_root_directory>\projects\target_apps\peripheral_examples\uart2_async` SDK directory.

The UART2 asynchronous example is developed under the Keil v5 tool. The Keil project file is the:

`<sdk_root_directory>\projects\target_apps\peripheral_examples\uart2_async\Keil_5\uart2_async.uvprojx`

7.7.1 Hardware Configuration

The common UART terminal configuration described in section 7.4 is required.

Table 3: UART2 Example Jumper Configuration

GPIO	Function	DA14585/586 DK-Basic	DA14585/586 DK-Pro
P0_4	UART2 TX	Connect J4.11 - J4.12	Connect J5.11 - J5.12
P0_5	UART2 RX	Connect J4.13 - J4.14	Connect J5.13 - J5.14

7.7.2 Running the Example

Once the user has built and loaded the example project to the DK, it executes an asynchronous write test (in function uart2_write_test) and the following lines displayed in the terminal window.

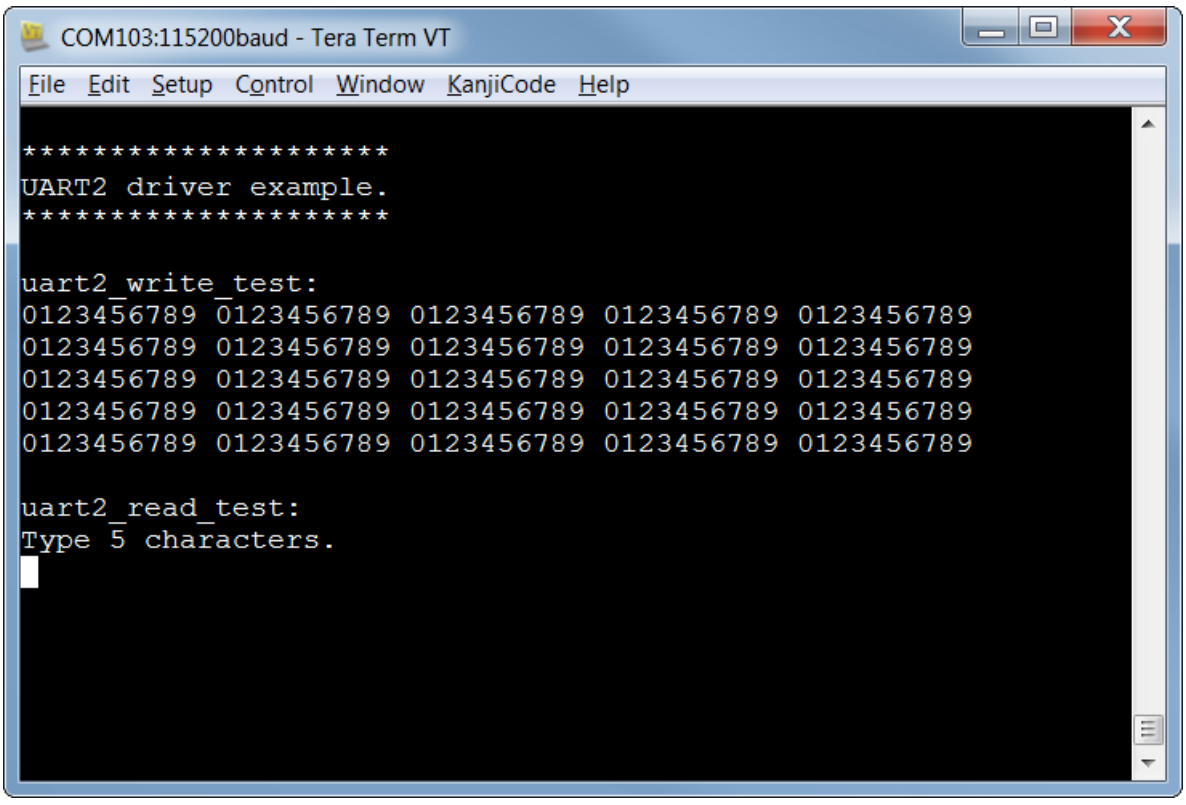


Figure 12: UART2 Example Console Output: Write Test

Next an asynchronous read test is executed (in function uart2_read_test) and the user is prompted to enter 5 characters. Assuming that the user types the five characters `a`, `b`, `c`, `d`, and `e`, then upon pressing `e` the typed characters are printed in the terminal window as shown in Figure 13.

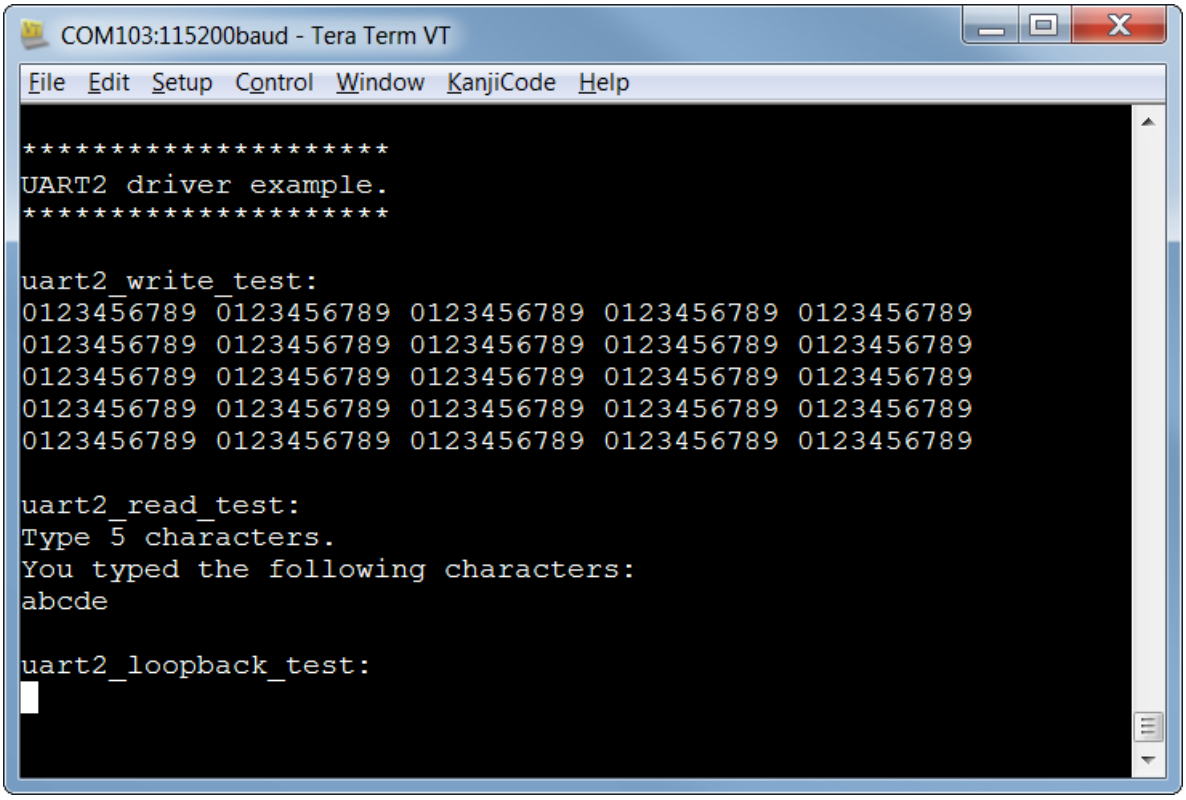


Figure 13: UART2 Example Console Output: Read Test

The last test executed is the loopback test (in function uart2_loopback_test) ,where every received character is echoed back to the sender. For example if the user types "Hello world!" then the following will be output (see Figure 14):

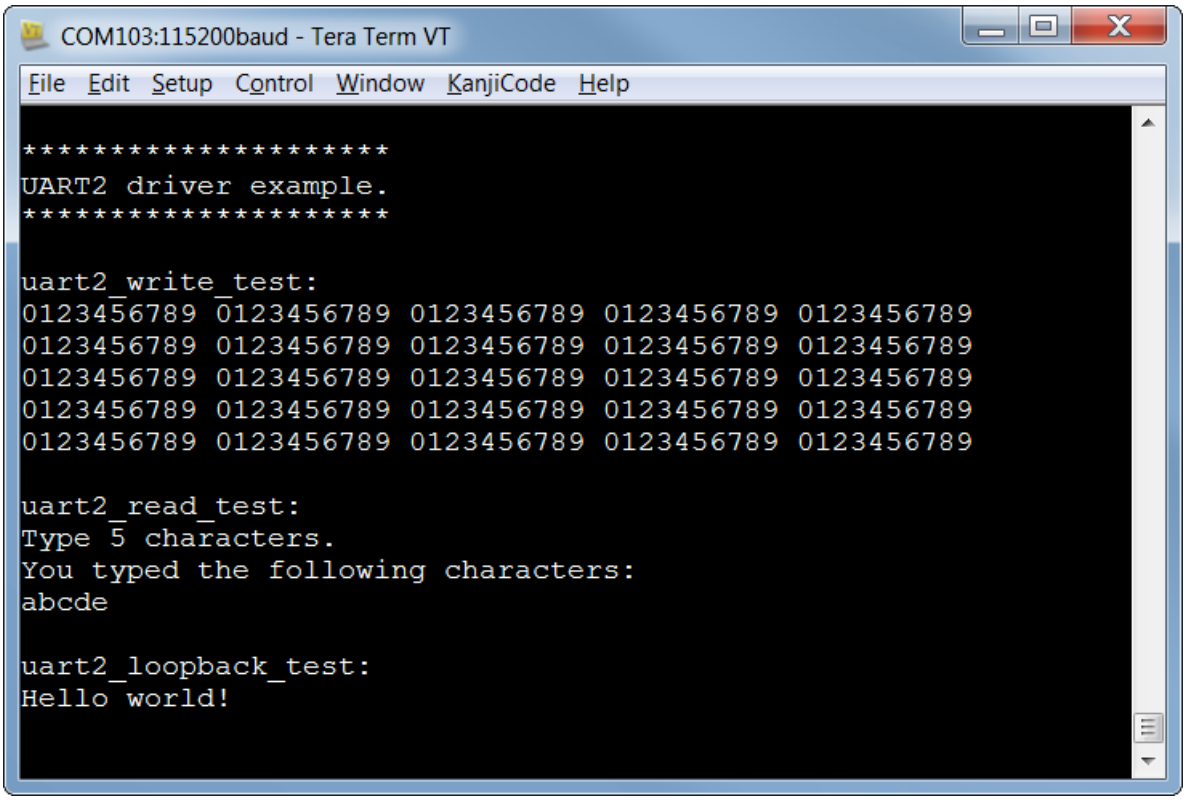


Figure 14: UART2 Example Console Output: Loopback Test

7.8 SPI Flash Memory Example

The SPI Flash memory example demonstrates how to initiate, read, write and erase an SPI Flash memory using the SPI Flash driver.

The project is located in the `<sdk_root_directory>\projects\target_apps\peripheral_examples\spi\spi_flash` SDK directory.

The SPI Flash memory example is developed under the Keil v5 tool. The Keil project file is the:

`<sdk_root_directory>\projects\target_apps\peripheral_examples\spi\spi_flash\Keil_5\spi_flash.uvprojx`

7.8.1 Hardware Configuration

The common UART terminal configuration described in section 7.4 is used but UART2 RX is left unconnected since this example does not require input from the terminal.

Table 4: SPI Flash Memory Example Jumper Configuration without UART2 RX

GPIO	Function	DA14585/586 DK-Basic	DA14585/586 DK-Pro
P0_0	SPI CLK	Connect J4.21 - J4.22	Connect J5.21 – J5.22
P0_3	SPI CS	Connect J4.19 - J4.20	Connect J5.19 – J5.20
P0_4	UART2 TX	Connect J4.11 - J4.12	Connect J5.11 - J5.12
P0_5	UART2 RX, SPI DI	Connect J6.1 - J4.13 (J4.14 is not connected)	Connect J6.1 - J5.13 (J5.14 is not connected)
P0_6	SPI DO	Connect J6.2 – J4.15	Connect J6.2 – J5.15

If reading from UART2 is necessary because of modifications made by the user then, since the UART2 RX default pin conflicts with the SPI MISO pin (P0_5), a separate pin will have to be used for UART2 RX. Assuming that P0_7 is used for UART2 RX then the header file `user_periph_setup.h` must be edited accordingly and following configuration can be used.

Table 5: SPI Flash Memory Example Jumper Configuration with UART2 RX

GPIO	Function	DA14585/586 DK-Basic	DA14585/586 DK-Pro
P0_0	SPI CLK	Connect J4.21 - J4.22	Connect J5.21 – J5.22
P0_3	SPI CS	Connect J4.19 - J4.20	Connect J5.19 – J5.20
P0_4	UART2 TX	Connect J4.11 - J4.12	Connect J5.11 - J5.12
P0_5	SPI DI	Connect J6.1 - J4.13	Connect J6.1 - J5.13
P0_6	SPI DO	Connect J6.2 – J4.15	Connect J6.2 – J5.15
P0_7	UART2 RX	Connect J4.17 - J4.14 (with a jumper wire)	Connect J5.17 - J5.14 (with a jumper wire)

7.8.2 Running the Example

Once the user has built and loaded the example project to the DK, a series of read and write operations will be performed on the SPI Flash memory (as shown in Figure 15).

The user also has to enter the characteristics of the SPI Flash in the header file:

```
<sdk_root_directory>\projects\target_apps\peripheral_examples\spi\spi_flash\include\user_periph_setup.h
```

The predefined characteristics are the following:

```
#define SPI_FLASH_SIZE   131072          // SPI Flash memory size in bytes
#define SPI_FLASH_PAGE   256             // SPI Flash memory page size in bytes
```

The user can also select the SPI module parameters, such as word mode, polarity, phase and frequency:

```
#define SPI_WORD_MODE SPI_8BIT_MODE     // Select SPI bit mode
#define SPI_SMN_MODE  SPI_MASTER_MODE   // {SPI_MASTER_MODE, SPI_SLAVE_MODE}
#define SPI_POL_MODE  SPI_CLK_INIT_HIGH // {SPI_CLK_INIT_LOW, SPI_CLK_INIT_HIGH}
#define SPI_PHA_MODE  SPI_PHASE_1       // {SPI_PHA_MODE_0, SPI_PHA_MODE_1}
#define SPI_MINT_EN   SPI_NO_MINT       // {SPI_MINT_DISABLE, SPI_MINT_ENABLE}
#define SPI_CLK_DIV   SPI_XTAL_DIV_2    // Select SPI clock divider between
                                        // 8, 4, 2 and 14
```

The spi_test function performs the following tests:

1. The GPIO pins used for the SPI Flash and the SPI module are initialized.
2. The SPI Flash device memory is erased. To erase a device on which protection has been activated, the full test has to be run once (see step 11a).
3. The contents of the SPI Flash are read and printed to the console.
4. The JEDEC ID is read and the device is detected, if a corresponding entry is found in the supported devices list (UM-B-004, DA14580 Peripheral drivers, Dialog Semiconductor).
5. The Manufacturer/Device ID and the Unique ID are read, if the device is known to support it.
6. 256 bytes of data is written to the SPI Flash using the Program Page instruction.
7. The contents of the SPI Flash are read and printed out on the console.
8. A sector of the SPI Flash is erased.
9. 512 bytes of data is written to the SPI Flash using the spi_write_data function, which is used for writing data longer than the SPI Flash page.
10. The 512 bytes of data are read back and printed to the console.
11. If supported by the SPI Flash memory device, a suitable test for the device's memory protection capabilities is performed:
    a. The whole memory area is configured as unprotected and a full memory array erase is performed.
    b. The memory is tested in various configurations to demonstrate the effect the memory protection of the various blocks of memory has on data storage and how previously stored data is affected by overwrite.
    c. Finally, once again the whole memory area is configured as unprotected and a full memory array erase is performed.

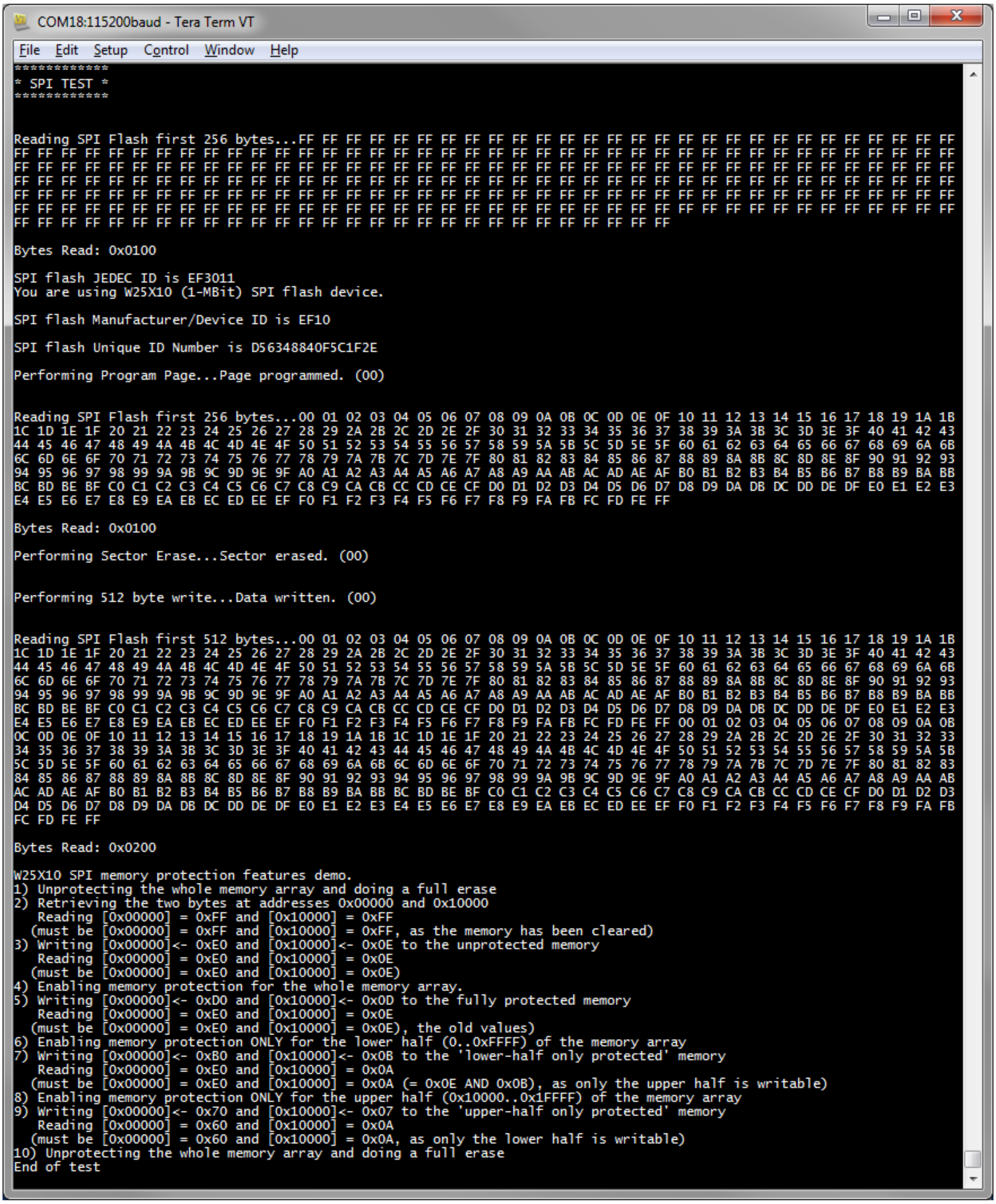


Figure 15: SPI Flash Memory Example

In case you use a supported external SPI Flash device other than W25X10, the results of the SPI memory protection features test will differ.

The user can change the test procedure by editing the function spi_test. The SPI Flash driver API is described in detail in Ref. [5].

The source code for this example can be found in function spi_test inside:

<sdk_root_directory>\projects\target_apps\peripheral_examples\spi\spi_flash\src\main.c
.

7.9 I2C EEPROM Example

The I2C EEPROM example demonstrates how to initiate, read, write and erase an I2C EEPROM memory.

The project is located in the `<sdk_root_directory>\projects\target_apps\peripheral_examples\i2c\i2c_eeprom` SDK directory.

The I2C EEPROM example is developed under the Keil v5 tool. The Keil project file is the:

`<sdk_root_directory>\projects/target_apps/peripheral_examples/i2c/i2c_eeprom/Keil_5/i2c_eeprom.uvprojx`

7.9.1 Hardware Configuration

The common UART terminal configuration described in section 7.4 is required.

Table 6: I2C EEPROM Example Jumper Settings

GPIO	Function	DA14585/586 DK-Basic	DA14585/586 DK-Pro
P0_2	I2C SCL		
P0_3	I2C SDA		
P0_4	UART2 TX	Connect J4.11 - J4.12	Connect J5.11 - J5.12
P0_5	UART2 RX	Connect J4.13 - J4.14	Connect J5.13 - J5.14

An external I2C EEPROM must be connected to the DA14585/586 DK using the pins shown in Table 6: P0_2 for SCL and P0_3 for SDA. If a different selection of GPIO pins is needed, the user should edit the `I2C_GPIO_PORT`, `I2C_SCL_PIN` and `I2C_SDA_PIN` defines in `user_periph_setup.h`.

7.9.2 Running the Example

Once the user has built and loaded the example project to the DK, a series of read and writes operations will be performed on the I2C EEPROM, as shown in Figure 16.

The predefined I2C EEPROM characteristics are the following:

```
#define I2C_EEPROM_SIZE     0x20000     // EEPROM size in bytes
#define I2C_EEPROM_PAGE     256         // EEPROM's page size in bytes
```

The user can also select the I2C module's parameters, such as slave address, speed mode, address mode and addressing scheme (1-byte/2-byte):

```
#define I2C_SLAVE_ADDRESS   0x50            // Set slave device address
#define I2C_SPEED_MODE      I2C_FAST        // Speed mode: I2C_STANDARD (100 kbits/s),
                                                          I2C_FAST:    (400 kbits/s)
#define I2C_ADDRESS_MODE    I2C_7BIT_ADDR   // Addressing mode: {I2C_7BIT_ADDR,
                                                                 I2C_10BIT_ADDR}
#define I2C_ADDRESS_SIZE    I2C_2BYTES_ADD  // Address width: {I2C_1BYTE_ADDR,
                                                              I2C_2BYTES_ADDR,
                                                              I2C_2BYTES_ADDR}
```

The i2c_test function performs the following tests:

- Initializes the GPIO pins used for the I2C EEPROM and the I2C module.
- Writes 256 bytes of data to the I2C EEPROM.
- Reads the contents of the I2C EEPROM.
- Writes and reads bytes 0x5A, 0x6A, 0x7A and 0xFF at addresses 22, 0, 255 and 30 respectively.
- Reads the contents of the I2C EEPROM.
- Releases the configured GPIO pins and the I2C module.
- This is shown in detail in Figure 16.

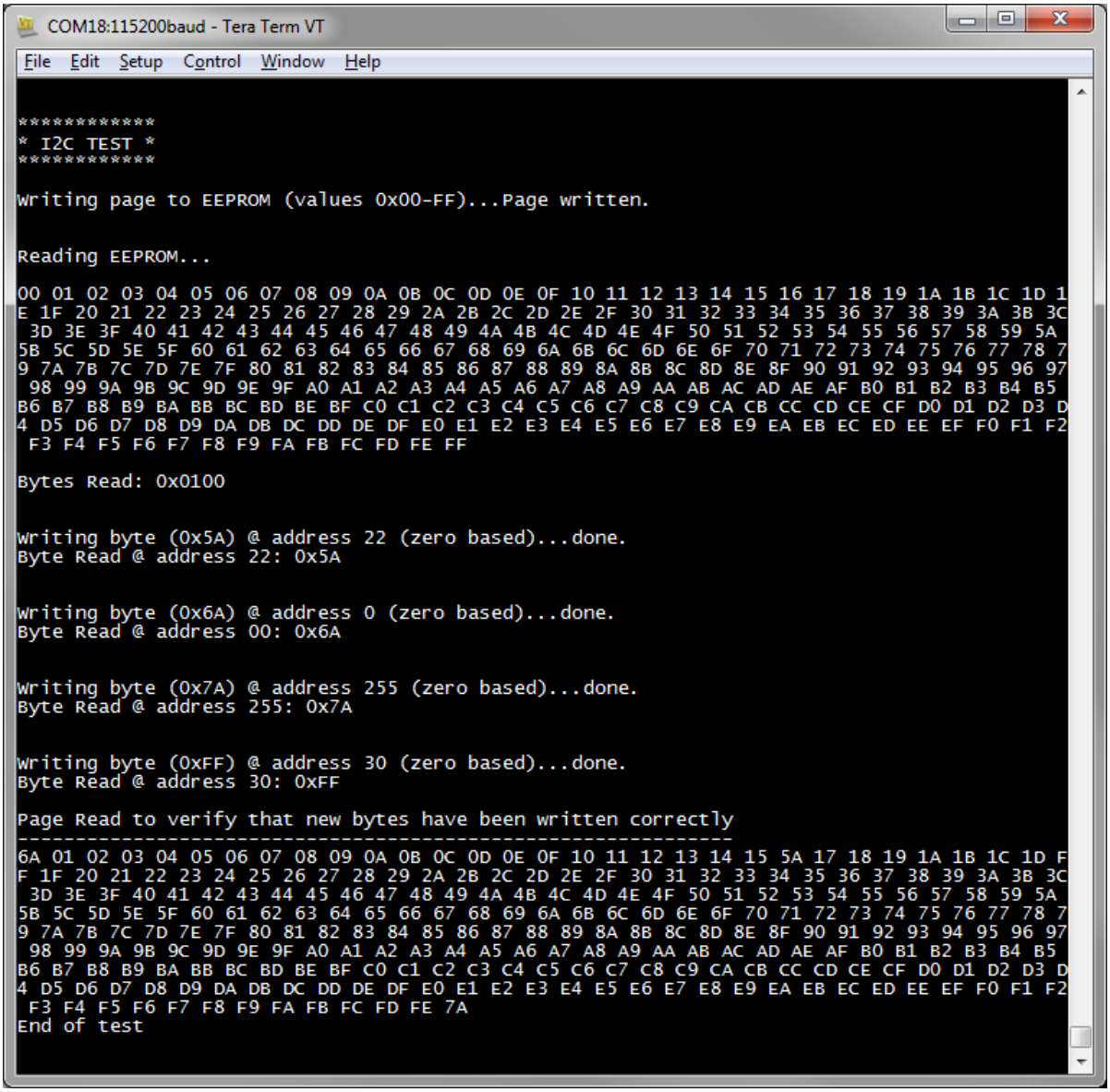


Figure 16: I2C EEPROM Example

The user can change the test procedure by editing the function i2c_test. The I2C EEPROM driver API is described in detail in Ref. [5].

The source code for this example is located in function `i2c_test` inside:

`<sdk_root_directory>\projects\target_apps\peripheral_examples\i2c\i2c_eeprom\src\main.c`.

7.10 Quadrature Decoder Example

The Quadrature decoder example demonstrates how to configure and read from the quadrature decoder peripheral. The Wakeup Timer setup for responding to GPIO activity is also demonstrated in this example.

The project is located in the `<sdk_root_directory>\projects\target_apps\peripheral_examples\quadrature_decoder` SDK directory.

The Quadrature decoder example is developed under the Keil v5 tool. The Keil project file is the:

`…\quadrature_decoder\Keil_5\quadrature_decoder.uvprojx`

7.10.1 Hardware Configuration

The common UART terminal configuration described in section 7.4 is required.

Table 7: Quadrature Decoder Example Jumper Settings

GPIO	Function	DA14585/586 DK-Basic	DA14585/586 DK-Pro
P0_0	CHX_A		
P0_1	CHX_A		
P0_4	UART2 TX	Connect J4.11 - J4.12	Connect J5.11 - J5.12
P0_5	UART2 RX	Connect J4.13 - J4.14	Connect J5.13 - J5.14
P0_6	K1 BUTTON	Not available	SW2
P1_1	K2 BUTTON	Not available	SW3

The quadrature encoder CHX_A and CHX_B pins have to be connected to P0_0 and P0_1 respectively. The common terminal of the quadrature decoder must be connected to ground. To enable the K1 and K2 buttons functionality on a DK-Basic, the user has to connect external HW to P0_6 and P1_1.

7.10.2 Running the Example

Once the user has built and loaded the example project to the DK, the console will display the screen shown in Figure 17.

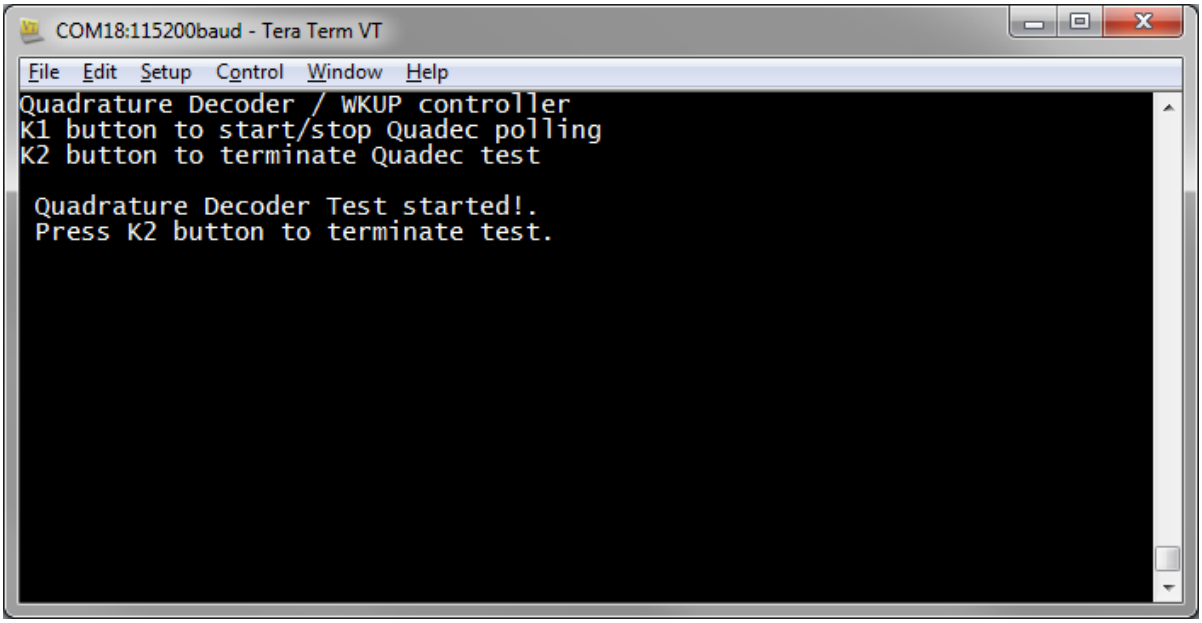


Figure 17: Quadrature Decoder Example

If the rotary decoder is activated (e.g. the mouse wheel is scrolled and the rotary decoder is attached to a mouse) the quadrature decoder-wakeup timer interrupt will be triggered, and after each trigger the terminal screen will report the axes relative coordinates. In this configuration, only the X channel terminals are configured and connected (see Figure 18).

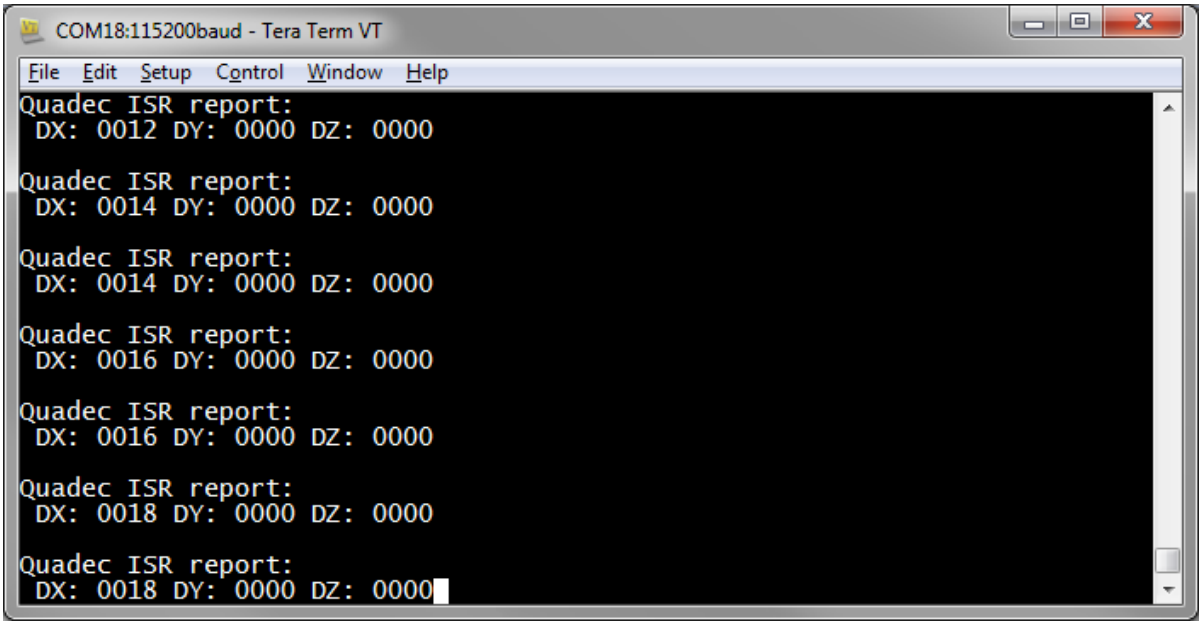


Figure 18: Quadrature Decoder ISR-Only Reports

If at any time the K1 button is pressed (the user should make sure that the correct jumper configuration for buttons K1 and K2 is selected, as described in Ref. [2]), polling of the relative coordinates will be enabled. Then the terminal window will start polling the quadrature decoder driver (see Figure 19). If the quadrature decoder is activated, a mixture of ISR and polling generated reports are displayed (see Figure 20).

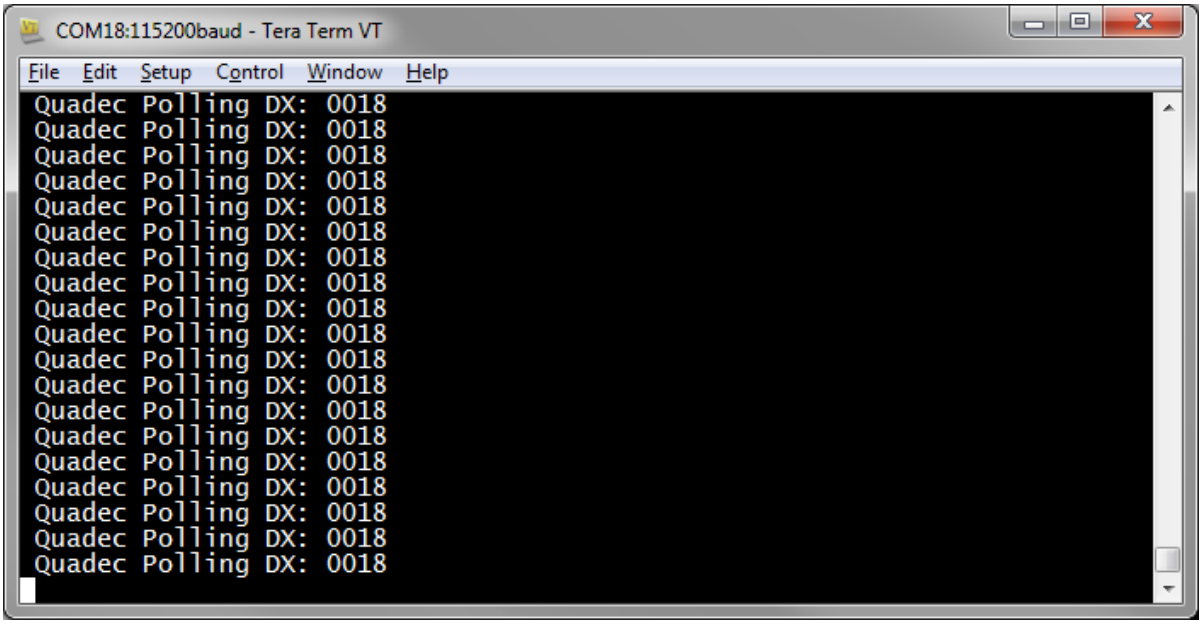


Figure 19: Quadrature Decoder Polling-Only Reports

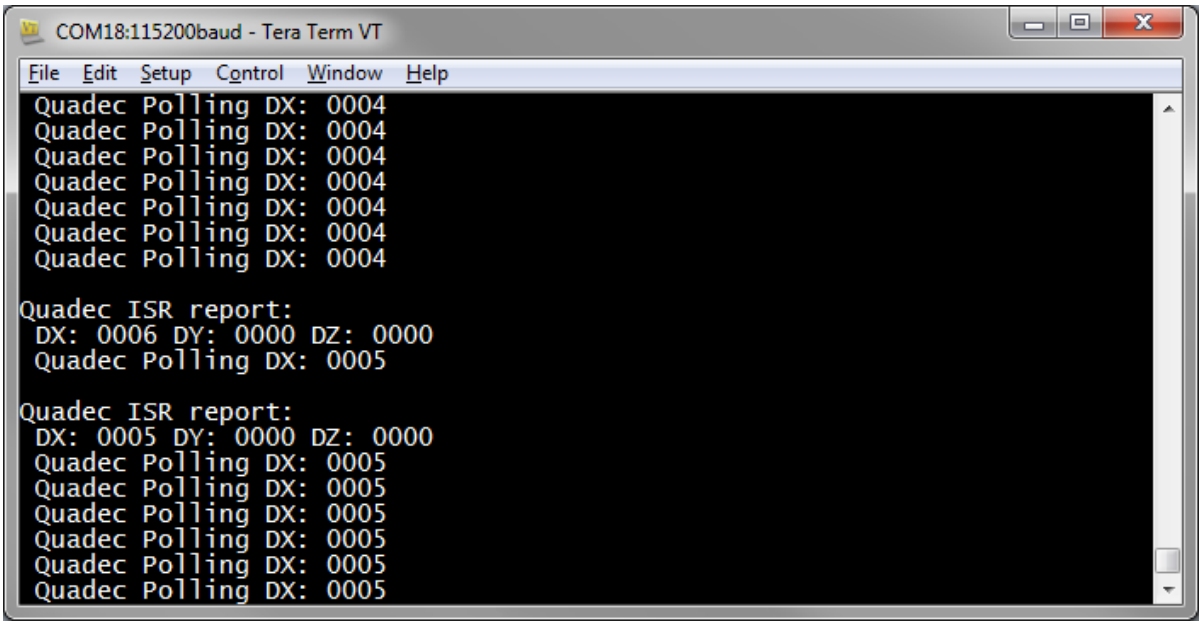


Figure 20: Quadrature Decoder Polling and ISR Reports

The polling can be stopped by pressing K1 button again. To terminate the test, one can press K2 at any time. The message "Quadrature Decoder Test terminated!" will be printed.

If the count of events needs to be changed before an interrupt is triggered, the parameter `QDEC_EVENTS_COUNT_TO_INT` in `user_periph_setup.h` can be modified accordingly. In the same file, one can change the clock divisor of the quadrature decoder by altering the parameter `QDEC_CLOCK_DIVIDER`.

The source code for this example is located in function quad_decoder_test inside:

`<sdk_root_directory>\projects\target_apps\peripheral_examples\quadrature_decoder\src\main.c`.

7.11 Systick Example

The Systick example demonstrates how to use the systick timer to generate an interrupt periodically. LED is changing its state upon each interrupt.

The project is located in the `<sdk_root_directory>\projects\target_apps\peripheral_examples\systick` SDK directory.

The Systick example is developed under the Keil v5 tool. The Keil project file is the:

`<sdk_root_directory>\projects\target_apps\peripheral_examples\systick\Keil_5\systick.uvprojx`

7.11.1 Hardware Configuration

This example does not use the UART terminal.

Table 8. Systick Example Jumper Settings

GPIO	Function	DA14585/586 DK-Basic	DA14585/586 DK-Pro
P1_0	LED	Connect J9.1 – J9.2	Connect J9.1 – J9.2

7.11.2 Running the Example

Once the user has built and loaded the example project to the DK, the LED will start to blink in a 1 s rhythm. The source code for this example is located in function systick_test inside:

`<sdk_root_directory>\projects\target_apps\peripheral_examples\systick\src\main.c.`

7.12 TIMER0 (PWM0, PWM1) Example

The TIMER0 (PWM0, PWM1) example demonstrates how to configure TIMER0 to produce PWM signals. A melody is produced on an externally connected buzzer.

The project is located in the `<sdk_root_directory>\projects\target_apps\peripheral_examples\timer0\timer0_pwm` SDK directory.

The TIMER0 (PWM0, PWM1) example is developed under the Keil v5 tool. The Keil project file is the:

`<sdk_root_directory>\projects\target_apps\peripheral_examples\timer0\timer0_pwm\Keil_5\timer0_pwm.uvprojx`

7.12.1 Hardware Configuration

The common UART terminal configuration described in section 7.4 is required.

Table 9: Timer0 Example Jumper Settings

GPIO	Function	DA14585/586 DK-Basic	DA14585/586 DK-Pro
P0_2	PWM0		
P0_3	PWM1		
P0_4	UART2 TX	Connect J4.11 - J4.12	Connect J5.11 - J5.12
P0_5	UART2 RX	Connect J4.13 - J4.14	Connect J5.13 - J5.14

In order to have an audio indication of the produced PWM signals, the user can connect a buzzer in P0_2 and ground (PWM0) or in P0_3 and ground (PWM1).

7.12.2 Running the Example

Once the user has built and loaded the example project to the DK, the PWM0 and PWM1 signals will become active and start producing an audible melody if a buzzer is connected to P0_2 or P0_3. While the melody is playing, stars are being drawn in the terminal window on each beat (interrupt handling - Figure 21).

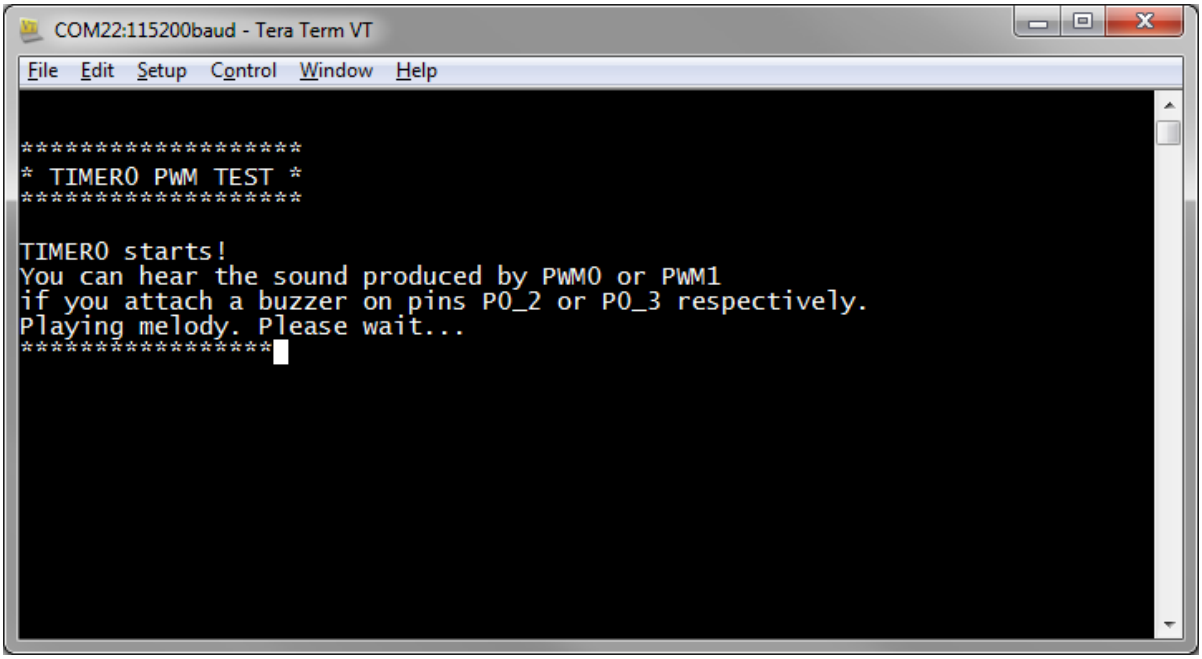


Figure 21: TIMER0 (PWM0, PWM1) Test Running

Upon completion, the following message will appear:

```
"Timer0 stopped
End of test"
```

The source code for this example is located in function timer0_test inside:

```
<sdk_root_directory>\projects\target_apps\peripheral_examples\timer0\timer0_pwm\src\main.c.
```

7.13 TIMER0 General Example

The TIMER0 general example demonstrates how to configure TIMER0 to count a specified amount of time and generate an interrupt. A LED is changing state upon each timer interrupt.

The project is located in the `<sdk_root_directory>\projects\target_apps\peripheral_examples\timer0\timer0_general` SDK directory.

The TIMER0 general example is developed under the Keil v5 tool. The Keil project file is the:

```
<sdk_root_directory>\projects\target_apps\peripheral_examples\timer0\timer0_genera\Keil_5\timer0_general.uvprojx
```

7.13.1 Hardware Configuration

The common UART terminal configuration described in section 7.4 is required.

Table 10: Timer0 General Example Jumper Settings

GPIO	Function	DA14585/586 DK-Basic	DA14585/586 DK-Pro
P0_4	UART2 TX	Connect J4.11 - J4.12	Connect J5.11 - J5.12

GPIO	Function	DA14585/586 DK-Basic	DA14585/586 DK-Pro
P0_5	UART2 RX	Connect J4.13 - J4.14	Connect J5.13 - J5.14
P1_0	LED	Connect J9.1 – J9.2	Connect J9.1 – J9.2

7.13.2 Running the Example

Once the user has built and loaded the example project to the DK, the LED will be changing its state every second until the end of the test. The duration of the test (expressed in seconds) can be set by user and it will also be printed in the console. See Figure 22.

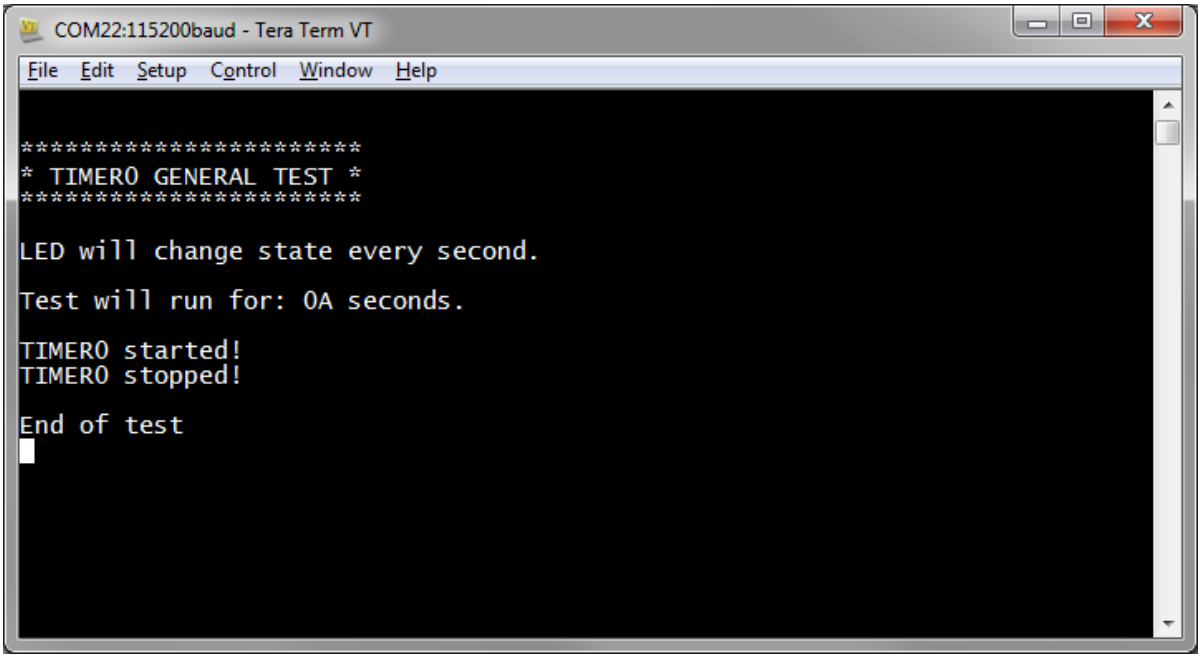


Figure 22: TIMER0 General Test Completed

The source code for this example is located in function timer0_general_test in the file:

`<sdk_root_directory>\projects\target_apps\peripheral_examples\timer0\timer0_general\src\main.c.`

7.14 TIMER2 (PWM2, PWM3, PWM4) Example

The TIMER2 (PWM2, PWM3, PWM4) example demonstrates how to configure TIMER2 to produce PWM signals. LEDs are changing light brightness in this example.

The project is located in the `<sdk_root_directory>\projects\target_apps\peripheral_examples\timer2\timer2_pwm` SDK directory.

The TIMER2 (PWM2, PWM3, PWM4) example is developed under the Keil v5 tool. The Keil project file is the:

`<sdk_root_directory>\projects\target_apps\peripheral_examples\timer2\timer2_pwm\Keil_5\timer2_pwm.uvprojx`

7.14.1 Hardware Configuration

The common UART terminal configuration described in 7.4 is required.

Table 11: TIMER2 Example Jumper Settings

GPIO	Function	DA14585/586 DK-Basic	DA14585/586 DK-Pro
P0_4	UART2 TX	Connect J4.11 - J4.12	Connect J5.11 - J5.12
P0_5	UART2 RX	Connect J4.13 - J4.14	Connect J5.13 - J5.14
P1_2	PWM2		
P0_7	PWM3		
P1_0	PWM4		

To use the LED segments inside D1 and D2 as a visual indication for signals PWM2, PWM3 and PWM4, the user has to connect PIN3 to PIN4 on connector J16 (PWM3), PIN1 to PIN2 on connector J16 (PWM4) and PIN3 to PIN4 on connector J15 (PWM2). The brightness of the LED segments is then directly influenced by the duty cycle of the PWM signals.

7.14.2 Running the Example

Once the user has built the Timer2 (PWM2, PWM3, PWM4) example and loaded to DK, the PWM2, PWM3 and PWM4 signals will become active. If the jumper configuration described in section 7.14.1 has been selected, there will be a visible indication of the PWM3 and PWM4 signals on segments of the D2 and D1 LED segments, as their brightness will be directly influenced by the PWM signals' duty cycle. The brightness will be changing automatically because each PWM duty cycle will chang in a loop function. The screen shown in Figure 23 will appear.

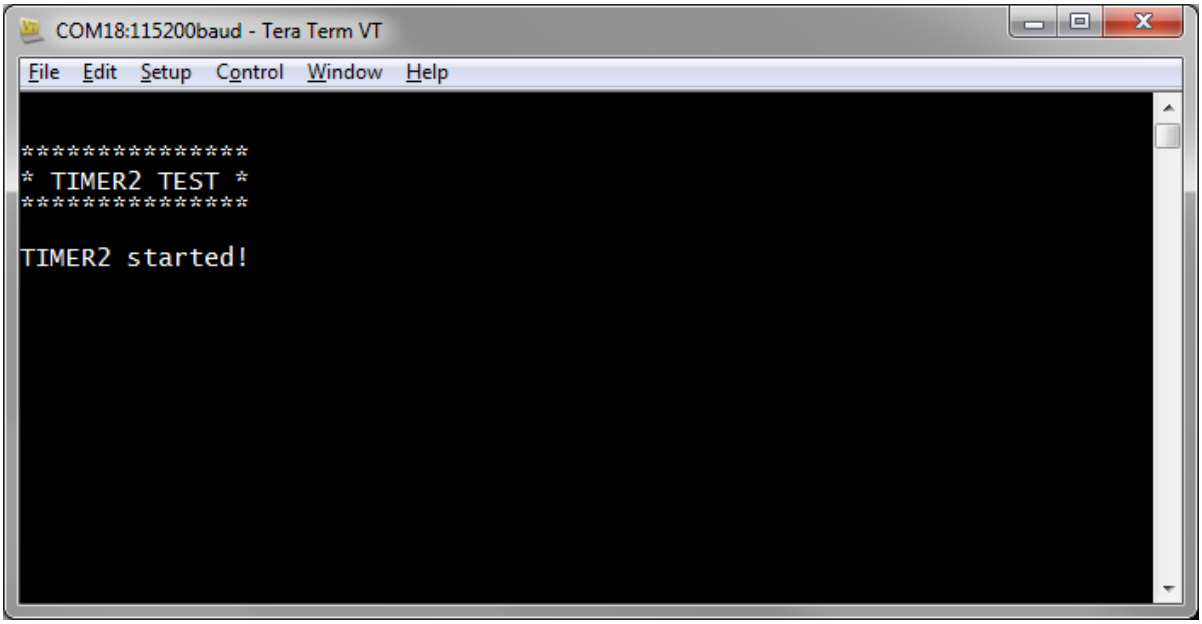


Figure 23: TIMER2 (PWM2, PWM3, PWM4) Test Running

After the test has been completed, the screen shown in Figure 24 will then appear.

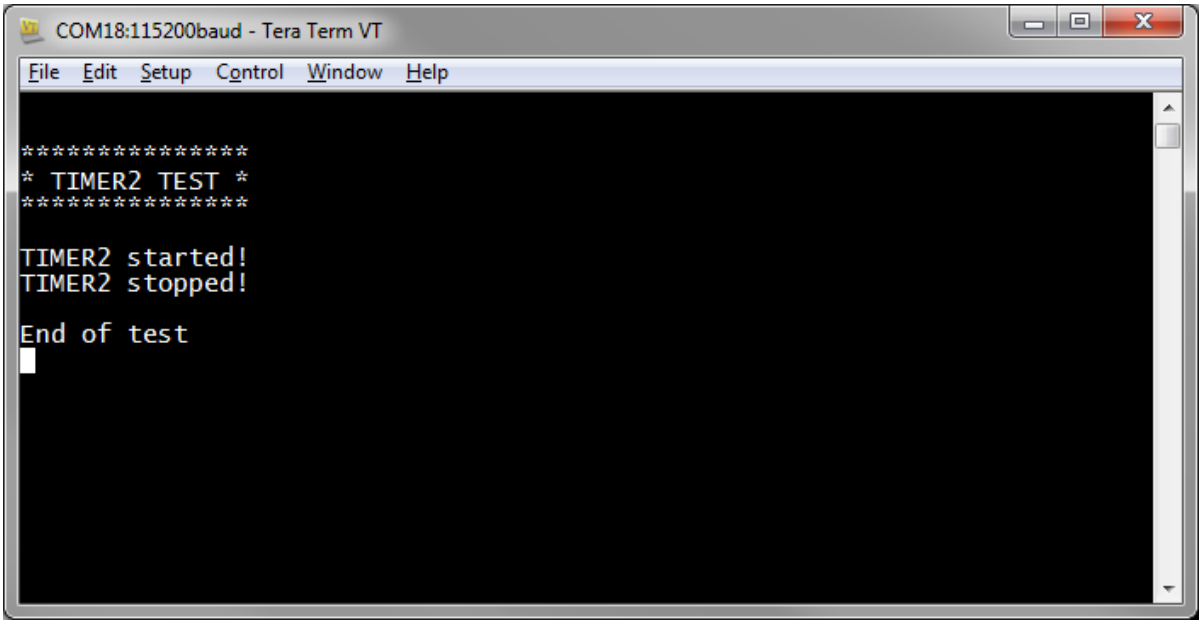


Figure 24: TIMER2 (PWM2, PWM3, PWM4) Test Completed

The source code for this example is located in function `timer2_test` inside:

`<sdk_root_directory>\projects\target_apps\peripheral_examples\timer2\timer2_pwm\src\main.c`.

7.15 Battery Example

The Battery example demonstrates how to read the battery level using the ADC.

The project is located in the `<sdk_root_directory>\projects\target_apps\peripheral_examples\adc\batt_lvl` SDK directory. The Battery example is developed under the Keil v5 tool. The Keil project file is the:

`<sdk_root_directory>\projects\target_apps\peripheral_examples\adc\batt_lvl\Keil_5\batt_lvl.uvprojx`

7.15.1 Hardware Configuration

The common UART terminal configuration described in section 7.4 is required.

Table 12: Battery Example Jumper Settings

GPIO	Function	DA14585/586 DK-Basic	DA14585/586 DK-Pro
P0_4	UART2 TX	Connect J4.11 - J4.12	Connect J5.11 - J5.12
P0_5	UART2 RX	Connect J4.13 - J4.14	Connect J5.13 - J5.14

Refer to [1] and [2] for details on how to set up a DA14585/586 DK for CR2032 battery operation.

7.15.2 Running the Example

Once the user has built and loaded the example project to the DK, the ADC is configured to provide a measurement of the battery level. The percentage left indication calculated for the coin-cell battery CR2032 is displayed on the terminal screen (see Figure 25).

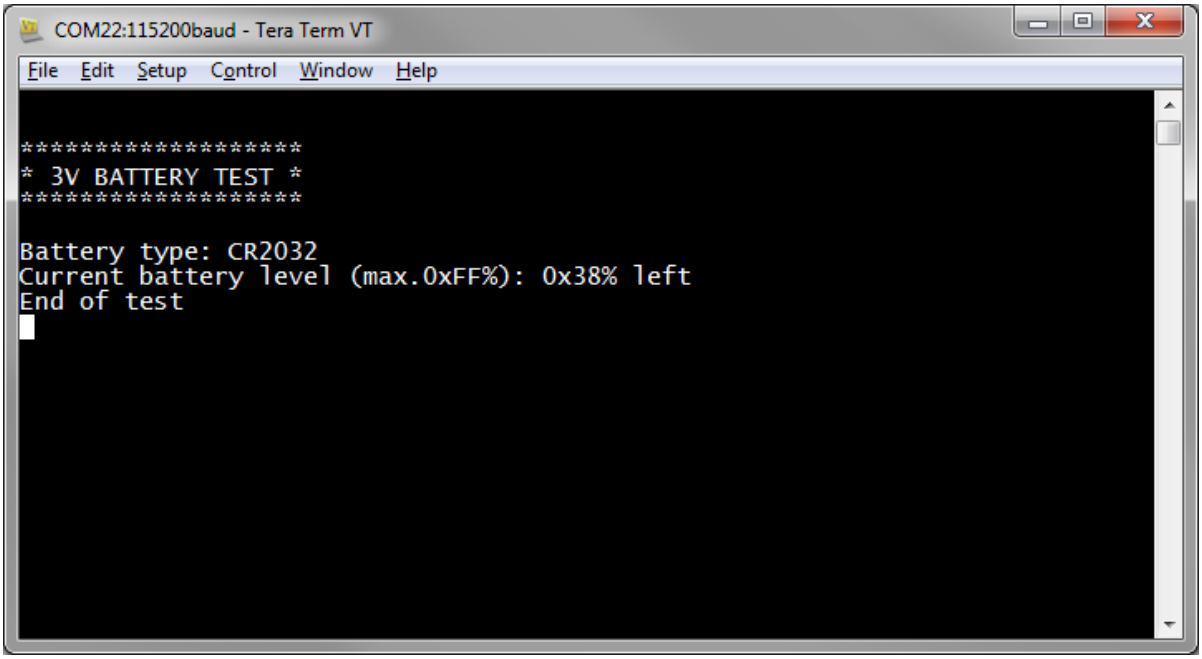


Figure 25: Battery Example

This example can be verified using an external voltage source (well-stabilized in the range 2.5 V to 3 V) instead of a 3 V battery. The DK must have been configured for 3 V operation.

⚠ CAUTION
The voltage of the external voltage source must never exceed 3 V.

For details about the configuration of the ADC, one should consult [5].

The source code for this example is located in function batt_test in:

<sdk_root_directory>\projects\target_apps\peripheral_examples\adc\batt_lvl\src\main.c.

8 Developing Bluetooth Low Energy Applications

8.1 The Seven Pillar Example Applications

The DA14585/586 SDK 6.x includes seven pillar BLE example application projects, to assist and guide on the development of Bluetooth Low Energy applications. Every pillar project inherits the functionality of a previous pillar project and adds also extra features. The base pillar project is the Pillar 1 (bare bone), as it is depicted in Figure 26. The goal of the pillar example projects is to provide, in a step-by-step approach, an introduction and explanation of the BLE features and functionality as supported by the DA14585/586 software platform and devices.

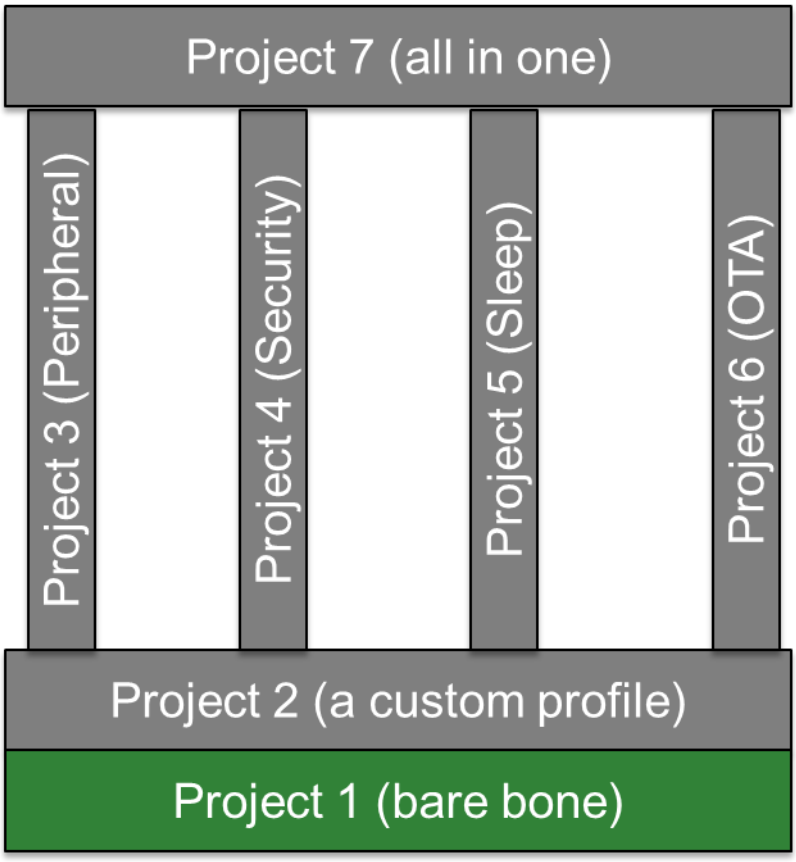


Figure 26: Pillar Example Projects

8.2 Pillar 1 (Bare Bone)

8.2.1 Application Description

The Pillar 1 (bare bone) BLE example application demonstrates basic BLE procedures such as advertising, connection, connection parameters update and implementation of the Device Information Service Server (DISS). The application uses the "Integrated processor" configuration.

8.2.2 Basic Operation

Supported services:

- Device Information service (UUID 0x180A).

Features:

- Supports Extended Sleep mode
- Basic Configuration Settings:
  - Advertising interval
  - Connection interval.
  - Slave latency
  - Supervision timeout
- Advertising data:
  - Device name.
  - Device information service support.
  - Manufacturer specific data (for advertising data update feature):
  - - Company identifier
  - - Proprietary data: 16-bit counter
- Advertising data is updated on the fly every 10s as follows:
  - Change proprietary data (increment by 1 the 16-bit counter)
  - Restart advertising update timer

The Pillar 1 application behavior is included in C source file `user_barebone.c`.

8.2.3 User Interface

A peer connected to Pillar 1 application is able to.

- Check the advertising device name.
- Check that the advertising data 16-bit counter is incremented in every advertising event or when the respective timer expires.
- Use the Device information service.

8.2.4 Loading the Project

The Pillar 1 application is developed under the Keil uVision v5tool. The Keil project file is the:

`<sdk_root_directory>\projects\target_apps\ble_examples\ble_app_barebone\Keil_5\ble_app_barebone.uvprojx`.

Figure 27 shows the Keil project layout with emphasis on the user related files, included in the Keil project folders `user_config`, `user_platform` and `user_app`. These folders contain the user configuration files of the Pillar 1 application.

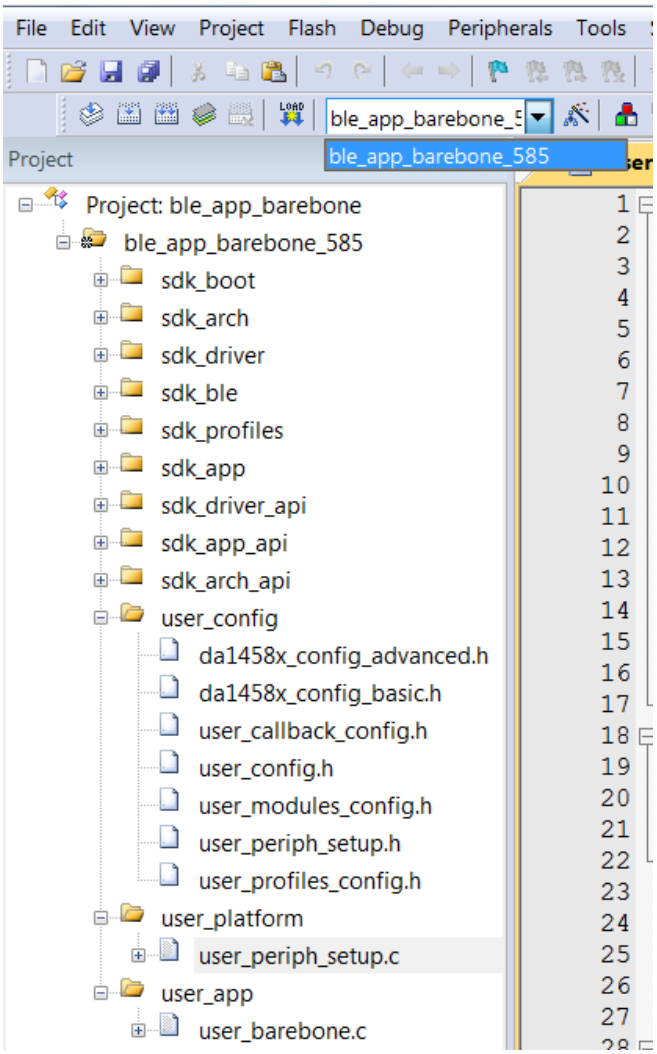


Figure 27: Pillar 1 Keil Project Layout

8.2.5 Going Through the Code

8.2.5.1 Initialization

The aforementioned Keil project folders (`user_config`, `user_platform` and `user_app`) contain the files that initialize and configure the Pillar 1 application.

- `da1458x_config_advanced.h`, holds DA14585/586 advanced configuration settings.
- `da1458x_config_basic.h`, holds DA14585/586 basic configuration settings.
- `user_callback_config.h`, callback functions that handle various events or operations.
- `user_config.h`, holds advertising parameters, connection parameters, etc.
- `user_modules_config.h`, defines which application modules are included or excluded from the user's application. For example:
  - `#define EXCLUDE_DLG_DISS (0)`, the Device information application profile is included. The SDK takes care of the Device information application profile message handling.
  - `#define EXCLUDE_DLG_DISS (1)`, the Device information application profile is excluded. The user application has to take care of the Device information application profile message handling.
- `user_profiles_config.h`, defines which BLE profiles (Bluetooth SIG adopted or custom ones) will be included in user's application. Particularly, the C header files (each header file denotes the respective BLE profile) that are included in the `user_profile_config.h` file are:
  - `CFG_PRF_DISS`, includes the Device Information Service Server Role
- `user_periph_setup.h`, holds hardware related settings relative to the used Development Kit.
- `user_periph_setup.c`, source code file that handles peripheral (GPIO, UART, etc.) configuration and initialization relative to the Development Kit.

8.2.5.2 Events Processing and Callbacks

Several events can occur during the lifetime of the BLE application and these events need to be handled in a specific manner. Also, operations need to be served depending on the application scenario. It depends on the application itself to define which events and operations are handled and how. The SDK is flexible enough to either call a default handler or call the user's defined event or operation handler.

The SDK mechanism that takes care of the above requirements, is the registration of callback functions for every event or operation. The C header file `user_callback_config.h`, which resides in user space, contains the registration of the callback functions.

The Pillar 1 application registers the following callback functions:

- General BLE events:

```
static const struct app_callbacks user_app_callbacks = {
    .app_on_connection              = user_app_connection,
    .app_on_disconnect              = user_app_disconnect,
    .app_on_update_params_rejected  = NULL,
    .app_on_update_params_complete  = NULL,
    .app_on_set_dev_config_complete = default_app_on_set_dev_config_complete,
    .app_on_adv_nonconn_complete    = NULL,
    .app_on_adv_undirect_complete   = user_app_adv_undirect_complete,
    .app_on_adv_direct_complete     = NULL,
    .app_on_db_init_complete        = default_app_on_db_init_complete,
    .app_on_scanning_completed      = NULL,
    .app_on_adv_report_ind          = NULL,
    .app_on_get_dev_appearance      = default_app_on_get_dev_appearance,
    .app_on_get_dev_slv_pref_params = default_app_on_get_dev_slv_pref_params,
    .app_on_set_dev_info            = default_app_on_set_dev_info,
    .app_on_data_length_change      = NULL,
```



```
    .app_on_update_params_request   = default_app_update_params_request,
#if (BLE_APP_SEC)
    .app_on_pairing_request         = NULL,
    .app_on_tk_exch_nomitm          = NULL,
    .app_on_irk_exch                = NULL,
    .app_on_csrk_exch               = NULL,
    .app_on_ltk_exch                = NULL,
    .app_on_pairing_succeded        = NULL,
    .app_on_encrypt_ind             = NULL,
    .app_on_mitm_passcode_req       = NULL,
    .app_on_encrypt_req_ind         = NULL,
    .app_on_security_req_ind        = NULL,
#endif // (BLE_APP_SEC)
};
```

- The above structure defines that a certain event will be processed by a default handler or by a user defined handler or it will not be processed at all (NULL entries). The user defined handlers (e.g. `user_app_connection()`, `user_app_disconnect()` and `user_app_adv_undirect_complete()`) are defined in C source file `user_barebone.c`.
- System specific events:

```
static const  struct arch_main_loop_callbacks user_app_main_loop_callbacks = {
    .app_on_init            = user_app_init,
    .app_on_ble_powered     = NULL,
    .app_on_sytem_powered   = NULL,
    .app_before_sleep       = NULL,
    .app_validate_sleep     = NULL,
    .app_going_to_sleep     = NULL,
    .app_resume_from_sleep  = NULL,
};
```

- The above structure defines that a certain event will be processed by a default handler or by a user defined handler or it will not be processed at all (NULL entries). The user defined handlers (e.g. `user_app_init()`) are defined in C source file `user_barebone.c`.
- BLE operations:

```
static const struct default_app_operations user_default_app_operations = {
    .default_operation_adv = user_app_adv_start,
};
```

- The above structure defines that a certain operation will be processed by a default handler or by a user defined handler or it will not be processed at all (NULL entries). The user defined handlers (e.g. `user_app_adv_start()`) is defined in C source file `user_barebone.c`.

8.2.5.3 BLE Application Abstract Code Flow

Figure 28 shows the abstract code flow diagram of the Pillar 1 application. The diagram depicts the SDK interaction with the callback functions registered in `user_callback_config.h` and the functions implemented in `user_barebone.c`.

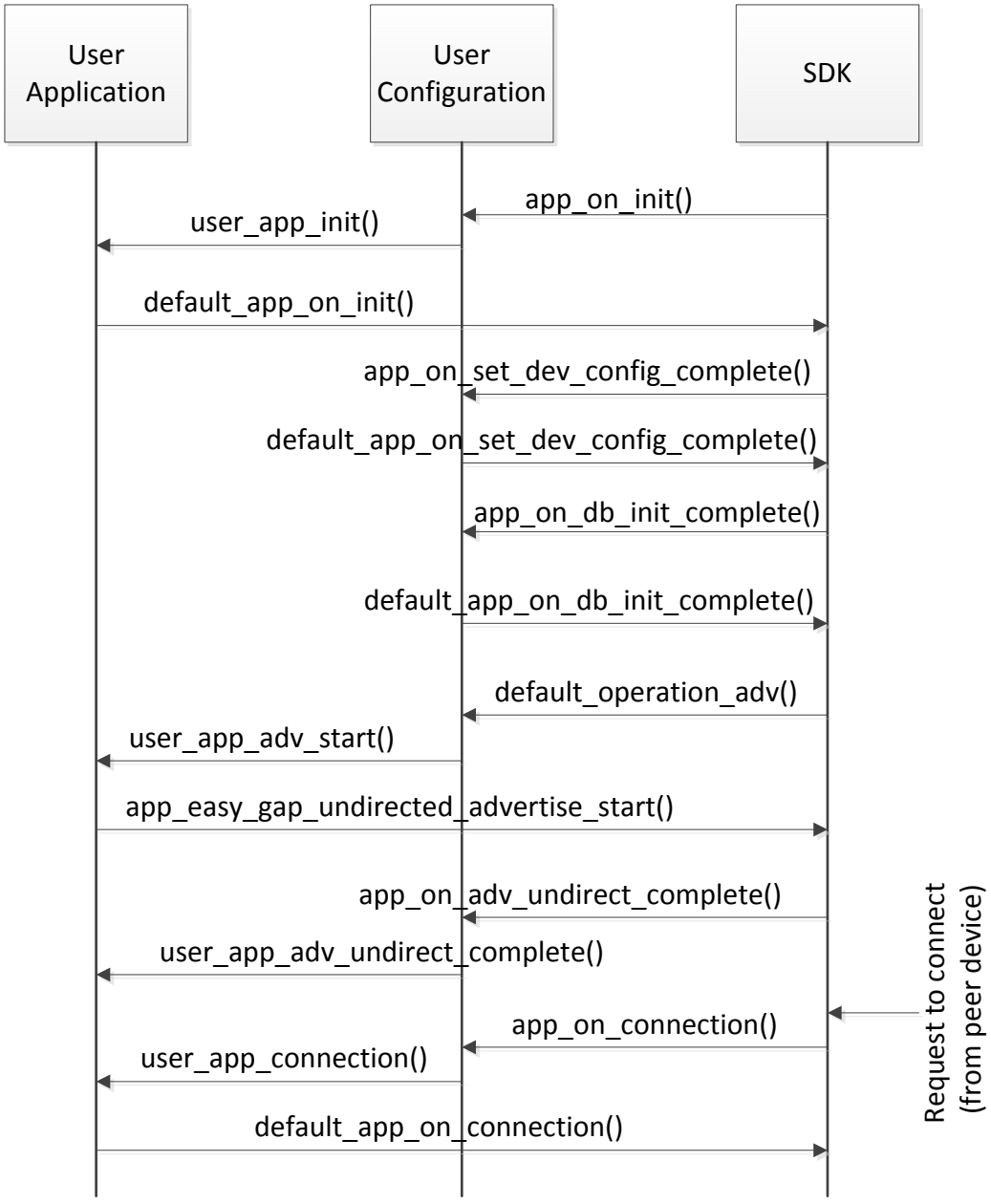


Figure 28: Pillar 1 Application - User Application Code Flow

8.2.6 Building the Project for Different Targets and Development Kits

The Pillar 1 application can be built for two different target processors: DA14585 and DA14586. The selection is done via the Keil tool as depicted in Figure 29.

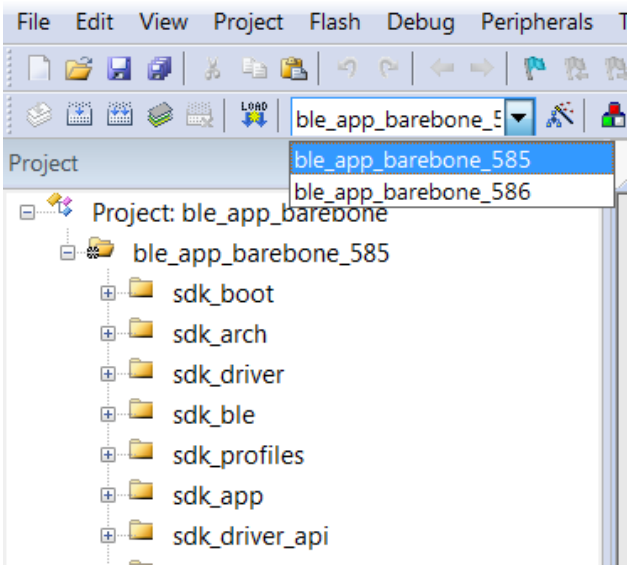


Figure 29: Building the Project for Different Targets

The user has also to select the correct Development Kit in order to build and run the application. This selection is done via the Configuration Wizard of the `user_periph_setup.h` file. See Figure 30.

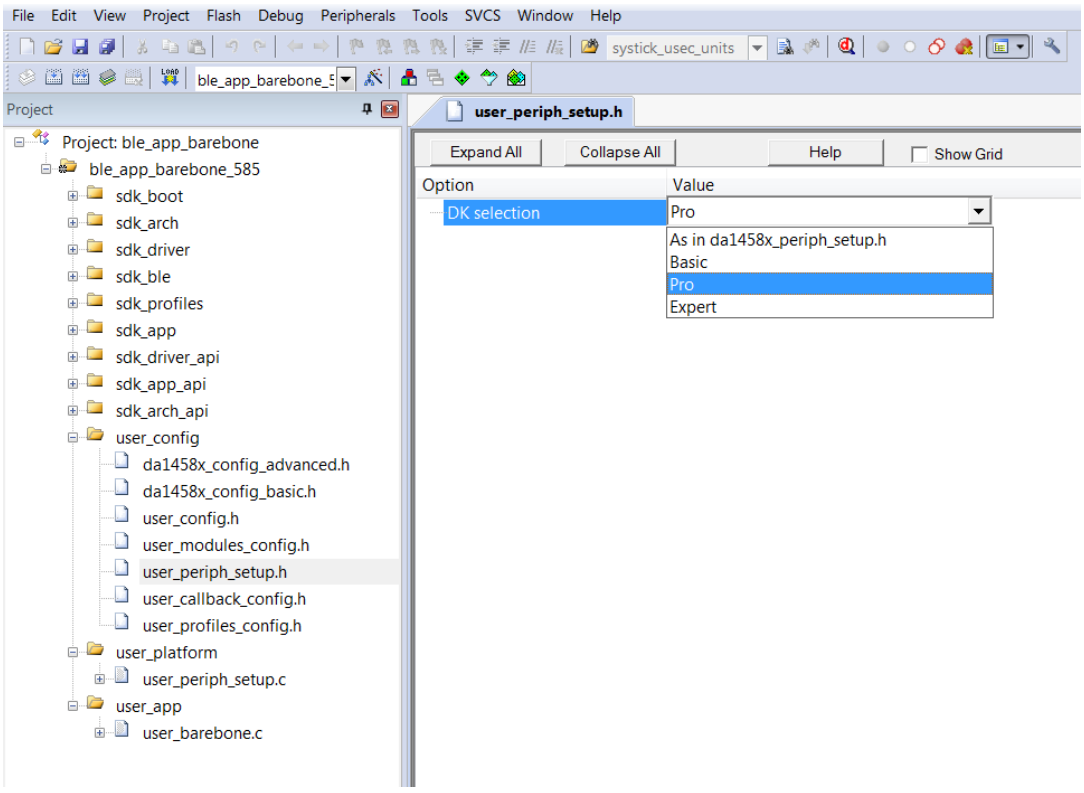


Figure 30: Development Kit Selection for Pillar 1 Application

After the proper selection of the target processor and development kit, the application is ready to be built.

8.2.7 Interacting with BLE Application

8.2.7.1 LightBlue iOS

The LightBlue iOS application can be used to connect an iPad/iPod/iPhone device to the application. In such a case the iPad/iPod/iPhone acts as a BLE Central and the application as a BLE Peripheral. Figure 31 shows the result when the iPad/iPod/iPhone device manages to connect to the DA14585/586 (the application's advertising device name is **DIALOG-BRBN**).

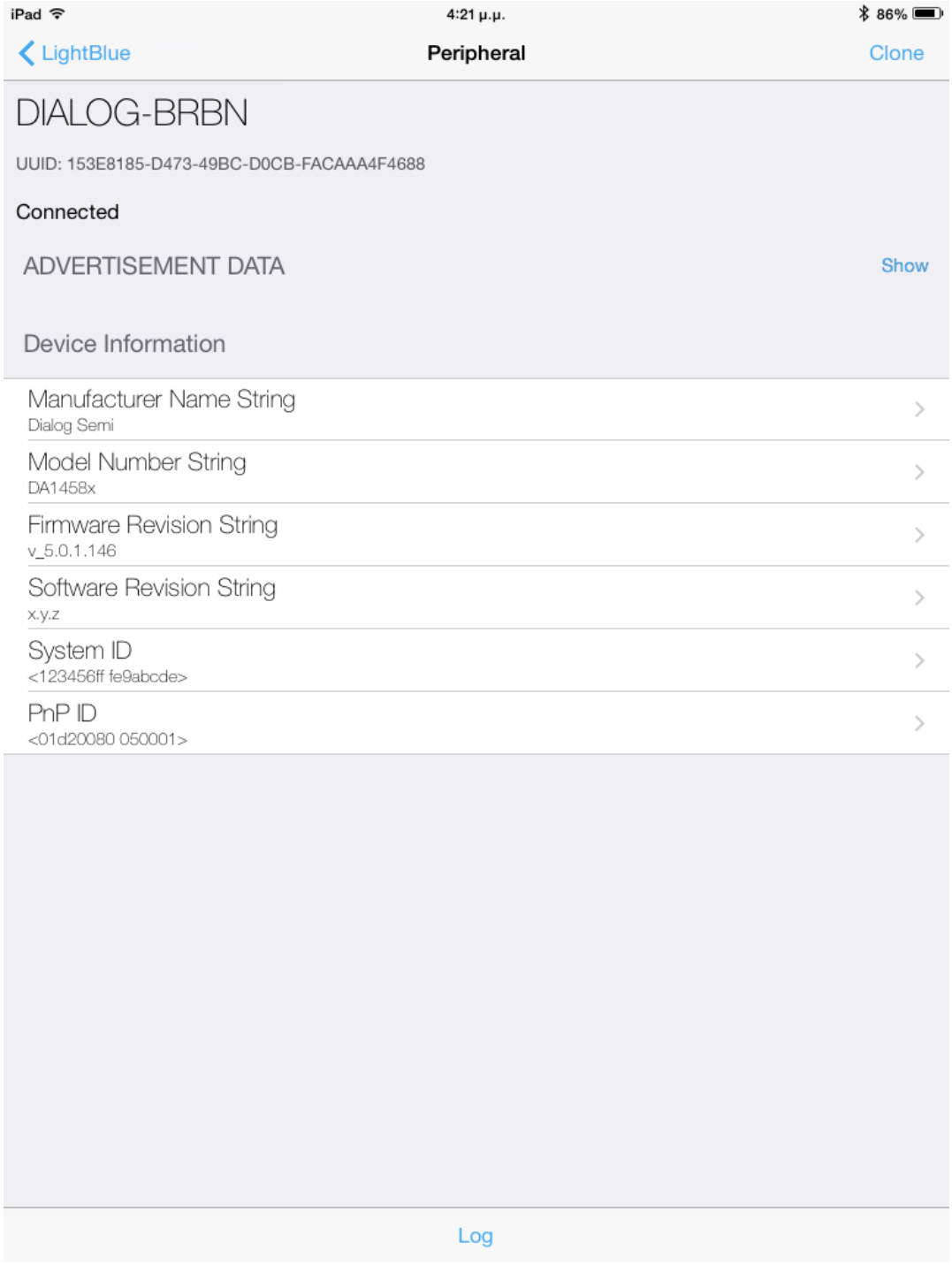


Figure 31: LightBlue Application Connected to Pillar 1 Application

8.3 Pillar 2 (Custom Profile)

8.3.1 Application Description

The Pillar 2 (custom profile) BLE example application demonstrates the same as the Pillar 1 application, plus the implementation of a custom service (128-bit UUID) defined by the user. The application demonstrates only the custom database creation. It uses the "Integrated processor" configuration.

8.3.2 Basic Operation

Supported services:

- Inherits the services from the Pillar 1 application, plus:
- Custom service defined by the user with 128-bit UUID.

Features:

- Inherits the features from Pillar 1 application, plus:
- Advertising data is updated on the fly every 10s:
  - Custom service support

The Pillar 2 behavior is included in C source file `user_profile.c`.

Table 13 shows the Custom service characteristic values along with their properties.

Table 13: Pillar 2 Custom Service Characteristic Values and Properties

Name	Properties	Length (B)	Description/Purpose
CONTROL POINT	WRITE	1	Accept commands from peer
LED STATE	WRITE NO RESPONSE	1	Toggles a LED connected to a GPIO
ADC VAL 1	READ, NOTIFY	2	Reads sample from an ADC channel
ADC VAL 2	READ	2	Reads sample from an ADC channel
BUTTON STATE	READ, NOTIFY	1	Reads the current state of a push button connected a GPIO
INDICATEABLE CHAR	READ, INDICATE	20	Demonstrate indications
LONG VAL CHAR	READ, WRITE. NOTIFY	50	Demonstrate writes to long characteristic value

The Pillar 2 application does not provide any behavior for the new added Custom service. It just demonstrates the database creation of the Custom service.

8.3.3 User Interface

A peer connected to the Pillar 2 application is able to do the same as in the Pillar 1 application, plus:

- Inspect the Custom service.

8.3.4 Loading the project

The Pillar 2 application is developed under the Keil v5 tool. The Keil project file is the:

`projects\target_apps\ble_examples\ble_app_profile\Keil_5\ble_app_profile.uvprojx`.

Figure 32 shows the Keil project layout with emphasis on the user related files, included in the Keil project folders `user_config`, `user_platform`, `user_custom_profile` and `user_app`. These folders contain the user configuration files of the Pillar 2 application.

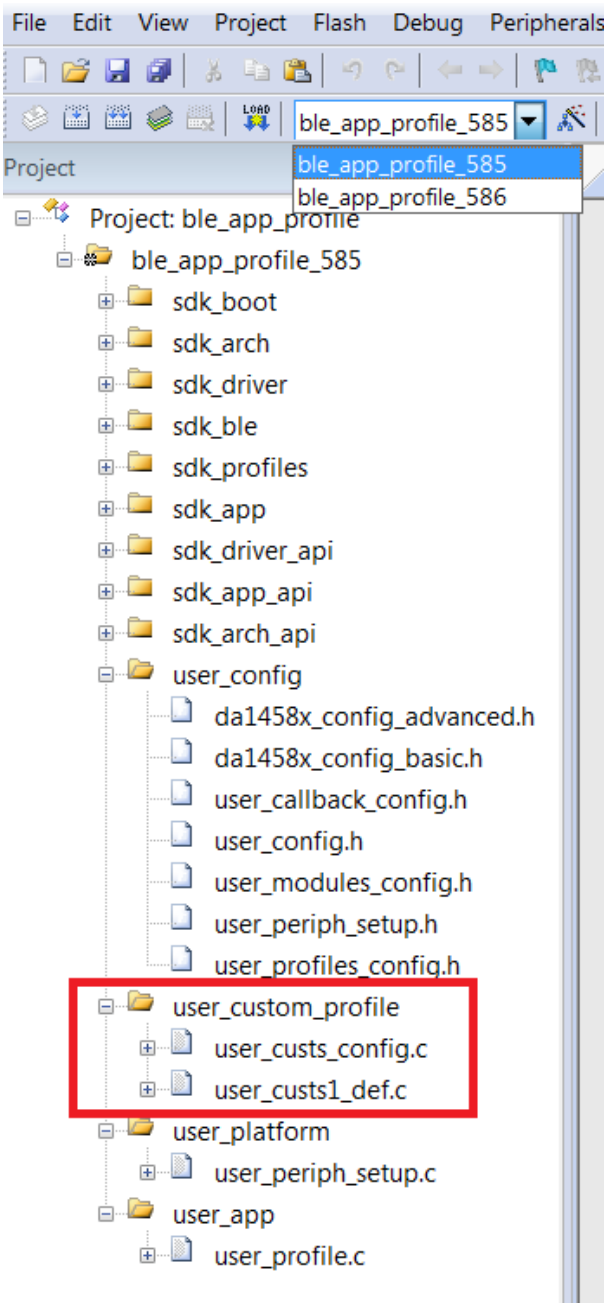


Figure 32: Pillar 2 Keil Project Layout

The newly added folder, compared to Pillar 1, is the `user_custom_profile`.

8.3.5 Going Through the Code
8.3.6 Initialization

The aforementioned Keil project folders (`user_config`, `user_platform`, `user_custom_profile` and `user_app`) contain the files that initialize and configure the Pillar 2 application.

- `da1458x_config_advanced.h`, holds DA14585/586 advanced configuration settings.
- `da1458x_config_basic.h`, holds DA14585/586 basic configuration settings.
- `user_callback_config.h`, callback functions that handle various events or operations.
- `user_config.h`, holds advertising parameters, connection parameters, etc.
- `user_config_sw_ver.h`, holds user specific information about software version.
- `user_modules_config.h`, defines which application modules are included or excluded from the user's application. For example:
  - #define EXCLUDE_DLG_DISS (0), the Device information application profile is included. The SDK takes care of the Device information application profile message handling.
  - #define EXCLUDE_DLG_DISS (1), the Device information application profile is excluded. The user application has to take care of the Device information application profile message handling.
- `user_profiles_config.h`, defines which BLE profiles (Bluetooth SIG adopted or custom ones) will be included in user's application. Particularly, the C header files (each header file denotes the respective BLE profile) that are included in the `user_profile_config.h` file are:
  - `CFG_PRF_DISS`, includes the Device Information service.
  - `CFG_PRF_CUST1`, includes the Custom 1 service.
- `user_custs1_def.c`, defines the structure of the Custom 1 profile database structure.
- `user_custs_config.c`, defines the cust_prf_funcs[] array, which contains the Custom profiles API functions calls.
- `user_periph_setup.h`, holds hardware related settings relative to the used Development Kit.
- `user_periph_setup.c`, source code file that handles peripheral (GPIO, UART, etc.) configuration and initialization relative to the Development Kit.

8.3.7 Events Processing and Callbacks

Several events can occur during the lifetime of the BLE application and these events need to be handled in a specific manner. Also, operations need to be served depending on the application scenario. It depends on the application itself to define which events and operations are handled and how. The SDK is flexible enough to either call a default handler or call the user's defined event or operation handler.

The SDK mechanism that takes care of the above requirements, is the registration of callback functions for every event or operation. The C header file `user_callback_config.h`, which resides in user space, contains the registration of the callback functions.

The Pillar 2 application registers the following callback functions:

- General BLE events:

```
static const struct app_callbacks user_app_callbacks = {
    .app_on_connection              = user_app_connection,
    .app_on_disconnect              = user_app_disconnect,
    .app_on_update_params_rejected  = NULL,
    .app_on_update_params_complete  = NULL,
    .app_on_set_dev_config_complete = default_app_on_set_dev_config_complete,
    .app_on_adv_nonconn_complete    = NULL,
    .app_on_adv_undirect_complete   = user_app_adv_undirect_complete,
    .app_on_adv_direct_complete     = NULL,
    .app_on_db_init_complete        = default_app_on_db_init_complete,
```



```
    .app_on_scanning_completed      = NULL,
    .app_on_adv_report_ind          = NULL,
    .app_on_get_dev_appearance      = default_app_on_get_dev_appearance,
    .app_on_get_dev_slv_pref_params = default_app_on_get_dev_slv_pref_params,
    .app_on_set_dev_info            = default_app_on_set_dev_info,
    .app_on_data_length_change      = NULL,
    .app_on_update_params_request   = default_app_update_params_request,
    #if (BLE_APP_SEC)
    .app_on_pairing_request         = NULL,
    .app_on_tk_exch_nomitm          = NULL,
    .app_on_irk_exch                = NULL,
    .app_on_csrk_exch               = NULL,
    .app_on_ltk_exch                = NULL,
    .app_on_pairing_succeded        = NULL,
    .app_on_encrypt_ind             = NULL,
    .app_on_mitm_passcode_req       = NULL,
    .app_on_encrypt_req_ind         = NULL,
    .app_on_security_req_ind        = NULL,
    #endif // (BLE_APP_SEC)
};
```

The above structure defines that a certain event will be processed by a default handler or by a user defined handler or it will not be processed at all (NULL entries). The user defined handlers (e.g user_app_connection(), user_app_disconnect() and user_app_adv_undirect_complete()) are defined in C source file `user_profile.c`.

- System specific events:

```
static const  struct arch_main_loop_callbacks user_app_main_loop_callbacks = {
    .app_on_init            = user_app_init,
    .app_on_ble_powered     = NULL,
    .app_on_sytem_powered   = NULL,
    .app_before_sleep       = NULL,
    .app_validate_sleep     = NULL,
    .app_going_to_sleep     = NULL,
    .app_resume_from_sleep  = NULL,
};
```

The above structure defines that a certain event will be processed by a default handler or by a user defined handler or it will not be processed at all (NULL entries). The user defined handler (e.g. user_app_init()) is defined in C source file `user_profile.c`.

- BLE operations:

```
static const struct default_app_operations user_default_app_operations = {
    .default_operation_adv = user_app_adv_start,
};
```

The above structure defines that a certain operation will be processed by a default handler or by a user defined handler or it will not be processed at all (NULL entries). The user defined handlers (e.g. user_app_adv_start()) is defined in C source file `user_profile.c`.

- Custom profile message handling:

```
static const catch_rest_event_func_t app_process_catch_rest_cb =
(catch_rest_event_func_t)user_catch_rest_hndl;
```

Callback function that contains the Custom profile messages handling in user application space. For Pillar 2 application this function is totally unused, since Pillar 2 application does not include any behavior related to the Custom service. Next Pillar examples will make use of it.

8.3.8 BLE Application Abstract Code Flow

Figure 33 shows the abstract code flow diagram of the Pillar 2 application. The diagram depicts the SDK interaction with the callback functions registered in `user_callback_config.h` and the functions implemented in `user_profile.c`.

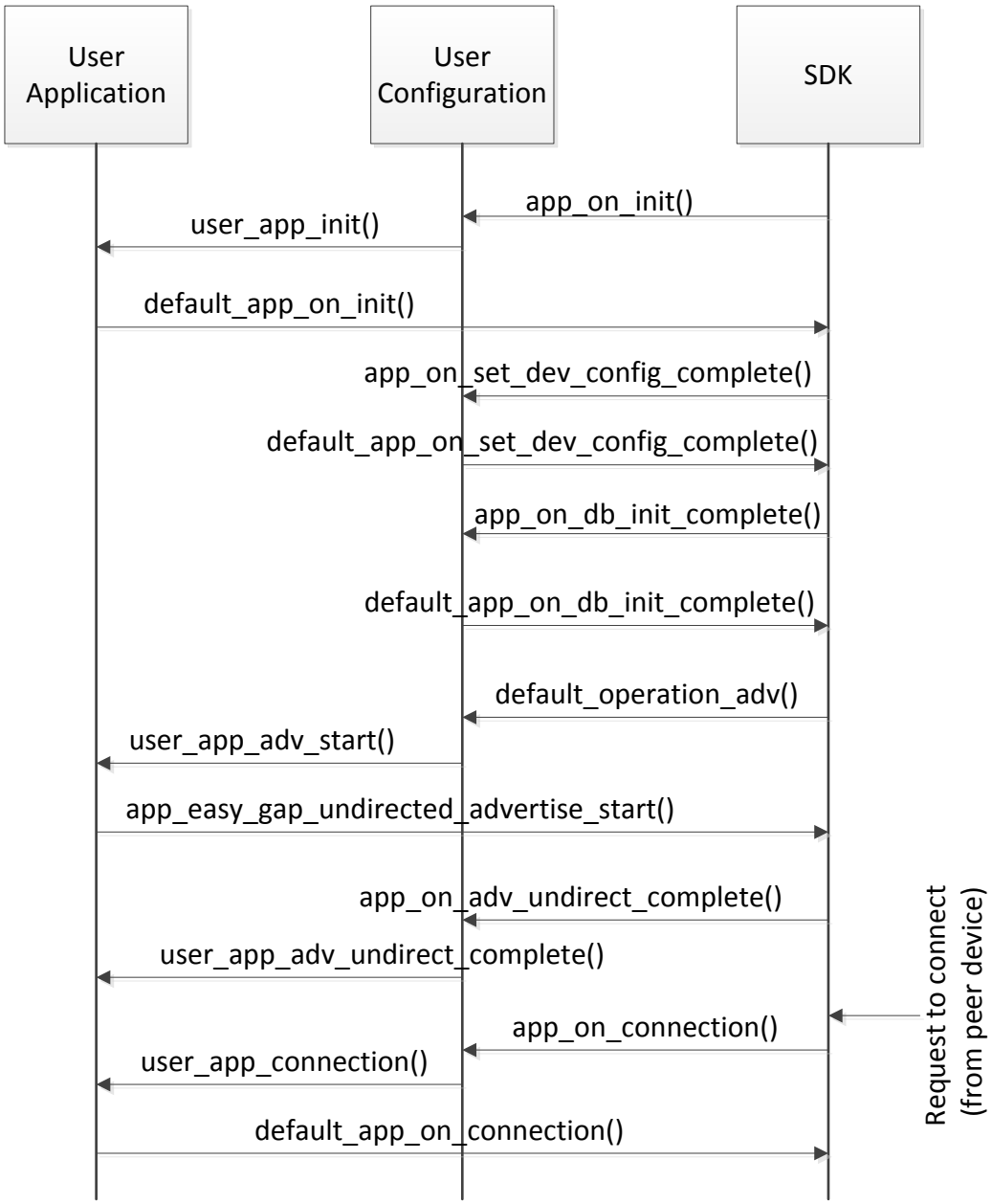


Figure 33: Pillar 2 Application - User Application Code Flow

8.3.9 Building the Project for Different Targets and Development Kits

The Pillar 2 application can be built for two different target processors: DA14585 and DA14586. The selection is done via the Keil tool as depicted in Figure 34.

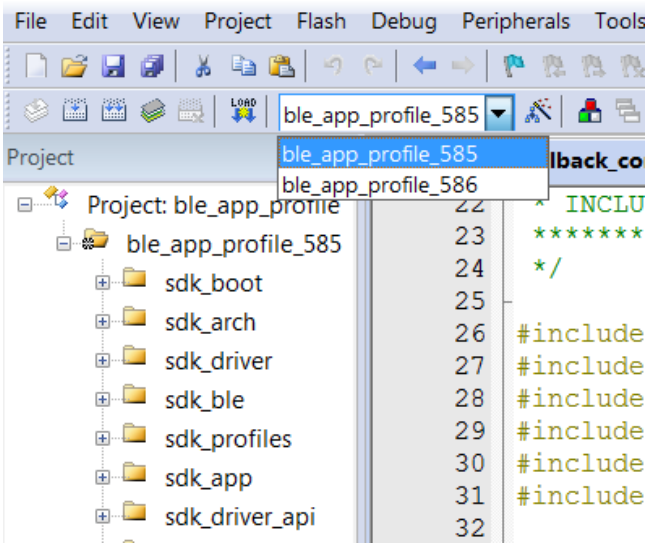


Figure 34: Building the Project for Different Targets

The user has also to select the correct Development Kit in order to build and run the application. This selection is done via the Configuration Wizard of the `user_periph_setup.h` file. See Figure 35.

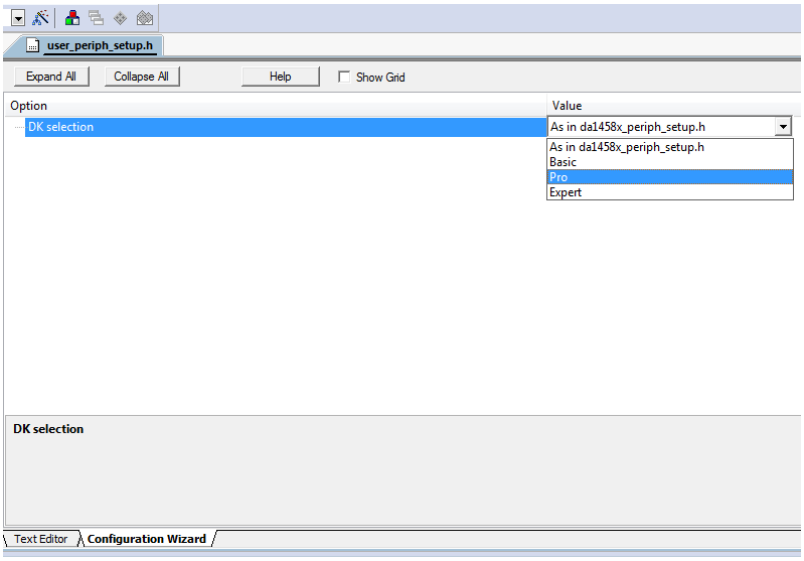


Figure 35: Development Kit Selection for Pillar 2 Application

After the proper selection of the target processor and development kit, the application is ready to be built.

8.3.10 Interacting with BLE Application
8.3.11 LightBlue iOS

The LightBlue iOS application can be used to connect an iPad/iPod/iPhone device to the application. In such a case the iPad/iPod/iPhone acts as a BLE Central and the application as a BLE Peripheral. Figure 36 shows the result when the iPad/iPod/iPhone device manages to connect to the DA14585/586 (the application's advertising device name is **DIALOG-PRFL**).

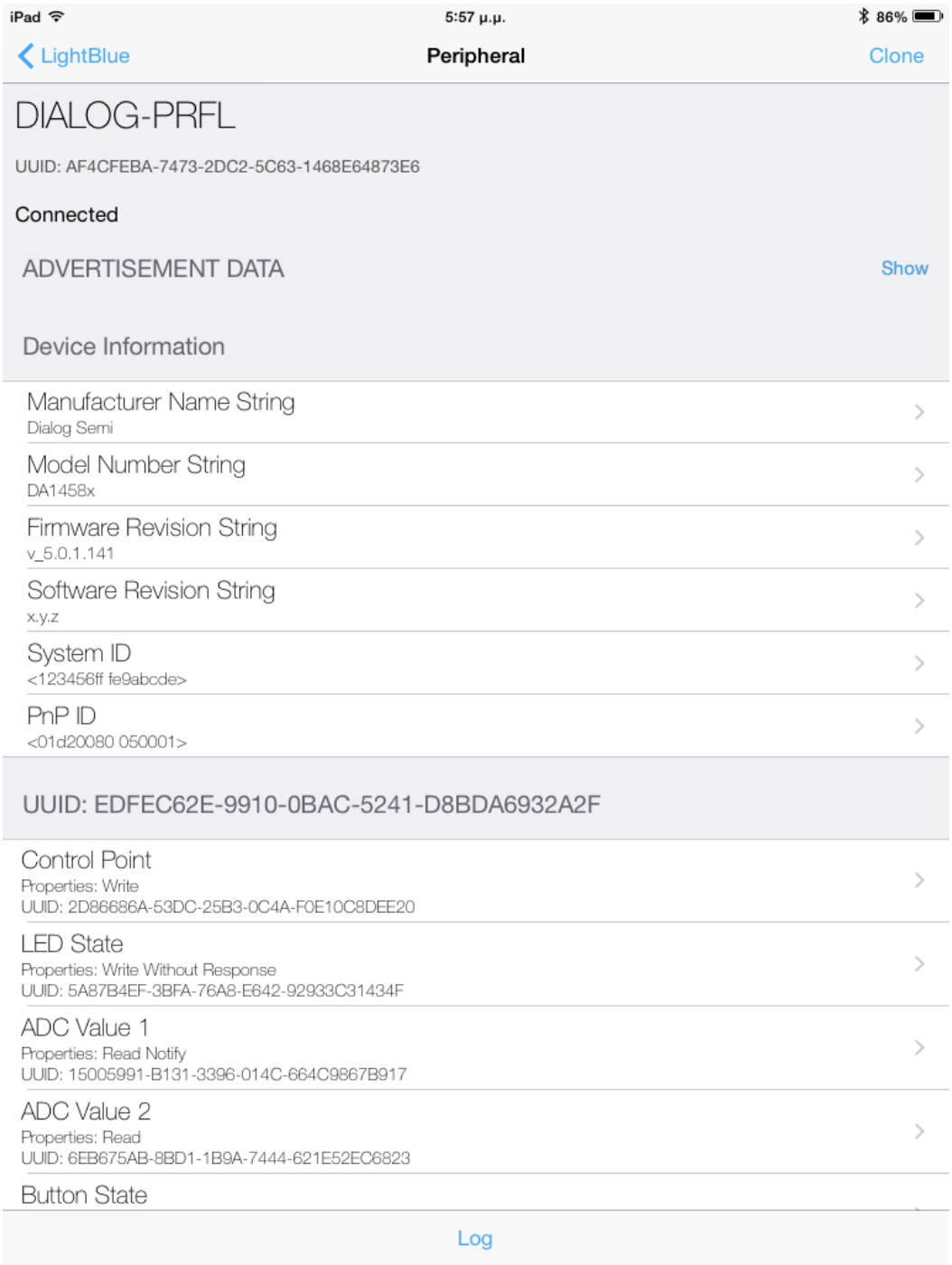


Figure 36: LightBlue Application Connected to Pillar 2 Application

8.4 Pillar 3 (Peripheral)

The Pillar 3 (peripheral) BLE example application demonstrates the same as the Pillar 2 application. The application also adds some basic interaction over the provided custom service (read/write/notify values). It uses the "Integrated processor" configuration.

8.4.1 Basic Operation

Supported services:

- Inherits the services from Pillar 2 application.

Features:

- Inherits the features from Pillar 2 application, plus:
- CONTROL POINT, LED STATE and ADC VAL 1 characteristic values introduce some behavior with a connected peer device.

The Pillar 3 application behavior is included in C source file `user_peripheral.c`.

Table 14 shows the Custom service characteristic values along with their properties.

Table 14. Pillar 3 Custom Service Characteristic Values and Properties

Name	Properties	Length (B)	Description/Purpose
CONTROL POINT	WRITE	1	Accept commands from peer
LED STATE	WRITE NO RESPONSE	1	Toggles a LED connected to a GPIO
ADC VAL 1	READ, NOTIFY	2	Reads sample from an ADC channel
ADC VAL 2	READ	2	Reads sample from an ADC channel
BUTTON STATE	READ, NOTIFY	1	Reads the current state of a push button connected a GPIO
INDICATEABLE CHAR	READ, INDICATE	20	Demonstrate indications
LONG VAL CHAR	READ, WRITE. NOTIFY	50	Demonstrate writes to long characteristic value

The Pillar 3 application provides behavior only for the highlighted characteristic values of the Custom service. The implementation code of the Custom service is included in C source file `user_custs1_impl.c`.

8.4.2 User Interface

A peer connected to the Pillar 3 application is able to do the same as in the Pillar 2 application, plus:

- Use the Custom service.
- Write to **Control Point** of the Custom Service:
  - Byte 0x00 disables **ADC VAL 1** auto notifications (disables respective timer).
  - Byte 0x01 enables **ADC VAL 1** auto notifications (enables respective timer).
- Write to **LED STATE** of the Custom Service:
  - Byte 0x00 turns an LED off.
  - Byte 0x01 turns an LED on.
- Read/notify **ADC VAL 1** value. When the **ADC VAL 1** auto notifications are turned on and the notify operation is required by the peer, a 16-bit counter is incremented. This counter value emulates the Analog value of the **ADC VAL 1** (there is no hardware support for reading an Analog value).

The selected LED (port and pin number of the DA14585/586) is defined by the user configuration depending on the underlying hardware (Development Kit). The user files `user_periph_setup.h` and `user_periph_setup.c` hold the peripheral configuration settings of the LED.

8.4.3 Loading the Project

The Pillar 3 application is developed under the Keil v5 tool. The Keil project file is the:

`<sdk_root_directory>\projects\target_apps\ble_examples\ble_app_peripheral\ble_app_peripheral.uvprojx`.

Figure 37 shows the Keil project layout with emphasis on the user related files, included in the Keil project folders `user_config`, `user_platform`, `user_custom_profile` and `user_app`. These folders contain the user configuration files of the Pillar 3 application.

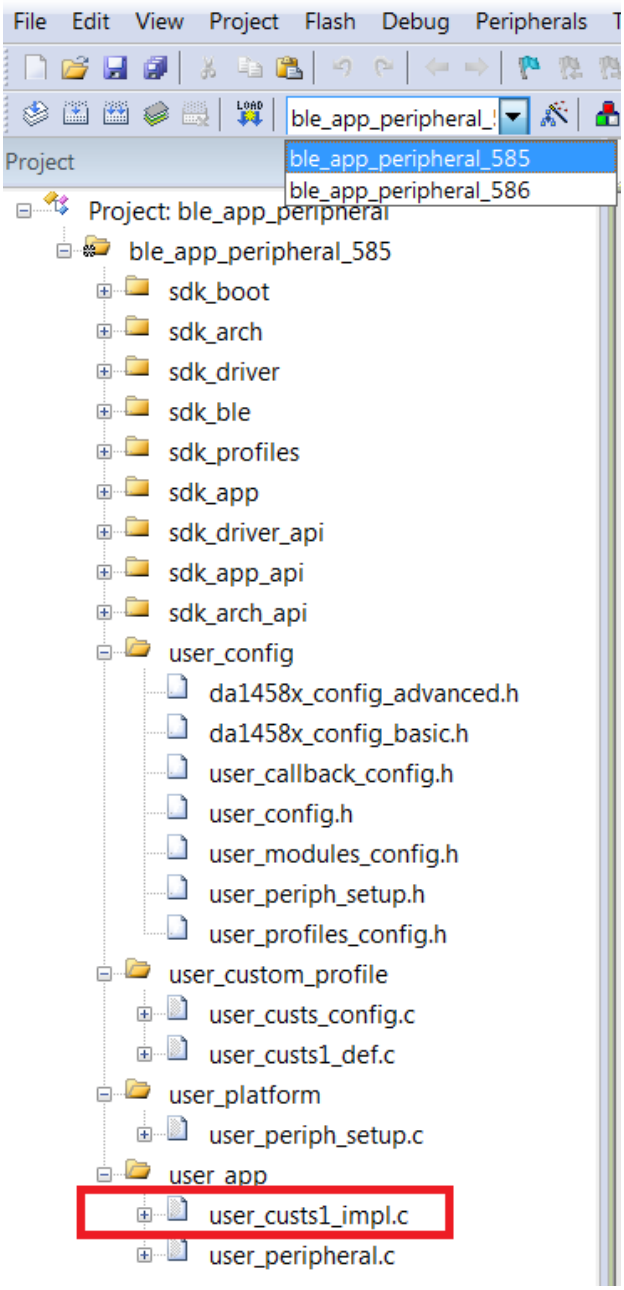


Figure 37: Pillar 3 Keil Project Layout

The newly added file, compared to Pillar 2, is the `user_custs1_impl.c`.

8.4.4 Going Through the Code
8.4.5 Initialization

The aforementioned Keil project folders (`user_config`, `user_platform`, `user_custom_profile` and `user_app`) contain the files that initialize and configure the Pillar 3 application.

- `da1458x_config_advanced.h`, holds DA14585/586 advanced configuration settings.
- `da1458x_config_basic.h`, holds DA14585/586 basic configuration settings.
- `user_callback_config.h`, callback functions that handle various events or operations.
- `user_config.h`, holds advertising parameters, connection parameters, etc.
- `user_config_sw_ver.h`, holds user specific information about software version.
- `user_modules_config.h`, defines which application modules are included or excluded from the user's application. For example:
  - #define EXCLUDE_DLG_DISS (0), the Device information application profile is included. The SDK takes care of the Device information application profile message handling.
  - #define EXCLUDE_DLG_DISS (1), the Device information application profile is excluded. The user application has to take care of the Device information application profile message handling.
- `user_profiles_config.h`, defines which BLE profiles (Bluetooth SIG adopted or custom ones) will be included in user's application. Particularly, the C header files (each header file denotes the respective BLE profile) that are included in the `user_profile_config.h` file are:
  - `CFG_PRF_DISS`, includes the Device Information service.
  - `CFG_PRF_CUST1`, includes the Custom 1 service.
- `user_custs1_def.c`, defines the structure of the Custom 1 profile database structure.
- `user_custs_config.c`, defines the cust_prf_funcs[] array, which contains the Custom profiles API functions calls.
- `user_periph_setup.h`, holds hardware related settings relative to the used Development Kit.
- `user_periph_setup.c`, source code file that handles peripheral (GPIO, UART, etc.) configuration and initialization relative to the Development Kit.

8.4.6 Events Processing and Callbacks

Several events can occur during the lifetime of the application. It depends on the application which of these events are handled and how. The SDK is flexible enough to either call a default handler or call the user's defined event handler upon the occurrence of a particular event. The configuration file `user_callback_config.h` contains the configuration array that defines if an event is processed or not (callback function is present or not). For example, in the Pillar 3 application the user_app_callbacks[] array has the following entries:

```
static const struct app_callbacks user_app_callbacks = {
    .app_on_connection              = user_app_connection,
    .app_on_disconnect              = user_app_disconnect,
    .app_on_update_params_rejected  = NULL,
    .app_on_update_params_complete  = NULL,
    .app_on_set_dev_config_complete = default_app_on_set_dev_config_complete,
    .app_on_adv_nonconn_complete    = NULL,
    .app_on_adv_undirect_complete   = user_app_adv_undirect_complete,
    .app_on_adv_direct_complete     = NULL,
    .app_on_db_init_complete        = default_app_on_db_init_complete,
    .app_on_scanning_completed      = NULL,
    .app_on_adv_report_ind          = NULL,
    #if (BLE_APP_SEC)
    .app_on_pairing_request         = NULL,
    .app_on_tk_exch_nomitm          = NULL,
    .app_on_irk_exch                = NULL,
```

```
    .app_on_csrk_exch               = NULL,
    .app_on_ltk_exch                = NULL,
    .app_on_pairing_succeded        = NULL,
    .app_on_encrypt_ind             = NULL,
    .app_on_mitm_passcode_req       = NULL,
    .app_on_encrypt_req_ind         = NULL,
    #endif // (BLE_APP_SEC)
};
```

The above array defines that a certain event will be processed by a default handler or by a user defined handler or it will not be processed at all (NULL entries). The user defined handlers (e.g. user_app_connection(), user_app_disconnect() and user_app_adv_undirect_complete()) are defined in C source file user_peripheral.c.

An important addition to Pillar 3 application is the user_catch_rest_hndl() handler that catches all the Custom profile messages and handles them in the user application space. The implementation of this handler is in C source file `user_custs1_impl.c`. These Custom profile messages are application specific and their handling is transferred to user space. The SDK is agnostic of the specific Custom profile messages and it is the user's application responsibility to handle them.

The `user_callback_config.h` configuration header file contains the registration of the callback function user_catch_rest_hndl(), as is described below.

```
static const catch_rest_event_func_t app_process_catch_rest_cb =
(catch_rest_event_func_t)user_catch_rest_hndl;
```


8.4.7 BLE Application Abstract Code Flow

Figure 38 shows the abstract code flow diagram of the Pillar 3 application. The diagram depicts the SDK interaction with the callback functions registered in `user_callback_config.h` and the functions implemented in `user_peripheral.c`.

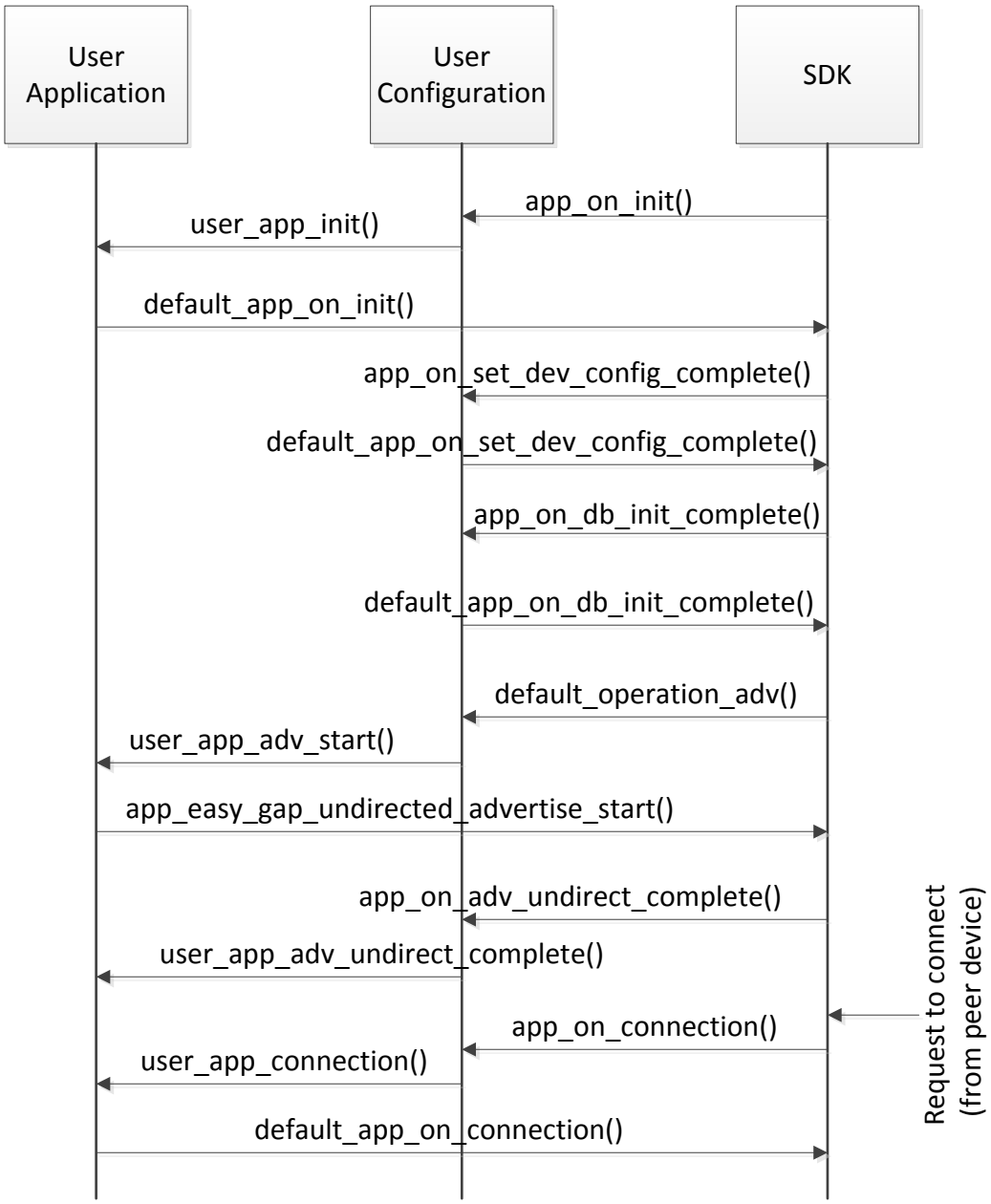


Figure 38: Pillar 3 Application - User Application Code Flow

8.4.8 Building the Project for Different Targets and Development Kits

The Pillar 3 application can be built for two different target processors: DA14585 and DA14586. The selection is done via the Keil tool as depicted in Figure 39.

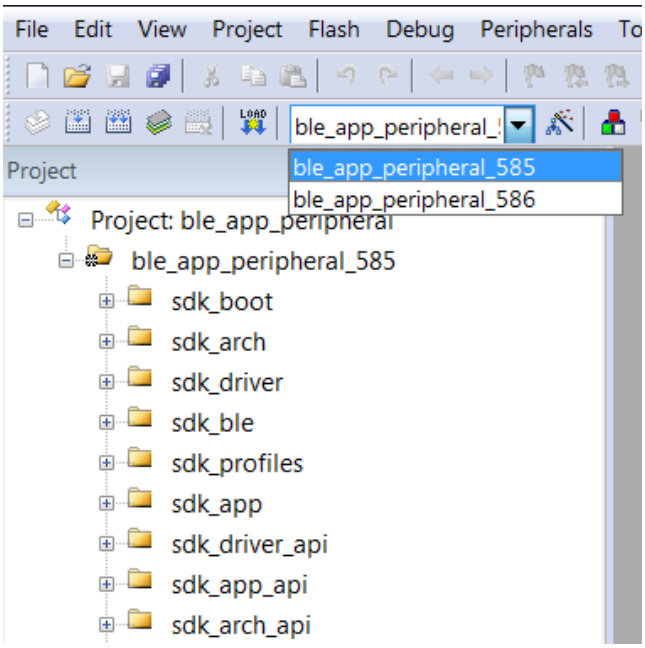


Figure 39: Building the Project for Different Targets

The user has also to select the correct Development Kit in order to build and run the application. This selection is done via the Configuration Wizard of the `user_periph_setup.h` file. See Figure 40.

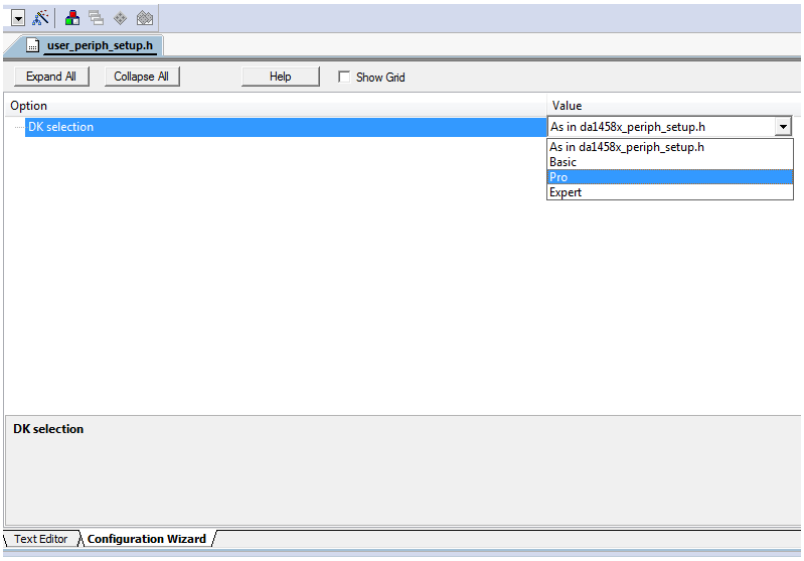


Figure 40: Development Kit Selection for Pillar 3 Application

After the proper selection of the target processor and development kit, the application is ready to be built.

8.4.9 Interacting with BLE Application
8.4.10 LightBlue iOS

The LightBlue iOS application can be used to connect an iPad/iPod/iPhone device to the application. In such a case the iPad/iPod/iPhone acts as a BLE Central and the application as a BLE Peripheral. Figure 41 shows the result when the iPad/iPod/iPhone device manages to connect to the DA14585/586(the application's advertising device name is **DIALOG-PRPH**).

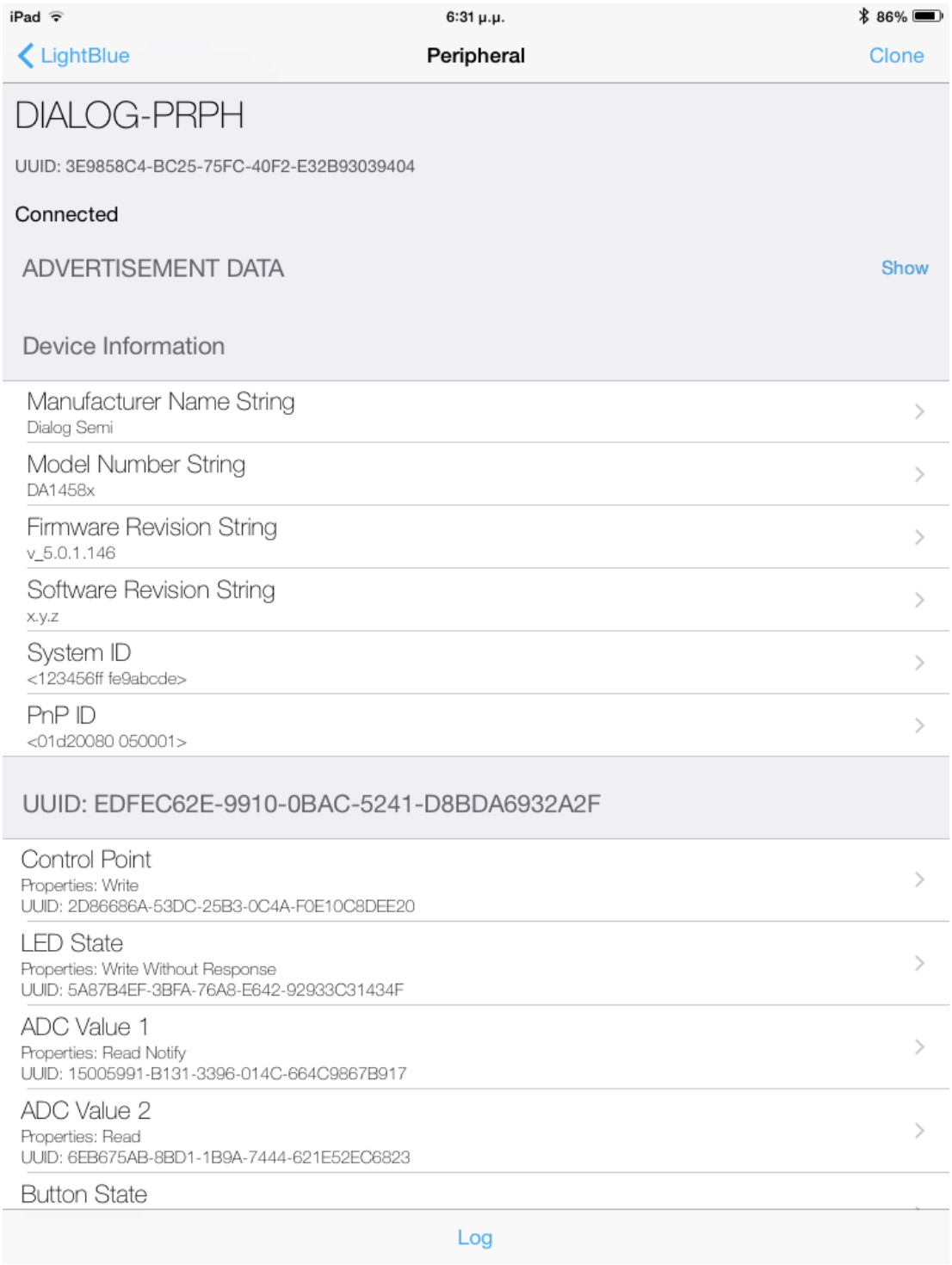


Figure 41: LightBlue Application Connected to Pillar 3 Application

8.5 Pillar 4 (Security)

8.5.1 Application Description

The Pillar 4 (security) BLE example application demonstrates the same as the Pillar 2 application. The application's main purpose is the testing of the various security models and bonding procedures which are selected in compile time. It uses the "Integrated processor" configuration.

8.5.2 Basic Operation

Supported services:

- Inherits the services from Pillar 2 application.

Features:

- Inherits the features from Pillar 2 application, plus:
- Access to the service inherited from Pillar 2 is protected according to current security configuration set through the `APP_SECURITY_PRESET_CONFIG` flag.

The Pillar 4 application behavior is included in C source file `user_security.c`.

Table 15 shows the Custom service characteristic values along with their properties.

Table 15: Pillar 4 Custom Service Characteristic Values and Properties

Name	Properties	Length (B)	Description/Purpose
CONTROL POINT	WRITE	1	Accept commands from peer
LED STATE	WRITE NO RESPONSE	1	Toggles a LED connected to a GPIO
ADC VAL 1	READ, NOTIFY	2	Reads sample from an ADC channel
ADC VAL 2	READ	2	Reads sample from an ADC channel
BUTTON STATE	READ, NOTIFY	1	Reads the current state of a push button connected a GPIO
INDICATEABLE CHAR	READ, INDICATE	20	Demonstrate indications
LONG VAL CHAR	READ, WRITE. NOTIFY	50	Demonstrate writes to long characteristic value

The Pillar 4 application does not provide any behavior for the added Custom service. It just demonstrates the various security features and the creation of the Custom service. Every access to the Custom service is possible only when certain security conditions are met. This may include valid bond and/or encrypted connection. It depends on the currently selected security setup.

8.5.3 User Interface

A peer connected to the Pillar 4 application is able to do the same as in the Pillar 2 application, plus:

- Use the Custom service with certain preconditions.

The preconditions required reading from or writing to a Custom service's characteristics or reading their descriptors depends on the currently selected security preset.

8.5.4 Loading the Project

The Pillar 4 application is developed under the Keil v5 tool. The Keil project file is the:

`projects\target_apps\ble_examples\ble_app_security\Keil_5\ble_app_security.uvprojx`

The following picture shows the Keil project layout with emphasis on the user related files, included in the Keil project folders `user_config`, `user_platform`, `user_custom_profile` and `user_app`. These folders contain the user configuration files of the Pillar 4 application.

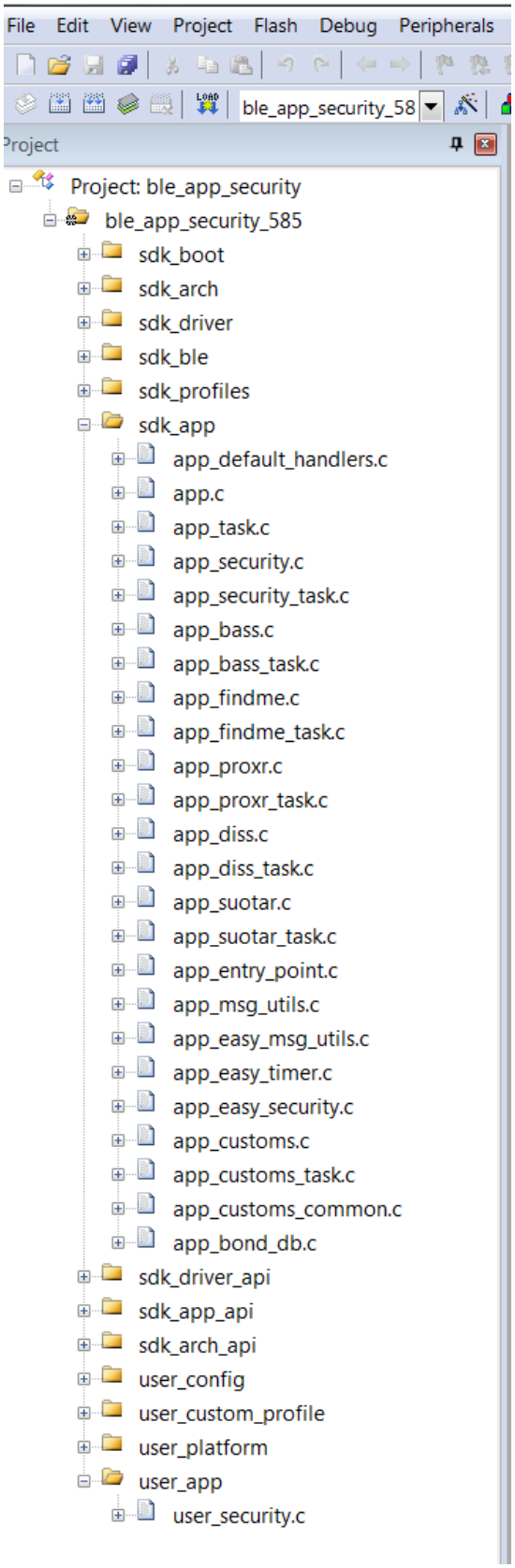


Figure 42: Pillar 4 Keil Project Layout

The newly added file, compared to Pillar 2, is the app_bond_db.c. It contains a simple bond information database API.

8.5.5 Going Through the Code
8.5.6 Initialization

The aforementioned Keil project folders (`user_config`, `user_platform`, `user_custom_profile` and `user_app`) contain the files that initialize and configure the Pillar 4 application.

- `da1458x_config_advanced.h`, holds DA14585/586advanced configuration settings.
- `da1458x_config_basic.h`, holds DA14585/586basic configuration settings.
- `user_callback_config.h`, callback functions that handle various events or operations.
- `user_config.h`, holds advertising parameters, connection parameters, etc. It also holds the flag used for configuring the security features at compile time. For example:
  - #define USER_CFG_PAIR_METHOD_JUST_WORKS, the device is using the Just Works pairing method.
  - #define USER_CFG_PAIR_METHOD_PASSKEY, the device is using Pass Key pairing method.
  - #define USER_CFG_PAIR_METHOD_OOB, the device is using the Out of Band (OOB) pairing method.
  - **Note**: At the time of writing this document, neither Android nor iOS support the Out of Band (OOB) mechanism for Bluetooth pairing.
  - The user can define one of the above pairing methods, if the application requires it. If none of the above flag is defined, then the security features are turned off.
- This configuration header file allows also for selecting Privacy Feature of the peripheral device. This feature allows the device to use random addresses to prevent peers from tracking it. Privacy feature is selected through the following two flags. For example:
  - #define USER_CFG_PRIV_GEN_STATIC_RND, the device is using a random address generated automatically by the BLE stack. This address is static during device's power cycle.
  - #define USER_CFG_PRIV_GEN_RSLV_RND, the device is using a resolvable random address, generated automatically by the BLE stack. This address is changing in certain time intervals. Only bonded devices that own the Identity Resolving Key, distributed during the pairing procedure, can resolve the Random Address and track the device.
  - If none of the above flags is selected the device is not using any Privacy Feature, and will use its public address.
  - Peer device's bond data can be stored on an external SPI Flash or I2C EEPROM memory.
  - #define USER_CFG_APP_BOND_DB_USE_SPI_FLASH, for SPI Flash
  - #define USER_CFG_APP_BOND_DB_USE_I2C_EEPROM, for I2C EEPROM.
  - If none of the above flags is defined the bond data have to be stored in the application RAM.
- `user_config_sw_ver.h`, holds user specific information about software version.
- `user_modules_config.h`, defines which application modules are included or excluded from the user's application. For example:
  - #define EXCLUDE_DLG_DISS (0), the Device information application profile is included. The SDK takes care of the Device information application profile message handling.
  - #define EXCLUDE_DLG_DISS (1), the Device information application profile is excluded. The user application has to take care of the Device information application profile message handling.
  - #define EXCLUDE_DLG_SEC (0), the Security application module is included in this Pillar application.
- `user_profiles_config.h`, defines which BLE profiles (Bluetooth SIG adopted or custom ones) will be included in user's application. Particularly, the C header files (each header file denotes the respective BLE profile) that are included in the `user_profile_config.h` file are:
  - `CFG_PRF_DISS`, includes the Device Information service.
  - `CFG_PRF_CUST1`, includes the Custom 1 service.

- user_custs1_def.c, defines the structure of the Custom 1 profile database structure.
- user_custs_config.c, defines the cust_prf_funcs[] array, which contains the Custom profiles API functions calls.
- user_periph_setup.h, holds hardware related settings relative to the used Development Kit. In this particular application it also defines the I2C pin configuration for the EEPROM module.
- user_periph_setup.c, source code file that handles peripheral (GPIO, UART, etc.) configuration and initialization relative to the Development Kit.

8.5.7 Events Processing and Callbacks

Several events can occur during the lifetime of the BLE application and these events need to be handled in a specific manner. Also, operations need to be served depending on the application scenario. It depends on the application itself to define which events and operations are handled and how. The SDK is flexible enough to either call a default handler or call the user's defined event or operation handler.

The SDK mechanism that takes care of the above requirements, is the registration of callback functions for every event or operation. The C header file user_callback_config.h, which resides in user space, contains the registration of the callback functions.

The Pillar 4 application registers the following callback functions:

- General BLE events:

```
static const struct app_callbacks user_app_callbacks = {
    .app_on_connection              = user_app_connection,
    .app_on_disconnect              = user_app_disconnect,
    .app_on_update_params_rejected  = NULL,
    .app_on_update_params_complete  = NULL,
    .app_on_set_dev_config_complete = default_app_on_set_dev_config_complete,
    .app_on_adv_nonconn_complete    = NULL,
    .app_on_adv_undirect_complete   = user_app_adv_undirect_complete,
    .app_on_adv_direct_complete     = NULL,
    .app_on_db_init_complete        = default_app_on_db_init_complete,
    .app_on_scanning_completed      = NULL,
    .app_on_adv_report_ind          = NULL,
     .app_on_get_dev_appearance      = default_app_on_get_dev_appearance,
    .app_on_get_dev_slv_pref_params = default_app_on_get_dev_slv_pref_params,
    .app_on_set_dev_info            = default_app_on_set_dev_info,
    .app_on_data_length_change      = NULL,
    .app_on_update_params_request   = default_app_update_params_request,
    #if (BLE_APP_SEC)
    .app_on_pairing_request         = default_app_on_pairing_request,
    .app_on_tk_exch_nomitm          = user_app_on_tk_exch_nomitm,
    .app_on_irk_exch                = NULL,
    .app_on_csrk_exch               = NULL,
    .app_on_ltk_exch                = default_app_on_ltk_exch,
    .app_on_pairing_succeded        = user_app_on_pairing_succeded,
    .app_on_encrypt_ind             = NULL,
    .app_on_mitm_passcode_req       = NULL,
    .app_on_encrypt_req_ind         = user_app_on_encrypt_req_ind,
    .app_on_security_req_ind        = NULL,
    #endif // (BLE_APP_SEC)
};
```

The above structure defines that a certain event will be processed by a default handler or by a user defined handler or it will not be processed at all (NULL entries). The user defined handlers (e.g user_app_connection(), user_app_disconnect(), user_app_adv_undirect_complete(), user_app_on_tk_exch_nomitm(), user_app_on_pairing_succeded() and user_app_on_encrypt_req_ind()) are defined in C source file user_security.c. Note that most

of them will be called from the newly enabled security application module (the preprocessor value BLE_APP_SEC must be defined).

- System specific events:

```
static const  struct arch_main_loop_callbacks user_app_main_loop_callbacks = {
    .app_on_init            = user_app_init,
    .app_on_ble_powered     = NULL,
    .app_on_sytem_powered   = NULL,
    .app_before_sleep       = NULL,
    .app_validate_sleep     = NULL,
    .app_going_to_sleep     = NULL,
    .app_resume_from_sleep  = NULL,
};
```

  The above structure defines that a certain event will be processed by a default handler or by a user defined handler or it will not be processed at all (NULL entries). The user defined handlers (e.g. user_app_init()) is defined in C source file `user_security.c`.

- BLE operations:

```
static const struct default_app_operations user_default_app_operations = {
    .default_operation_adv = user_app_adv_start,
};
```

  The above structure defines that a certain operation will be processed by a default handler or by a user defined handler or it will not be processed at all (NULL entries). The user defined handlers (e.g. user_app_adv_start()) is defined in C source file `user_security.c`.

- Custom profile message handling:

```
static const catch_rest_event_func_t app_process_catch_rest_cb =
(catch_rest_event_func_t)user_catch_rest_hndl;
```

  Callback function that contains the Custom profile messages handling in user application space. For Pillar 4 application this function is totally unused, since Pillar 4 application does not include any behavior related to the Custom service. Service is protected from the unauthorised access automatically at database level and requires no additional action from the user application space.

8.5.8 BLE Application Abstract Code Flow

Figure 43 shows the abstract code flow diagram of the Pillar 4 application. The diagram depicts the SDK interaction with the callback functions registered in `user_callback_config.h` and the functions implemented in `user_security.c`. It shows only the part that is new, compared to the previous Pillar application. The connection establishment procedure is the exactly the same as in the previous application and the following function flow diagram shows the subsequent pairing procedure using the passkey entry method.

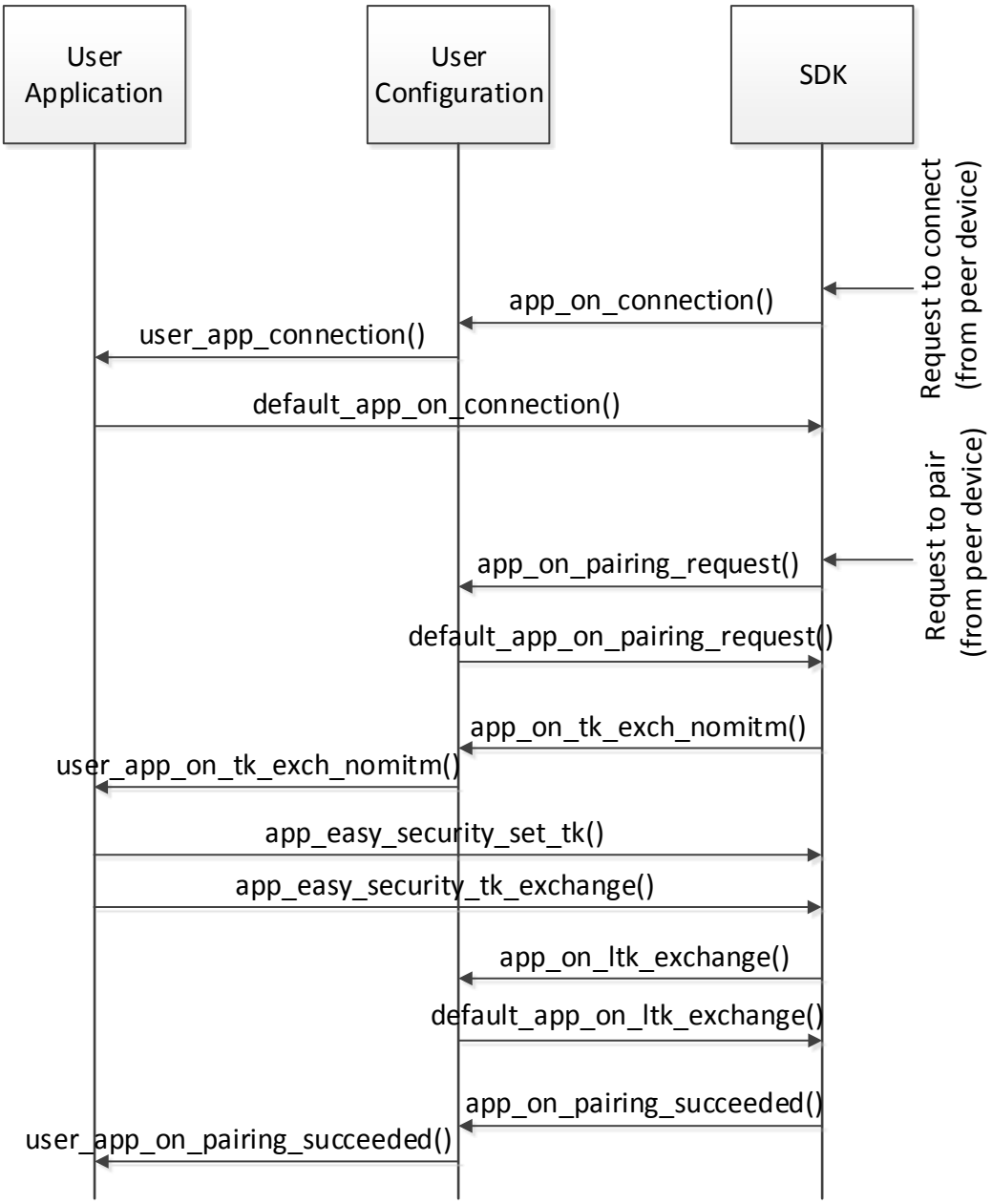


Figure 43: Pillar 4 Application - User Application Code Flow for Pairing using Passkey Entry

`app_easy_security_set_tk()` and `app_easy_security_tk_exchange()` are called from the `user_app_on_tk_exchange_nomitm()`. Note that `app_on_tk_exch_nomitm()` is called also in case of a pairing method that uses passkey entry. This is because the Pillar 4 application requires no user input due to the current input/output capabilities. It is the peer device that needs to enter the passkey.

One of the alternative security configurations allows the use of the Just Works method of pairing. In such case the function call diagram looks slightly different. See Figure 44.

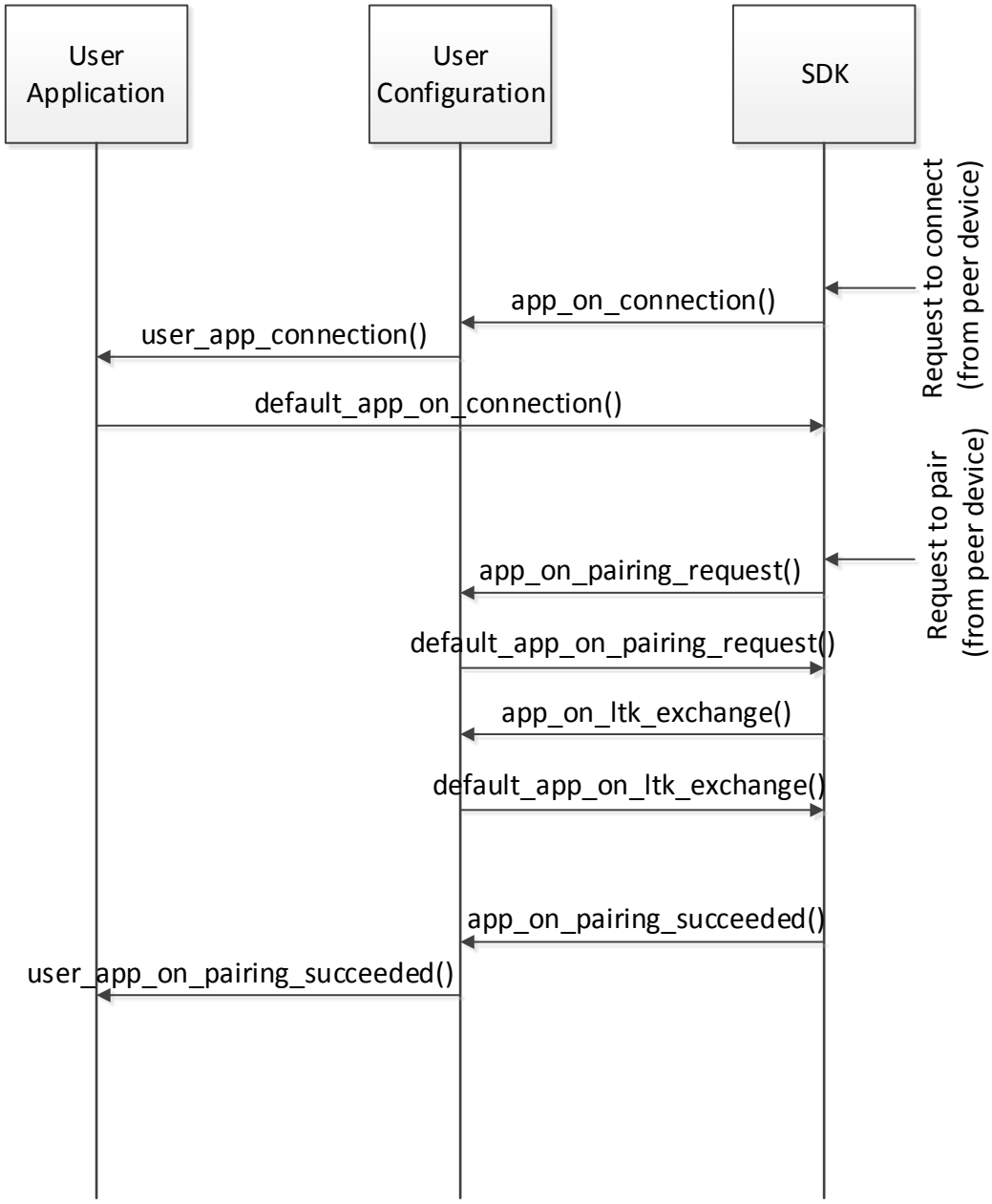


Figure 44: Pillar 4 Application - User Application Code Flow for Pairing using Just Works

8.5.9 Building the Project for Different Targets and Development Kits

The Pillar 4 application can be built for two different target processors: DA14585 and DA14586. The selection is done via the Keil tool as depicted in Figure 45.

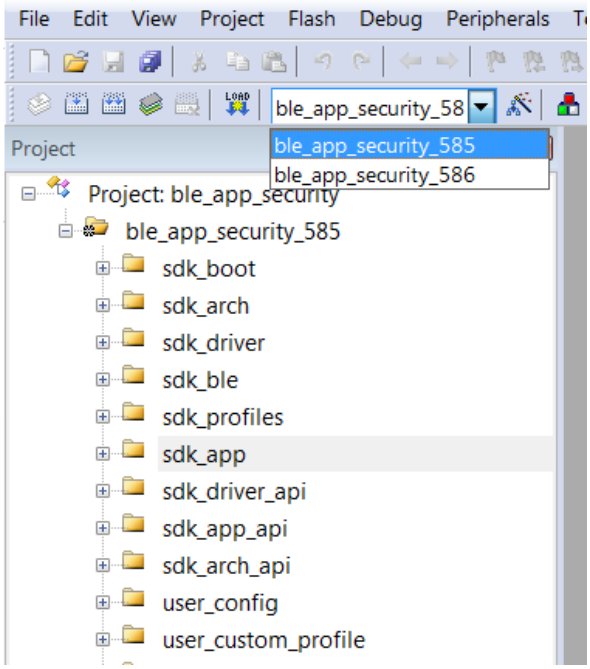


Figure 45: Building the Project for Different Targets

The user also has to select the correct Development Kit in order to build and run the application. This selection is done via the Configuration Wizard of the `user_periph_setup.h` file. See Figure 46.

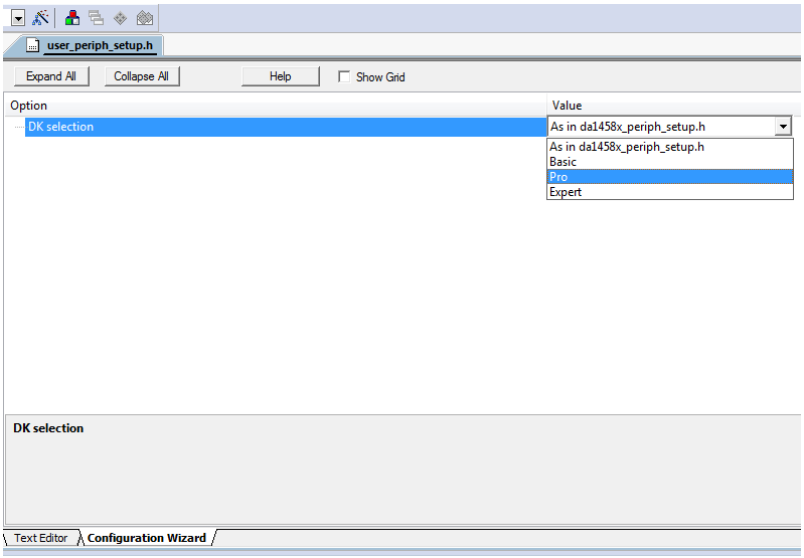


Figure 46: Development Kit Selection for Pillar 4 Application

After the proper selection of the target processor and development kit, the application is ready to be built.

8.5.10 Interacting with BLE Application
8.5.11 LightBlue iOS

The LightBlue iOS application can be used to connect an iPad/iPod/iPhone device to the application. In such a case the iPad/iPod/iPhone acts as a BLE Central and the application as a BLE Peripheral. Figure 47 shows the result when the iPad/iPod/iPhone device manages to connect to the DA14585/586 (the application's advertising device name is **DIALOG-SECURITY**). Depending on the currently selected security setup you can be asked to allow devices to pair or to enter passkey. By default the Pillar 4 application is using **123456** as the passkey value.

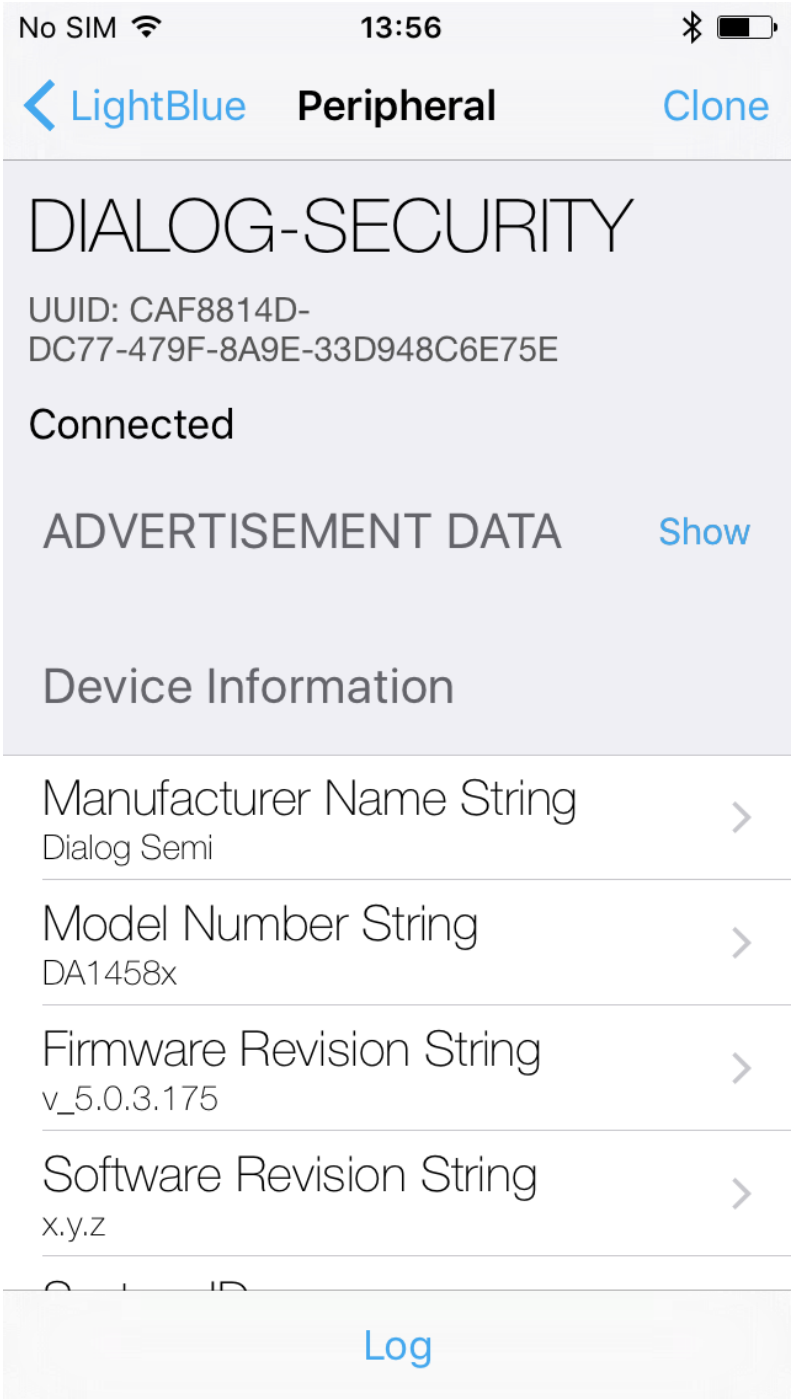


Figure 47: LightBlue Application Connected to Pillar 4 Application

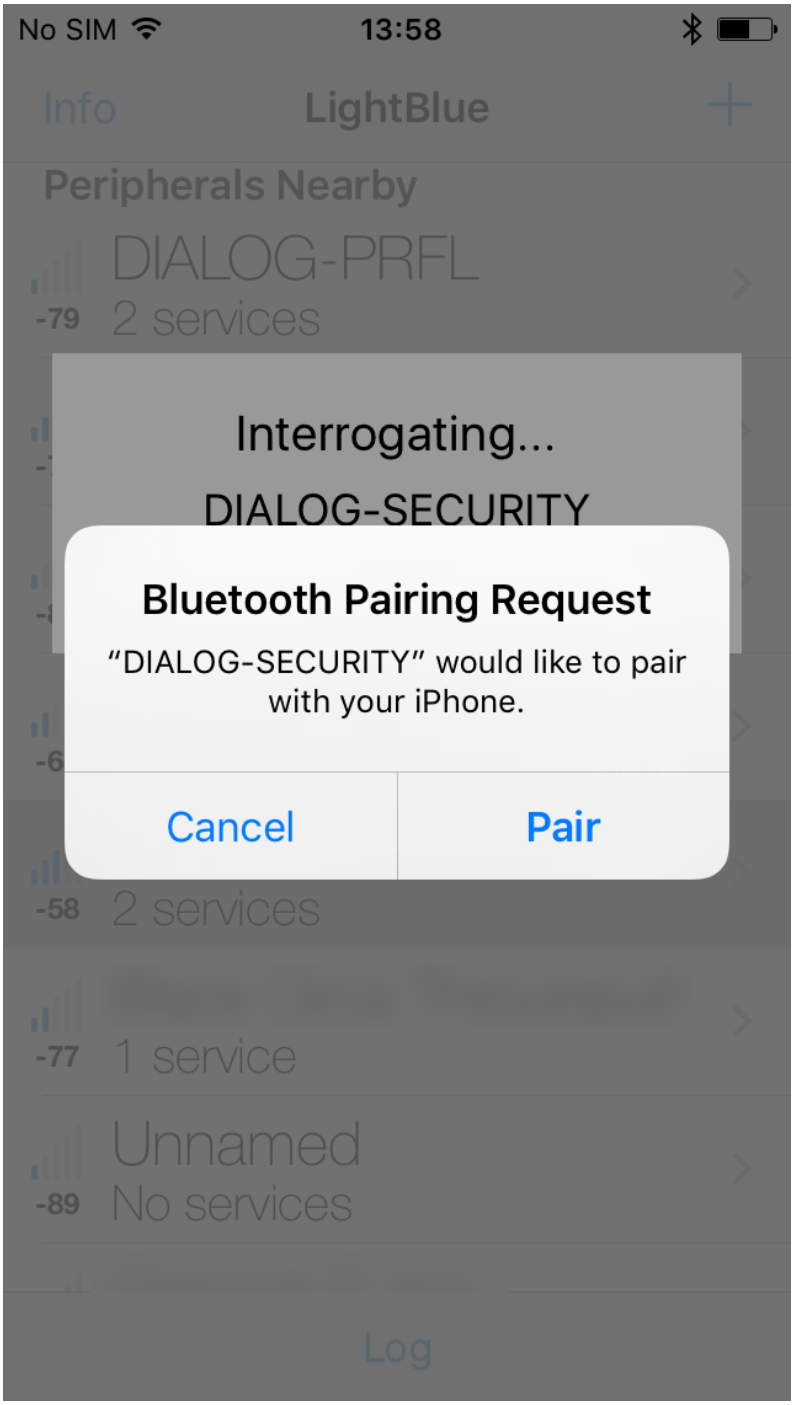


Figure 48: LightBlue Application Pairing with Pillar 4 Application using Just Works

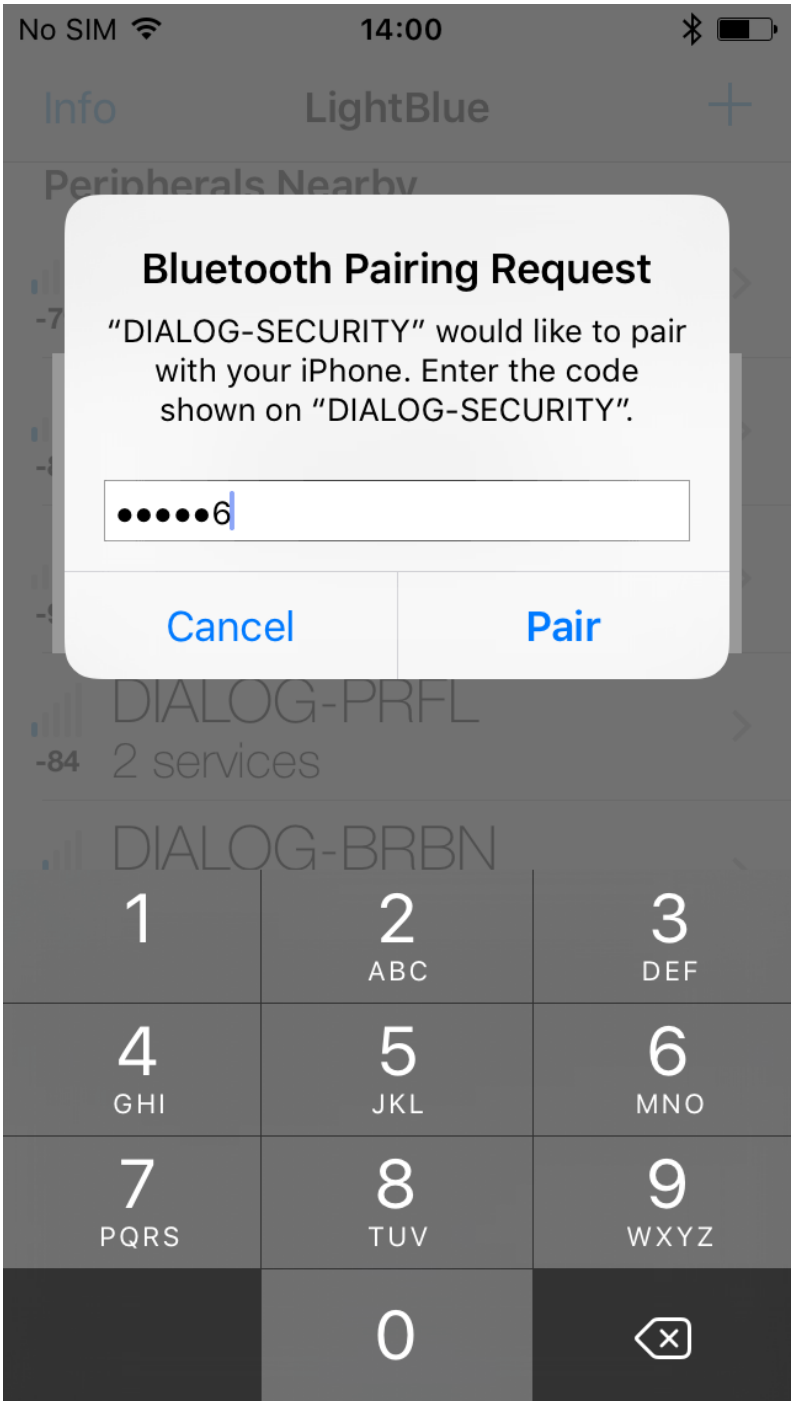


Figure 49: LightBlue Application Pairing with Pillar 4 Application Using Passkey with MITM

For Pillar 4 application the default passkey is set to **123456.**

8.6 Pillar 5 (Sleep Mode)

8.6.1 Application Description

The Pillar 5 (sleep mode) BLE example application demonstrates the same as the Pillar 2 application. The application also adds some basic interaction over the provided custom service (read/write/notify values). The main purpose of this application example is to show how to use the sleep mode API and change in runtime the sleep mode. The two sleep modes that are used are:

- Extended sleep without OTP copy.
- Extended sleep with OTP copy.

In **Extended sleep without OTP copy**, the application code is stored in a memory source other than the OTP memory. After wake-up from sleep, no OTP mirror, no boot from SPI, I2C, UART interface shall happen. The RAM blocks which include code will be retained. The RAM blocks which include retention data will be retained. RAM block 4 is always retained (BLE state).This mode resembles to the extended sleep mode of the DA14580 chip.

In **Extended sleep with OTP copy**, the application code is stored in OTP memory. After wake-up from sleep, the OTP mirror will take place. The RAM blocks which include code will not be retained. The RAM blocks which include retention data will be retained. RAM block 4 is always retained (BLE state). This mode resembles to the deep sleep mode of the DA14580 chip.

As in the SDK of the DA14580 chip, the user can preselect to use either the Extended sleep mode without OTP copy or the Extended sleep mode with OTP copy, by providing the respective sleep configuration in the user_config.h file.

```
/*****************************************
 * Default sleep mode. Possible values are:
 *
 * - ARCH_SLEEP_OFF
 * - ARCH_EXT_SLEEP_ON
 * - ARCH_EXT_SLEEP_OTP_COPY_ON
 *
 *****************************************
 */
```

One of the above sleep modes will be used when the system logic decides that the DA14585 chip is ready to sleep.

For more information about Sleep modes refer to section 7.1.1 of [5].

8.6.2 Basic Operation

Supported services:

- Inherits the services from Pillar 2 application.

Features:

- Inherits the features from Pillar 2 application, plus:
- CONTROL POINT, ADC VAL 2 and BUTTON STATE characteristic values introduce some behavior with a connected peer device.

The Pillar 5 application behavior is included in C source file `user_sleepmode.c`.

Table 16 shows the Custom service characteristic values along with their properties.

Table 16. Pillar 5 Custom Service Characteristic Values and Properties

Name	Properties	Length (B)	Description/Purpose
CONTROL POINT	WRITE	1	Accept commands from peer
LED STATE	WRITE NO RESPONSE	1	Toggles a LED connected to a GPIO
ADC VAL 1	READ, NOTIFY	2	Reads sample from an ADC channel
ADC VAL 2	READ	2	Reads sample from an ADC channel
BUTTON STATE	READ, NOTIFY	1	Reads the current state of a push button connected a GPIO

Name	Properties	Length (B)	Description/Purpose
INDICATEABLE CHAR	READ, INDICATE	20	Demonstrate indications
LONG VAL CHAR	READ, WRITE. NOTIFY	50	Demonstrate writes to long characteristic value

The Pillar 5 application provides behavior only for the highlighted characteristic values of the Custom service. The implementation code of the Custom service is included in C source file `user_sleepmode_task.c`.

8.6.3 User Interface

When the device is powered up or disconnected:

- Device advertises for a defined amount of time (`APP_ADV_DATA_UPDATE_TO`), default value is 10 s. As long as the device is in the advertising state its sleep mode is set to Extended sleep with OTP copy (the OTP copy is emulated when the system runs in `DEVELOPMENT_DEBUG` mode).
- After the expiration of the above timeout, and if the device does not enter the connected state, it stops advertising. Now the device does nothing and waits for an external event to exit the sleeping state.
- The user can wake up the device by pressing a button. After the button press the device will start to advertise again for the predefined time.
- When the device enters the connected state then the sleep mode is turned to Extended sleep without OTP copy.

A peer connected to the Pillar 5 application is able to do the following:

- Use Custom Service.
- Write to **Control Point** of Custom Service.
  - Byte 0x00 disables **PWM** timer (turns LED off and restores sleep mode).
  - Byte 0x01 enables **PWM** timer (turns LED on and puts device into active mode) – user can attach speaker to selected pin to hear PWM frequency change.
  - Byte 0x02 disables **ADC VAL 2** update.
  - Byte 0x03 enables **ADC VAL 2** update. **ADC VAL 2** characteristic is updated at every connection event.
- Read/notify **BUTTON STATE** value. When the button is pressed or released characteristic value is updated accordingly.
- Read **ADC VAL 2**. When functionality is enabled in **Control Point** user can read value from ADC input.

8.6.4 Loading the Project

The Pillar 5 application is developed under the Keil v5 tool. The Keil project name is the:

`<sdk_root_directory>\projects\target_apps\ble_examples\ble_app_sleepmode\Keil_5\ble_app_sleepmode.uvprojx`

Figure 50 shows the Keil project layout with emphasis on the user related files, included in the Keil project folders `user_config`, `user_platform`, `user_custom_profile` and `user_app`. These folders contain the user configuration files of the Pillar 5 application.

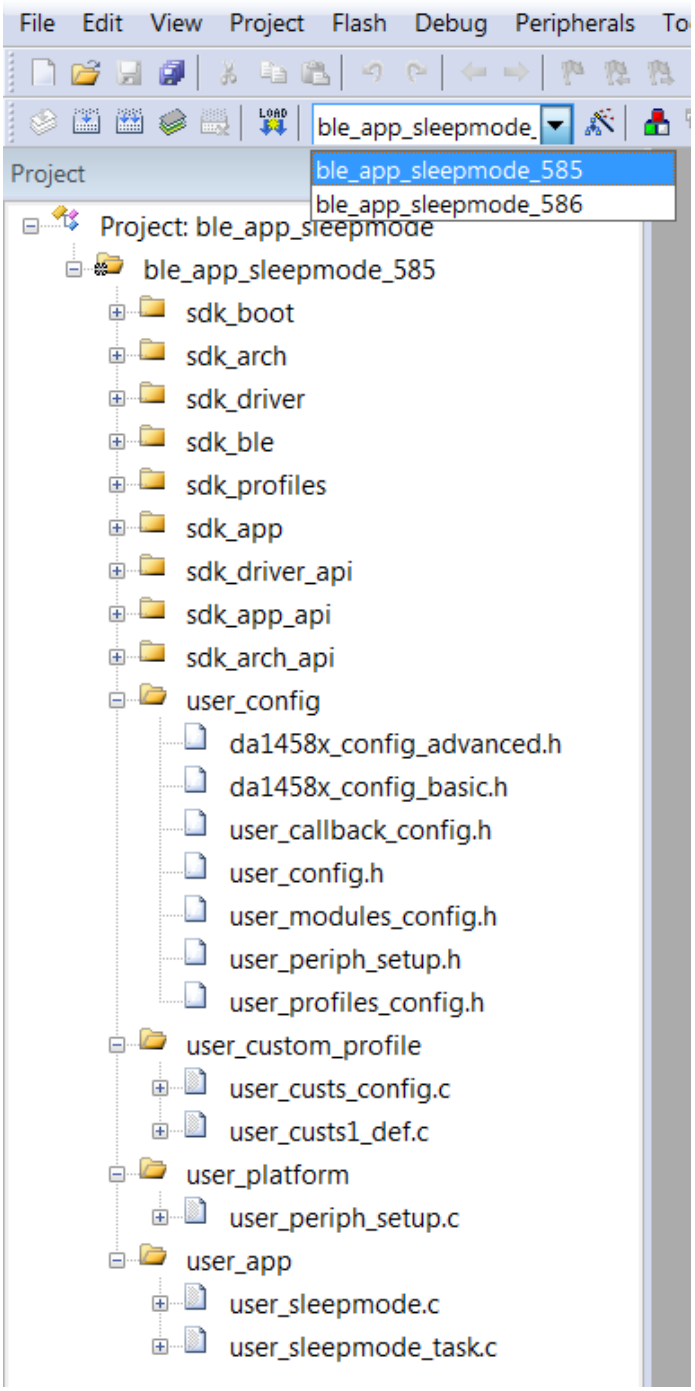


Figure 50: Pillar 5 Keil Project Layout

8.6.5 Going Through the Code
8.6.6 Initialization

The aforementioned Keil project folders (`user_config`, `user_platform`, `user_custom_profile` and `user_app`) contain the files that initialize and configure the Pillar 5 application.

- `da1458x_config_advanced.h`, holds DA14585/586 advanced configuration settings.
- `da1458x_config_basic.h`, holds DA14585/586 basic configuration settings.
- `user_callback_config.h`, callback functions that handle various events or operations.
- `user_config.h`, holds advertising parameters, connection parameters, etc.
- `user_config_sw_ver.h`, holds user specific information about software version.
- `user_modules_config.h`, defines which application modules are included or excluded from the user's application. For example:
  - #define EXCLUDE_DLG_DISS (0), the Device information application profile is included. The SDK takes care of the Device information application profile message handling.
  - #define EXCLUDE_DLG_DISS (1), the Device information application profile is excluded. The user application has to take care of the Device information application profile message handling.
- `user_profiles_config.h`, defines which BLE profiles (Bluetooth SIG adopted or custom ones) will be included in user's application. Particularly, the C header files (each header file denotes the respective BLE profile) that are included in the `user_profile_config.h` file are:
  - `CFG_PRF_DISS`, includes the Device Information service.
  - `CFG_PRF_CUST1`, includes the Custom 1 service.
- `user_custs1_def.c`, defines the structure of the Custom 1 profile database structure.
- `user_custs_config.c`, defines the cust_prf_funcs[] array, which contains the Custom profiles API functions calls.
- `user_periph_setup.h`, holds hardware related settings relative to the used Development Kit.
- `user_periph_setup.c`, source code file that handles peripheral (GPIO, UART, etc.) configuration and initialization relative to the Development Kit.

8.6.7 Events Processing and Callbacks

Several events can occur during the lifetime of the application. It depends on the application which of these events are handled and how. The SDK is flexible enough to either call a default handler or call the user's defined event handler upon the occurrence of a particular event. The configuration file `user_callback_config.h` contains the configuration array that defines if an event is processed or not (callback function is present or not). For example, in the Pillar 5 application the user_app_callbacks[] array has the following entries:

```
static const struct app_callbacks user_app_callbacks = {
    .app_on_connection              = user_app_connection,
    .app_on_disconnect              = user_app_disconnect,
    .app_on_update_params_rejected  = NULL,
    .app_on_update_params_complete  = NULL,
    .app_on_set_dev_config_complete = default_app_on_set_dev_config_complete,
    .app_on_adv_nonconn_complete    = NULL,
    .app_on_adv_undirect_complete   = user_app_adv_undirect_complete,
    .app_on_adv_direct_complete     = NULL,
    .app_on_db_init_complete        = default_app_on_db_init_complete,
    .app_on_scanning_completed      = NULL,
    .app_on_adv_report_ind          = NULL,
    .app_on_get_dev_appearance      = default_app_on_get_dev_appearance,
    .app_on_get_dev_slv_pref_params = default_app_on_get_dev_slv_pref_params,
    .app_on_set_dev_info            = default_app_on_set_dev_info,
    .app_on_data_length_change      = NULL,
```



```
    .app_on_update_params_request   = default_app_update_params_request,
    #if (BLE_APP_SEC)
    .app_on_pairing_request         = NULL,
    .app_on_tk_exch_nomitm          = NULL,
    .app_on_irk_exch                = NULL,
    .app_on_csrk_exch               = NULL,
    .app_on_ltk_exch                = NULL,
    .app_on_pairing_succeded        = NULL,
    .app_on_encrypt_ind             = NULL,
    .app_on_mitm_passcode_req       = NULL,
    .app_on_encrypt_req_ind         = NULL,
    .app_on_security_req_ind        = NULL,
    #endif // (BLE_APP_SEC)
};
```

The above array defines that a certain event will be processed by a default handler or by a user defined handler or it will not be processed at all (NULL entries). The user defined handlers (e.g user_app_connection(), user_app_disconnect() and user_app_adv_undirect_complete()) are defined in C source file user_sleepmode.c.

Similar to Pillar 3 project Pillar 5 also defines user_catch_rest_hndl() handler that catches all the Custom service messages and handles them in the user application space. The implementation of this handler is in C source file user_sleepmode_task.c.

An important addition to Pillar 5 application is the app_button_enable()function, which is called from the user_app_adv_undirect_complete() event handler. After the advertising is completed it configures one of the user buttons as a wake up trigger, just before the application goes to sleep. The registered wakeup callback function app_button_press_cb() is set to restore the BLE core stack and peripherals back to fully functional state.

The `user_callback_config.h` configuration header file contains the registration of the callback function user_catch_rest_hndl(), as is described below.

```
static const catch_rest_event_func_t app_process_catch_rest_cb =
(catch_rest_event_func_t)user_catch_rest_hndl;
```

The `user_callback_config.h` configuration header file contains the registration of the callback function user_app_on_wakeup(), as is described below.

```
static const  struct arch_main_loop_callbacks user_app_main_loop_callbacks = {
    .app_on_init            = user_app_init,
    .app_on_ble_powered     = NULL,
    .app_on_system_powered  = NULL,
    .app_before_sleep       = NULL,
    .app_validate_sleep     = NULL,
    .app_going_to_sleep     = NULL,
    .app_resume_from_sleep  = NULL,
};
```

8.6.8 BLE Application Abstract Code Flow

Figure 51 shows the abstract code flow diagram of the Pillar 5 application. The diagram depicts the SDK interaction with the callback functions registered in `user_callback_config.h` and the functions implemented in `user_sleepmode.c`. It shows only the part that is new, compared to the Pillar 2 application. The connection establishment procedure is the exactly the same as in previous application and the following function flow diagram shows the platform entering sleep mode and the wakeup calls triggered by the button press. Advertising is restarted after wakeup.

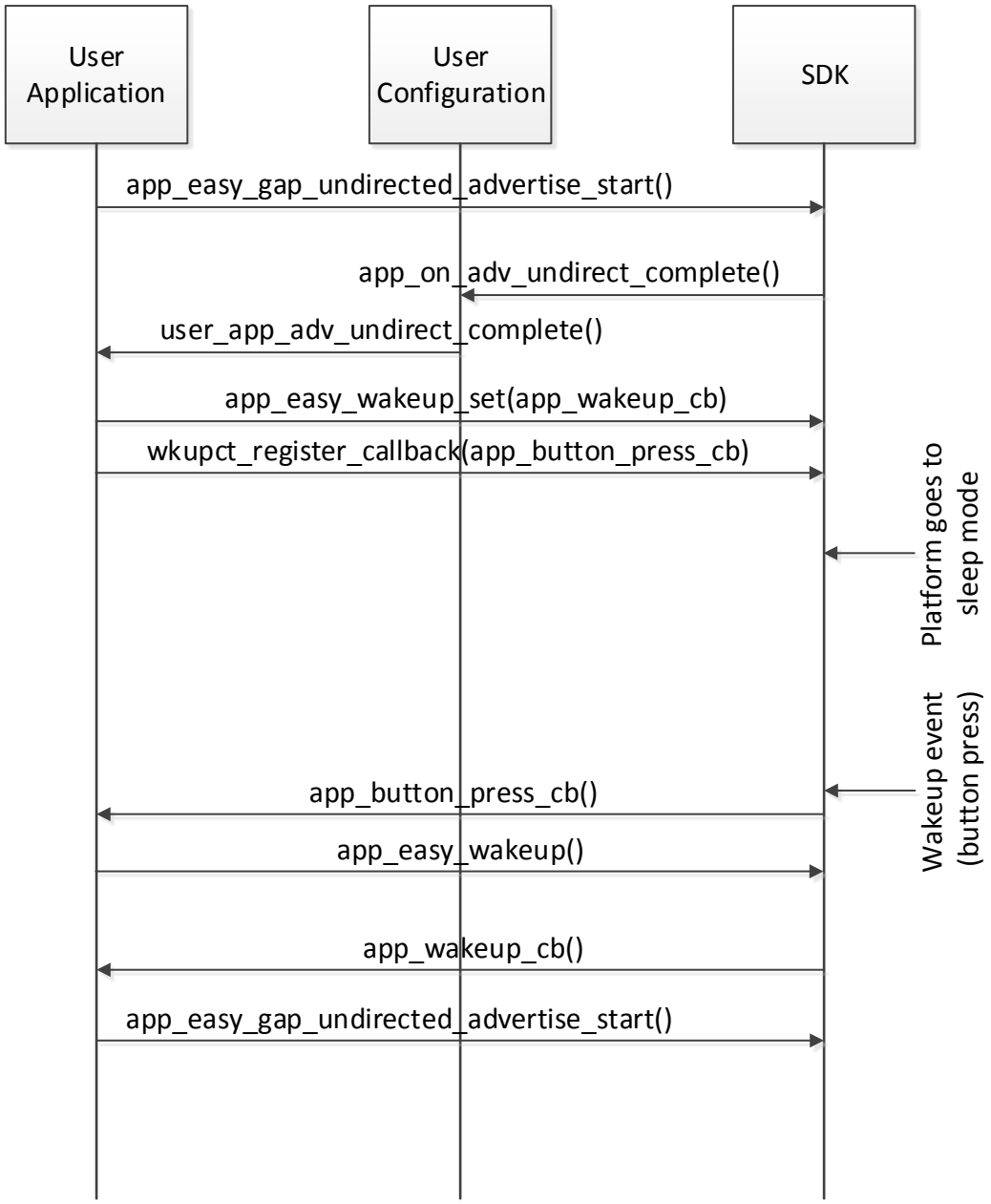


Figure 51: Pillar 5 Application - User Application Code Flow

8.6.9 Building the Project for Different Targets and Development Kits

The Pillar 5 application can be built for two different target processors: DA14585 and DA14586. The selection is done via the Keil tool as depicted in Figure 52.

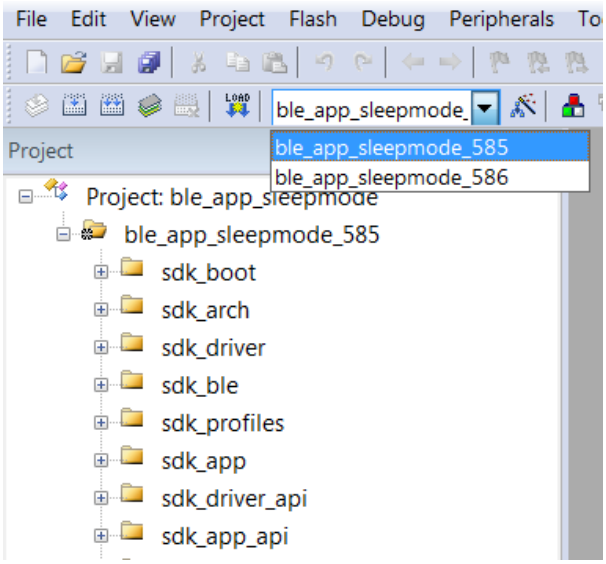


Figure 52: Building the Project for Different Targets

The user has also to select the correct Development Kit in order to build and run the application. This selection is done via the Configuration Wizard of the `user_periph_setup.h` file. See Figure 53.

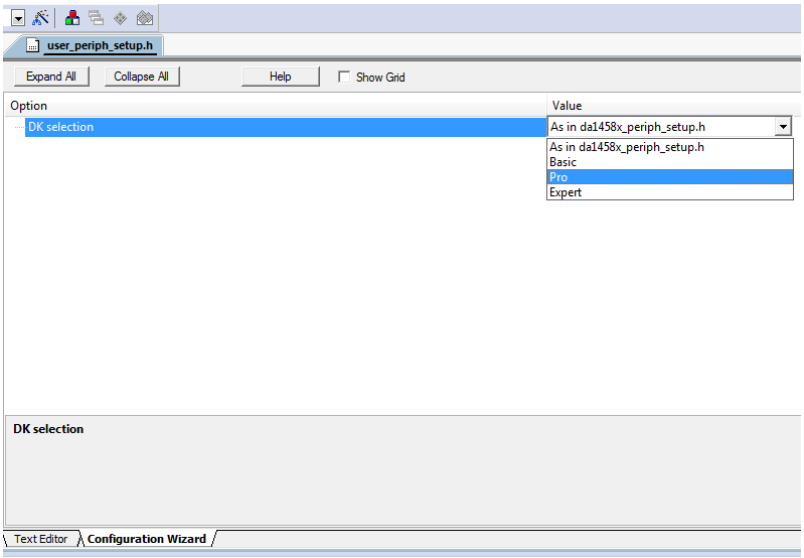


Figure 53: Development Kit Selection for Pillar 5 Application

After the proper selection of the target processor and development kit, the application is ready to be built.

8.6.10 Interacting with BLE Application

8.6.11 LightBlue iOS

The LightBlue iOS application can be used to connect an iPad/iPod/iPhone device to the application. In such a case the iPad/iPod/iPhone acts as a BLE Central and the application as a BLE Peripheral. Figure 54 shows the result when the iPad/iPod/iPhone device manages to connect to the DA14585/586 (the application's advertising device name is **DIALOG-SLEEP**).

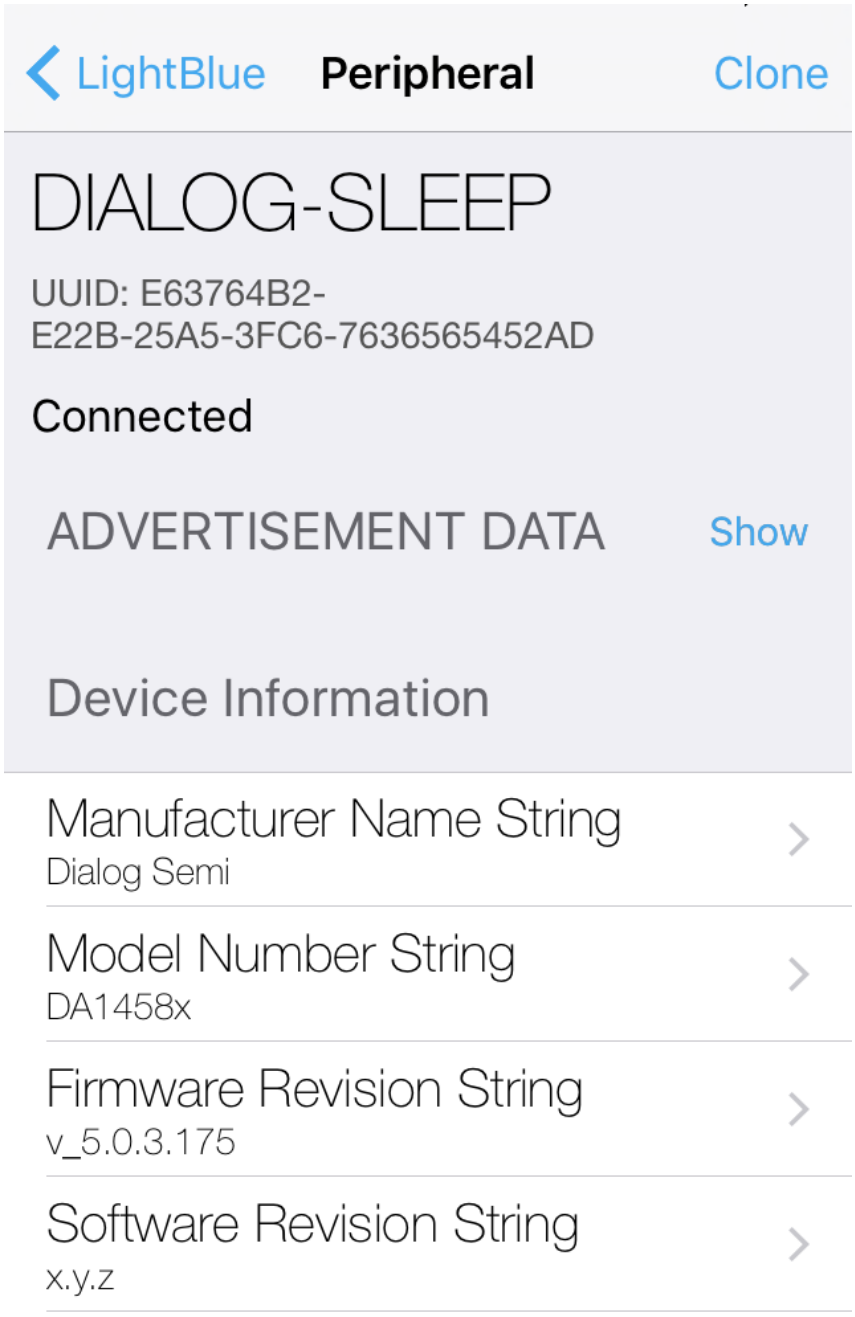


Figure 54: LightBlue Application Connected to Pillar 5 Application

8.7 Pillar 6 (OTA)

The Pillar 6 (OTA) BLE example application demonstrates the same as the Pillar 2 application, plus the SUOTAR service (16-bit UUID). The SUOTAR service requires no action on user space side and can be used by peer device for Software Updates Over The Air (SUOTA). The project uses the "Integrated processor" configuration.

8.7.1 Basic Operation

Supported services:

- Inherits the services from Pillar 2 application.
- SUOTAR service defined by the user with 16-bit UUID.

Features:

- Inherits the features from Pillar 2 application, plus:
- SUOTA firmware updates to external SPI/I2C memories

The Pillar 6 behavior is included in C source file `user_ota.c`.

Table 17 shows the Custom service characteristic values along with their properties.

Table 17: Pillar 6 Custom Service Characteristic Values and Properties

Name	Properties	Length (B)	Description/Purpose
CONTROL POINT	WRITE	1	Accept commands from peer
LED STATE	WRITE NO RESPONSE	1	Toggles a LED connected to a GPIO
ADC VAL 1	READ, NOTIFY	2	Reads sample from an ADC channel
ADC VAL 2	READ	2	Reads sample from an ADC channel
BUTTON STATE	READ, NOTIFY	1	Reads the current state of a push button connected a GPIO
INDICATEABLE CHAR	READ, INDICATE	20	Demonstrate indications
LONG VAL CHAR	READ, WRITE. NOTIFY	50	Demonstrate writes to long characteristic value

The Pillar 6 application does not provide any behavior for the Custom service.

Table 18: Pillar 6 SUOTAR Service Characteristic Values and Properties

Name	Properties	Length (B)	Description/Purpose
SUOTA MEMORY DEVICE	READ, WRITE	4	Defines what is the target physical memory and base address of the patch.
GPIO MAP	READ, WRITE	4	Port and pin map for the physical memory device as well as device address in case of I2C EEPROMs.
MEMORY INFORMATION	READ	4	Number of bytes transferred.
SUOTA PATCH LENGTH	READ, WRITE	2	Block length of SUOTA image to be sent at a time.
SUOTA DATA	READ, WRITE, WRITE NO RESPONSE	20	20 bytes of SUOTA data, word aligned. MS byte first.
SUOTA SERVICE STATUS	READ, NOTIFY	1	SUOTA Service Status.

8.7.2 User Interface

A peer connected to the Pillar 6 application is able to do the same as in the Pillar 2 application, plus:

- Update firmware image over a Bluetooth Low Energy link. The detailed procedure description as well as image preparation process can be found in 9.

8.7.3 Loading the Project

The Pillar 6 application is developed under the Keil v5 tool. The Keil project file is the:

`<sdk_root_directory>\projects\target_apps\ble_examples\ble_app_ota\Keil_5\ble_app_ota.uvprojx`

Figure 55 shows the Keil project layout with emphasis on the user related files, included in the Keil project folders `user_config`, `user_platform`, `user_custom_profile` and `user_app`. These folders contain the user configuration files of the Pillar 6 application.

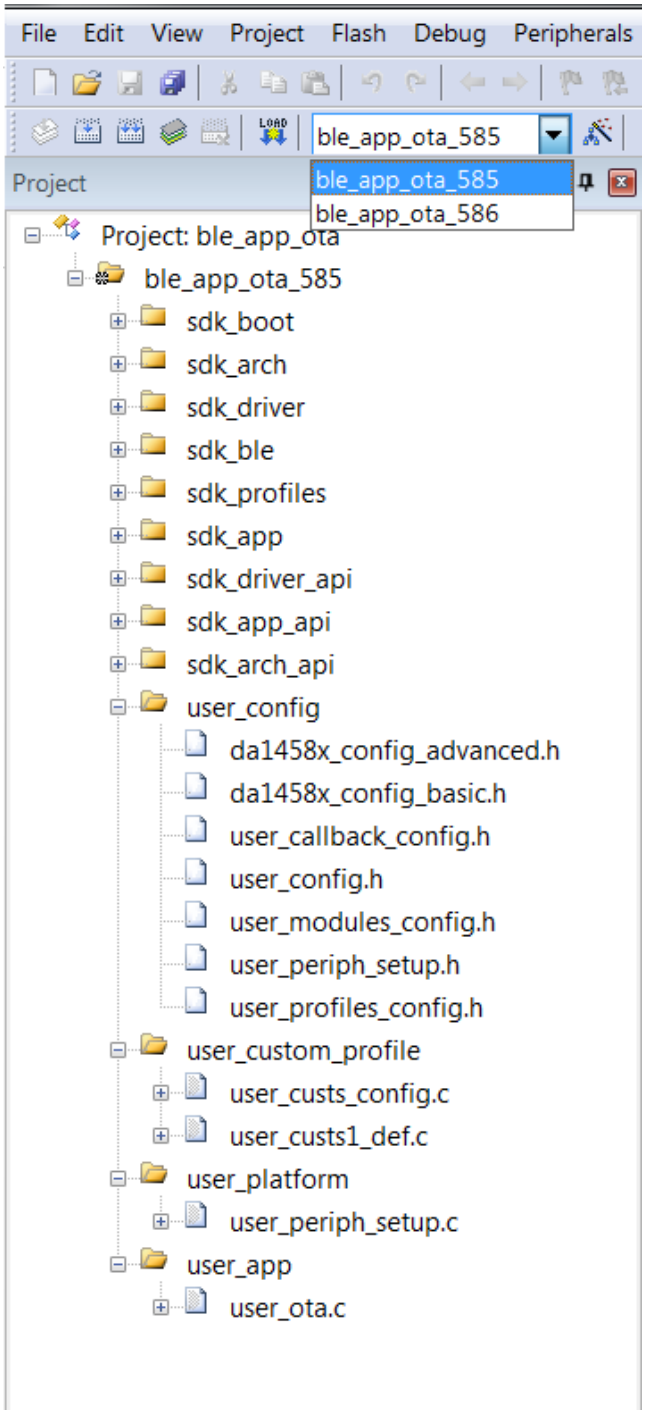


Figure 55: Pillar 6 Keil Project Layout

There is no new file added compared to the Pillar 2 project.

8.7.4 Going Through the Code
8.7.5 Initialization

The aforementioned Keil project folders (`user_config`, `user_platform`, `user_custom_profile` and `user_app`) contain the files that initialize and configure the Pillar 6 application.

- `da1458x_config_advanced.h`, holds DA14585/586 advanced configuration settings.
- `da1458x_config_basic.h`, holds DA14585/586 basic configuration settings.
- `user_callback_config.h`, callback functions that handle various events or operations.
- `user_config.h`, holds advertising parameters, connection parameters, etc.
- `user_config_sw_ver.h`, holds user specific information about software version.
- `user_modules_config.h`, defines which application modules are included or excluded from the user's application. For example:
  - #define EXCLUDE_DLG_DISS (0), the Device information application profile is included. The SDK takes care of the Device information application profile message handling.
  - #define EXCLUDE_DLG_DISS (1), the Device information application profile is excluded. The user application has to take care of the Device information application profile message handling.
- `user_profiles_config.h`, defines which BLE profiles (Bluetooth SIG adopted or custom ones) will be included in user's application. Particularly, the C header files (each header file denotes the respective BLE profile) that are included in the `user_profile_config.h` file are:
  - `CFG_PRF_DISS`, includes the Device Information service.
  - `CFG_PRF_SUOTAR`, includes the SUOTAR service.
  - `CFG_PRF_CUST1`, includes the Custom 1 service.
- `It also exposes some configuration flags for the SUOTAR service:`
  - #define CFG_SUOTAR_I2C_DISABLE , Disable I2C external memory module
  - #define CFG_SUOTAR_SPI_DISABLE , Disable SPI external memory module
- `user_custs1_def.c`, defines the structure of the Custom 1 profile database structure.
- `user_custs_config.c`, defines the cust_prf_funcs[] array, which contains the Custom profiles API functions calls.
- `user_periph_setup.h`, holds hardware related settings relative to the used Development Kit.
- `user_periph_setup.c`, source code file that handles peripheral (GPIO, UART, etc.) configuration and initialization relative to the Development Kit.

8.7.6 Events Processing and Callbacks

Several events can occur during the lifetime of the BLE application and these events need to be handled in a specific manner. Also, operations need to be served depending on the application scenario. It depends on the application itself to define which events and operations are handled and how. The SDK is flexible enough to either call a default handler or call the user's defined event or operation handler.

The SDK mechanism that takes care of the above requirements, is the registration of callback functions for every event or operation. The C header file `user_callback_config.h`, which resides in user space, contains the registration of the callback functions.

The Pillar 6 application registers the following callback functions:

- General BLE events:

```
static const struct app_callbacks user_app_callbacks = {
    .app_on_connection              = user_app_connection,
    .app_on_disconnect              = user_app_disconnect,
    .app_on_update_params_rejected  = NULL,
    .app_on_update_params_complete  = NULL,
```



```
    .app_on_set_dev_config_complete = default_app_on_set_dev_config_complete,
    .app_on_adv_nonconn_complete    = NULL,
    .app_on_adv_undirect_complete   = user_app_adv_undirect_complete,
    .app_on_adv_direct_complete     = NULL,
    .app_on_db_init_complete        = default_app_on_db_init_complete,
    .app_on_scanning_completed      = NULL,
    .app_on_adv_report_ind          = NULL,
    .app_on_get_dev_appearance      = default_app_on_get_dev_appearance,
    .app_on_get_dev_slv_pref_params = default_app_on_get_dev_slv_pref_params,
    .app_on_set_dev_info            = default_app_on_set_dev_info,
    .app_on_data_length_change      = NULL,
    .app_on_update_params_request   = default_app_update_params_request,
    #if (BLE_APP_SEC)
    .app_on_pairing_request         = default_app_on_pairing_request,
    .app_on_tk_exch_nomitm          = default_app_on_tk_exch_nomitm,
    .app_on_irk_exch                = NULL,
    .app_on_csrk_exch               = default_app_on_csrk_exch,
    .app_on_ltk_exch                = default_app_on_ltk_exch,
    .app_on_pairing_succeded        = NULL,
    .app_on_encrypt_ind             = NULL,
    .app_on_mitm_passcode_req       = NULL,
    .app_on_encrypt_req_ind         = default_app_on_encrypt_req_ind,
    .app_on_security_req_ind        = NULL,
    #endif // (BLE_APP_SEC)
};
```

The above structure defines that a certain event will be processed by a default handler or by a user defined handler or it will not be processed at all (NULL entries). The user defined handlers (e.g. user_app_connection(), user_app_disconnect() and user_app_adv_undirect_complete()) are defined in C source file `user_ota.c`.

- System specific events:

```
static const  struct arch_main_loop_callbacks user_app_main_loop_callbacks = {
    .app_on_init            = user_app_init,
    .app_on_ble_powered     = NULL,
    .app_on_sytem_powered   = NULL,
    .app_before_sleep       = NULL,
    .app_validate_sleep     = NULL,
    .app_going_to_sleep     = NULL,
    .app_resume_from_sleep  = NULL,
};
```

The above structure defines that a certain event will be processed by a default handler or by a user defined handler or it will not be processed at all (NULL entries). The user defined handlers (e.g. user_app_init()) is defined in C source file `user_ota.c`.

- BLE operations:

```
static const struct default_app_operations user_default_app_operations = {
    .default_operation_adv = user_app_adv_start,
};
```

The above structure defines that a certain operation will be processed by a default handler or by a user defined handler or it will not be processed at all (NULL entries). The user defined handlers (e.g. user_app_adv_start()) is defined in C source file `user_ota.c`.

- Custom profile message handling:

```
static const catch_rest_event_func_t app_process_catch_rest_cb =
(catch_rest_event_func_t)user_catch_rest_hndl;
```

Callback function that contains the Custom profile messages handling in user application space. For Pillar 6 application this function is totally unused, since Pillar 6 application does not include any behavior related to the Custom service.

8.7.7 BLE Application Abstract Code Flow

The code flow of the functions implemented in `user_ota.c` for the Pillar 6 application is the same as shown in Figure 33 for the Pillar2 application. The whole OTA functionality requires no custom user application code to be written.

8.7.8 Building the Project for Different Targets and Development Kits

The Pillar 6 application can be built for two different target processors: DA14585 and DA14586. The selection is done via the Keil tool as depicted in Figure 56.

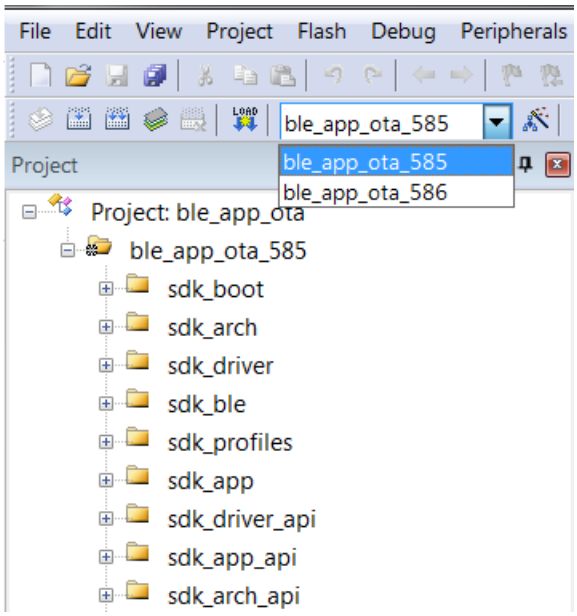


Figure 56: Building the Project for Different Targets

The user has also to select the correct Development Kit in order to build and run the application. This selection is done via the Configuration Wizard of the `user_periph_setup.h` file. See Figure 57.

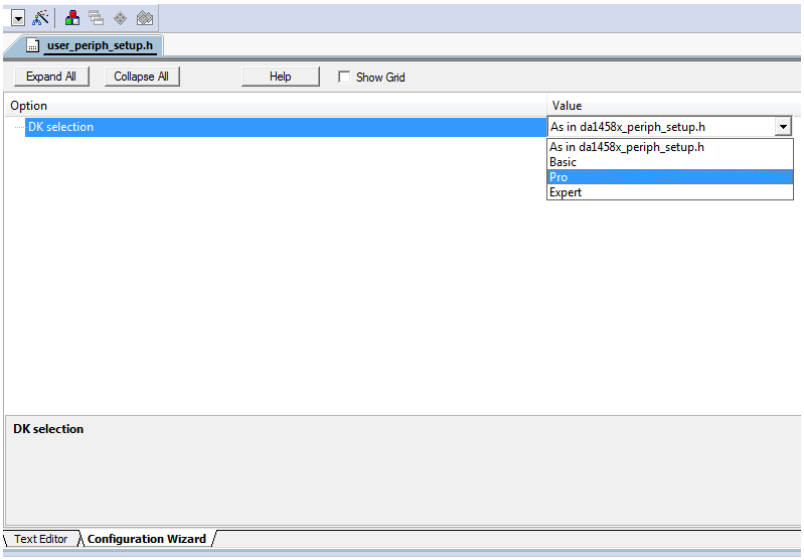


Figure 57: Development Kit Selection for Pillar 6 Application

After the proper selection of the target processor and development kit, the application is ready to be built.

8.7.9 Interacting with BLE Application

8.7.10 LightBlue iOS

The LightBlue iOS application can be used to connect an iPad/iPod/iPhone device to the application. In such a case the iPad/iPod/iPhone acts as a BLE Central and the application as a BLE Peripheral. The following picture shows the result when the iPad/iPod/iPhone device manages to connect to the DA14585/586 (the application's advertising device name is **DIALOG-OTA**).

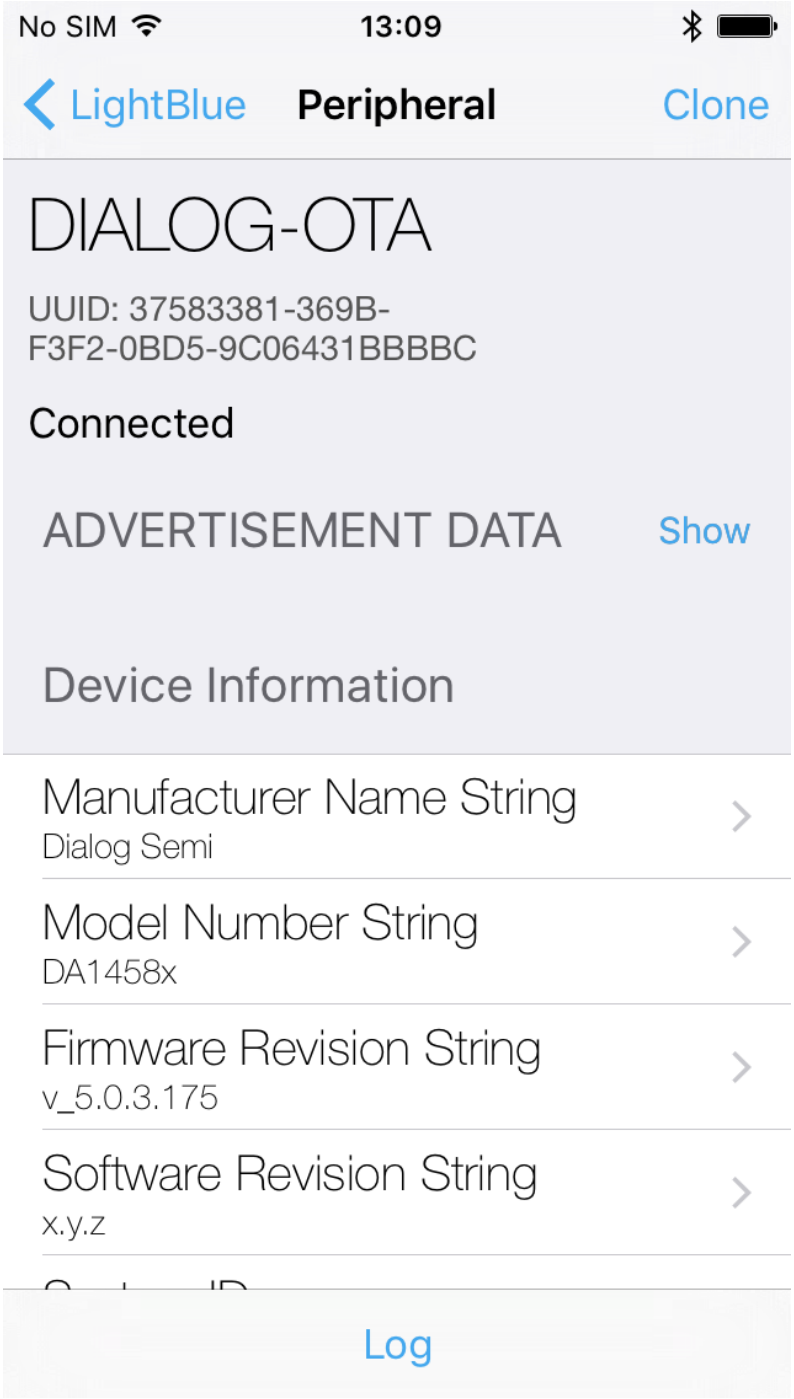


Figure 58: LightBlue Application Connected to Pillar 6 Application

8.7.11 SUOTA Application

The Dialog SUOTA application can be installed from the App Store on iOS devices and from the Google Play Store on Android based platforms. Only devices that are advertising the SUOTAR service are shown to the user. The detailed SUOTA update procedure is described in 9.

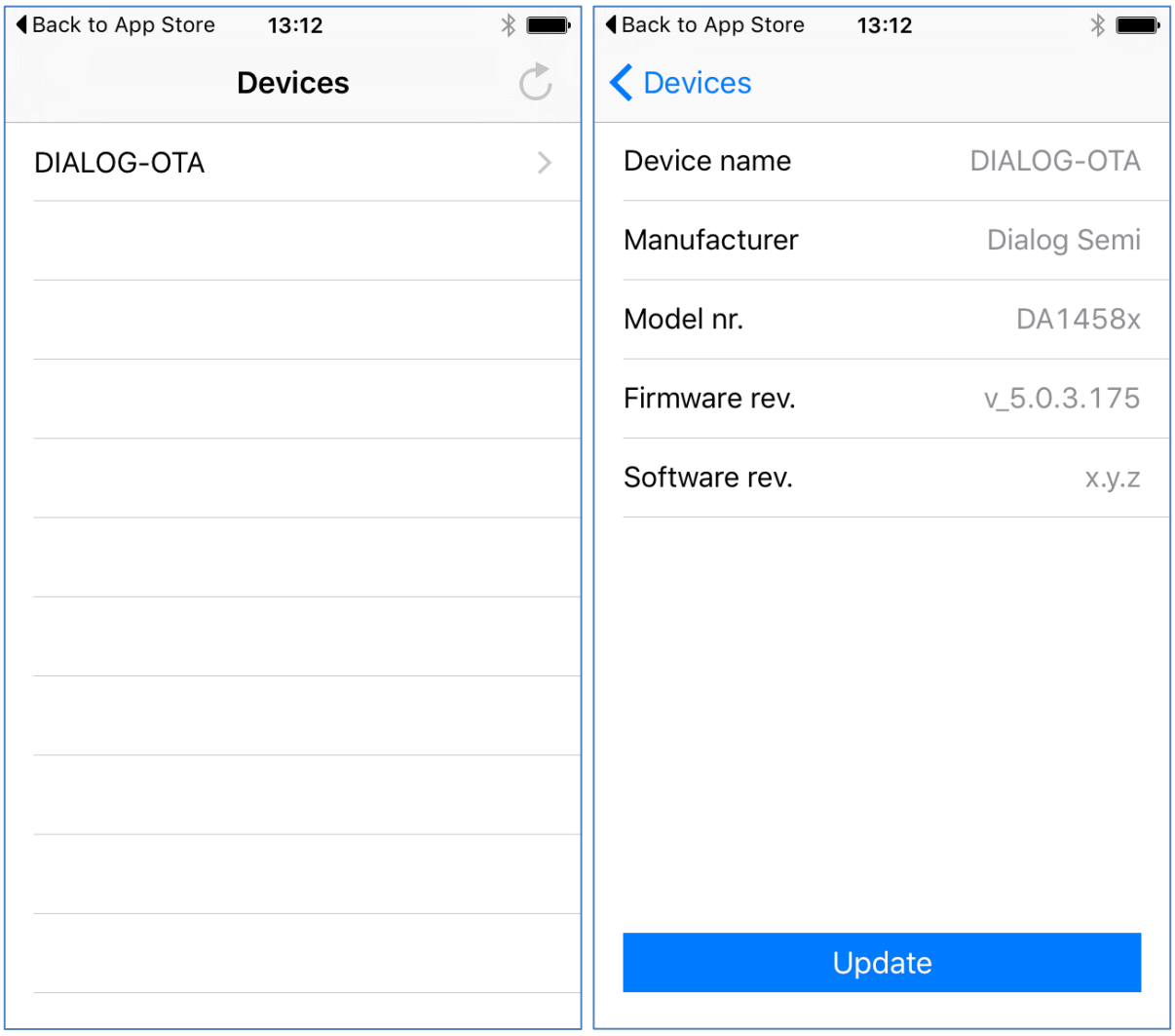


Figure 59: Dialog SUOTA Application Discovering Pillar 6 Application

8.8 Pillar 7 (All in One)

8.8.1 Application Description

The Pillar 7 (All in One) BLE example application demonstrates the same functionality as all previous applications. The project uses the "Integrated processor" configuration.

8.8.2 Basic Operation

Supported services:

- Inherits the services from Pillar 2 application.
- Inherits the SUOTAR service from Pillar 6.

Features:

- Inherits the features from Pillar 2 (Custom Profile) application.
- Inherits the features from Pillar 3 (Peripheral) application.
- Inherits the features from Pillar 4 (Security) application.
- Inherits the features from Pillar 5 (Sleep) application.
- Inherits the SUOTA functionality from Pillar 6 application.

The Pillar 7 behavior is included in C source file `user_all_in_one.c`.

Table 19 shows the Custom service characteristic values along with their properties.

Table 19: Pillar 7 Custom Service Characteristic Values and Properties

Name	Properties	Length (B)	Description/Purpose
CONTROL POINT	WRITE	1	Accept commands from peer
LED STATE	WRITE NO RESPONSE	1	Toggles a LED connected to a GPIO
ADC VAL 1	READ, NOTIFY	2	Reads sample from an ADC channel
ADC VAL 2	READ	2	Reads sample from an ADC channel
BUTTON STATE	READ, NOTIFY	1	Reads the current state of a push button connected a GPIO
INDICATEABLE CHAR	READ, INDICATE	20	Demonstrate indications
LONG VAL CHAR	READ, WRITE. NOTIFY	50	Demonstrate writes to long characteristic value

The Pillar 7 application provides the same behavior for the Custom Service as Pillar 3 (Peripheral) and Pillar 5 (Sleep). The implementation code of the Custom service is included in C source file `user_custs1_impl.c`.

Table 20: Pillar 7 SUOTAR Service Characteristic Values and Properties

Name	Properties	Length (B)	Description/Purpose
SUOTA MEMORY DEVICE	READ, WRITE	4	Defines what is the target physical memory device and base address of the patch.
GPIO MAP	READ, WRITE	4	Port and pin map for the physical memory device as well as device address in case of I2C EEPROMs.
MEMORY INFORMATION	READ	4	Number of bytes transferred.
SUOTA PATCH LENGTH	READ, WRITE	2	Block length of SUOTA image to be sent at a time.
SUOTA PATCH DATA	READ, WRITE, WRITE NO RESPONSE	20	20 bytes of SUOTA data, word aligned. MS byte first.
SUOTA SERVICE STATUS	READ, NOTIFY	1	SUOTA Service Status.

Pillar 7 provides the same SUOTA functionality as Pillar 6.

8.8.3 User Interface

A peer connected to the Pillar 7 application is able to do the same as in the Pillars 2, 3, 4, 5, plus:

- Perform software updates as in the Pillar 6 application.

8.8.4 Loading the Project

The Pillar 7 application is developed under the Keil v5 tool. The Keil project file is the:

`<sdk_root_directory>\projects\target_apps\ble_examples\ble_app_all_in_one\Keil_5\ble_app_ota.uvprojx`

Figure 60 shows the Keil project layout with emphasis on the user related files, included in the Keil project folders `user_config`, `user_platform`, `user_custom_profile` and `user_app`. These folders contain the user configuration files of the Pillar 7 application.

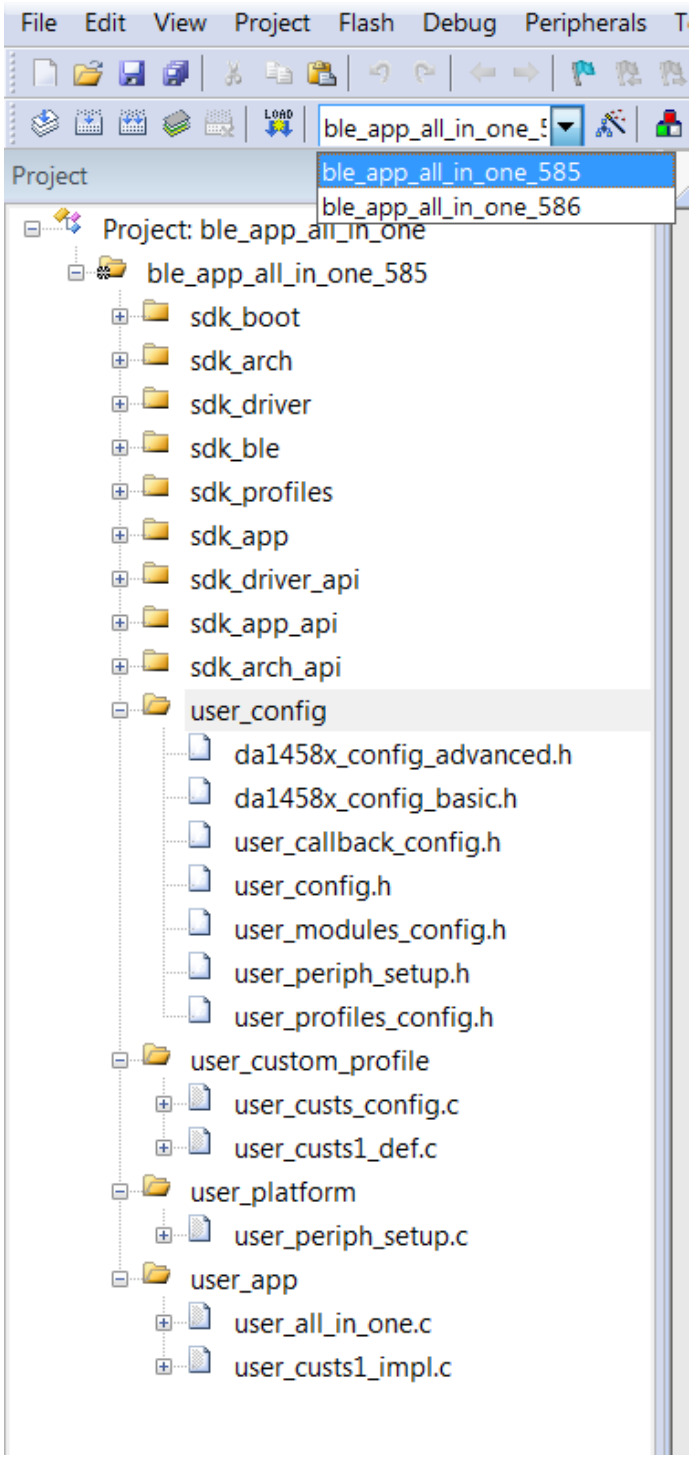


Figure 60: Pillar 7 Keil Project Layout

Just like in Pillar 3 the `user_custs1_impl.c` contain the implementation code of the Custom service.

8.8.5 Going Through the Code
8.8.6 Initialization

The aforementioned Keil project folders (`user_config`, `user_platform`, `user_custom_profile` and `user_app`) contain the files that initialize and configure the Pillar 7 application.

- `da1458x_config_advanced.h`, holds DA14585/586 advanced configuration settings.
- `da1458x_config_basic.h`, holds DA14585/586 basic configuration settings.
- `user_callback_config.h`, callback functions that handle various events or operations.
- `user_config.h`, holds advertising parameters, connection parameters. It also contains all the security configuration defines that the Pillar 4 (Security) application was using. For example:
  - #define USER_CFG_PAIR_METHOD_JUST_WORKS, the device is using the Just Works pairing method.
  - #define USER_CFG_PAIR_METHOD_PASSKEY, the device is using Pass Key pairing method.
  - #define USER_CFG_PAIR_METHOD_OOB, the device is using the Out of Band (OOB) pairing method.
  - Note: At the time of writing this document, neither Android nor iOS support the Out of Band (OOB) mechanism for Bluetooth pairing.
  - The user can define one of the above pairing methods, if the application requires it. If none of the above flag is defined, then the security features are turned off.
- This configuration header file allows also for selecting Privacy Feature of the peripheral device. This feature allows the device to use random addresses to prevent peers from tracking it. Privacy feature is selected through the following two flags. For example:
  - #define USER_CFG_PRIV_GEN_STATIC_RND, the device is using a random address generated automatically by the BLE stack. This address is static during device's power cycle.
  - #define USER_CFG_PRIV_GEN_RSLV_RND, the device is using a resolvable random address, generated automatically by the BLE stack. This address is changing in certain time intervals. Only bonded devices that own the Identity Resolving Key, distributed during the pairing procedure, can resolve the Random Address and track the device.
  - If none of the above flags is selected the device is not using any Privacy Feature, and will use its public address.
  - Peer device's bond data can be stored on an external SPI Flash or I2C EEPROM memory.
  - #define USER_CFG_APP_BOND_DB_USE_SPI_FLASH, for SPI Flash.
  - #define USER_CFG_APP_BOND_DB_USE_I2C_EEPROM, for I2C EEPROM.
  - If none of the above flags is defined the bond data have to be stored in the application RAM.
- `user_config_sw_ver.h`, holds user specific information about software version.
- `user_modules_config.h`, defines which application modules are included or excluded from the user's application. For example:
  - #define EXCLUDE_DLG_DISS (0), the Device information application profile is included. The SDK takes care of the Device information application profile message handling.
  - #define EXCLUDE_DLG_DISS (1), the Device information application profile is excluded. The user application has to take care of the Device information application profile message handling.
- `user_profiles_config.h`, defines which BLE profiles (Bluetooth SIG adopted or custom ones) will be included in user's application. Particularly, the C header files (each header file denotes the respective BLE profile) that are included in the `user_profile_config.h` file are:
  - `CFG_PRF_DISS`, includes the Device Information service.
  - `CFG_PRF_SUOTAR`, includes the SUOTAR service.

- CFG_PRF_CUST1, includes the Custom 1 service.

- It also exposes some configuration flags for the SUOTAR service:
  - #define CFG_SUOTAR_I2C_DISABLE , Disable I2C external memory module
  - #define CFG_SUOTAR_SPI_DISABLE , Disable SPI external memory module
- user_custs1_def.c, defines the structure of the Custom 1 profile database structure.
- user_custs_config.c, defines the cust_prf_funcs[] array, which contains the Custom profiles API functions calls.
- user_periph_setup.h, holds hardware related settings relative to the used Development Kit. In this particular application it also defines the I2C pin configuration for the EEPROM module.
- user_periph_setup.c, source code file that handles peripheral (GPIO, UART, etc.) configuration and initialization relative to the Development Kit.

8.8.7 Events Processing and Callbacks

Several events can occur during the lifetime of the BLE application and these events need to be handled in a specific manner. Also, operations need to be served depending on the application scenario. It depends on the application itself to define which events and operations are handled and how. The SDK is flexible enough to either call a default handler or call the user's defined event or operation handler.

The SDK mechanism that takes care of the above requirements, is the registration of callback functions for every event or operation. The C header file user_callback_config.h, which resides in user space, contains the registration of the callback functions.

The Pillar 7 application registers the following callback functions:

- General BLE events:

```
static const struct app_callbacks user_app_callbacks = {
    .app_on_connection              = user_app_connection,
    .app_on_disconnect              = user_app_disconnect,
    .app_on_update_params_rejected  = NULL,
    .app_on_update_params_complete  = NULL,
    .app_on_set_dev_config_complete = default_app_on_set_dev_config_complete,
    .app_on_adv_nonconn_complete    = NULL,
    .app_on_adv_undirect_complete   = user_app_adv_undirect_complete,
    .app_on_adv_direct_complete     = NULL,
    .app_on_db_init_complete        = default_app_on_db_init_complete,
    .app_on_scanning_completed      = NULL,
    .app_on_adv_report_ind          = NULL,
    .app_on_get_dev_appearance      = default_app_on_get_dev_appearance,
    .app_on_get_dev_slv_pref_params = default_app_on_get_dev_slv_pref_params,
    .app_on_set_dev_info            = default_app_on_set_dev_info,
    .app_on_data_length_change      = NULL,
    .app_on_update_params_request   = default_app_update_params_request,
    #if (BLE_APP_SEC)
    .app_on_pairing_request         = default_app_on_pairing_request,
    .app_on_tk_exch_nomitm          = user_app_on_tk_exch_nomitm,
    .app_on_irk_exch                = NULL,
    .app_on_csrk_exch               = NULL,
    .app_on_ltk_exch                = default_app_on_ltk_exch,
    .app_on_pairing_succeded        = user_app_on_pairing_succeded,
    .app_on_encrypt_ind             = NULL,
    .app_on_mitm_passcode_req       = NULL,
    .app_on_encrypt_req_ind         = user_app_on_encrypt_req_ind,
    .app_on_security_req_ind        = NULL,
    #endif // (BLE_APP_SEC)
};
```

The above structure defines that a certain event will be processed by a default handler or by a user defined handler or it will not be processed at all (NULL entries). The user defined handlers (e.g user_app_connection(), user_app_disconnect(), user_app_adv_undirect_complete(), user_app_on_tk_exch_nomitm(), user_app_on_pairing_succeded() and user_app_on_encrypt_req_ind()) are defined in C source file `user_all_in_one.c`.

- System specific events:

```
static const  struct arch_main_loop_callbacks user_app_main_loop_callbacks = {
    .app_on_init            = user_app_init,
    .app_on_ble_powered     = NULL,
    .app_on_sytem_powered   = NULL,
    .app_before_sleep       = NULL,
    .app_validate_sleep     = NULL,
    .app_going_to_sleep     = NULL,
    .app_resume_from_sleep  = NULL,
};
```

The above structure defines that a certain event will be processed by a default handler or by a user defined handler or it will not be processed at all (NULL entries). The user defined handlers (e.g. user_app_init()) is defined in C source file `user_all_in_one.c`.

- BLE operations:

```
static const struct default_app_operations user_default_app_operations = {
    .default_operation_adv = user_app_adv_start,
};
```

The above structure defines that a certain operation will be processed by a default handler or by a user defined handler or it will not be processed at all (NULL entries). The user defined handlers (e.g. user_app_adv_start()) is defined in C source file `user_all_in_one.c`.

- Custom profile message handling:

```
static const catch_rest_event_func_t app_process_catch_rest_cb =
(catch_rest_event_func_t)user_catch_rest_hndl;
```

Callback function that contains the Custom profile messages handling in user application space. For the Pillar 7 application this function is handling the same write or read request as the Pillar 3 (Peripheral) and Pillar 5 (Sleep) applications do.

8.8.8 BLE Application Abstract Code Flow

Figure 61 shows the abstract code flow diagram of the Pillar 7 application in a simplified form. The diagram depicts the SDK interaction with the callback functions registered in `user_callback_config.h` and the functions implemented in `user_all_in_one.c`. The initialization sequence is the same as in the Pillar 1 application and is not shown on this diagram.

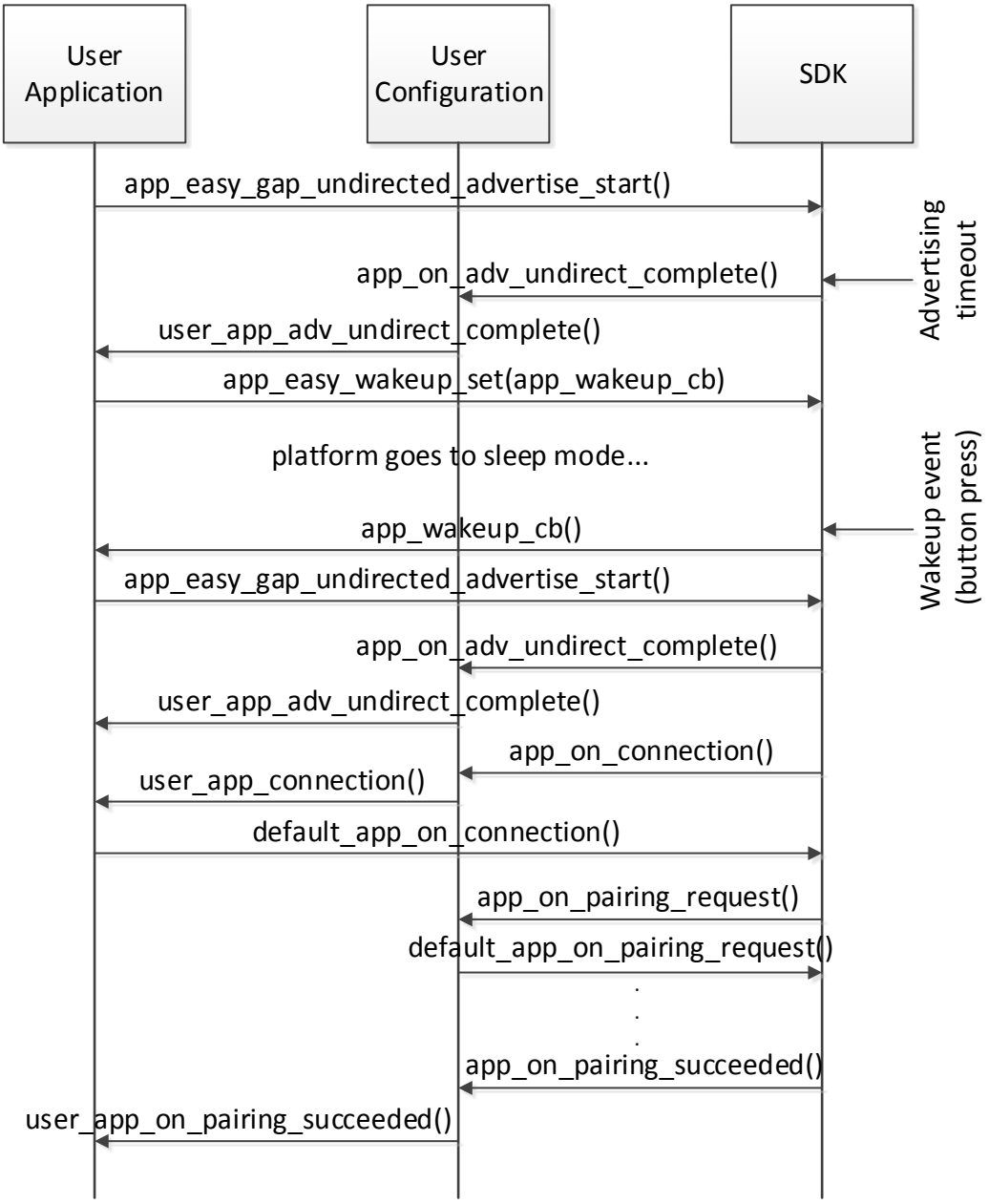


Figure 61: Pillar 7 Application - User Application Simplified Code Flow

8.8.9 Building the Project for Different Targets and Development Kits

The Pillar 7 application can be built for two different target processors: DA14585 and DA14586. The selection is done via the Keil tool as depicted in Figure 62.

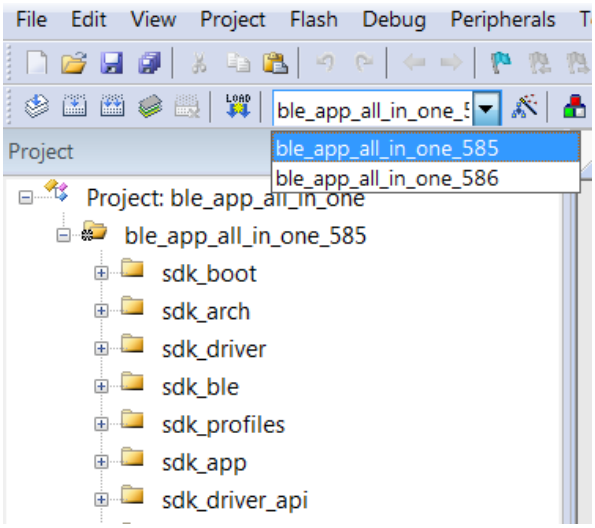


Figure 62: Building the project for different targets

The user has also to select the correct Development Kit in order to build and run the application. This selection is done via the Configuration Wizard of the `user_periph_setup.h` file. See Figure 63.

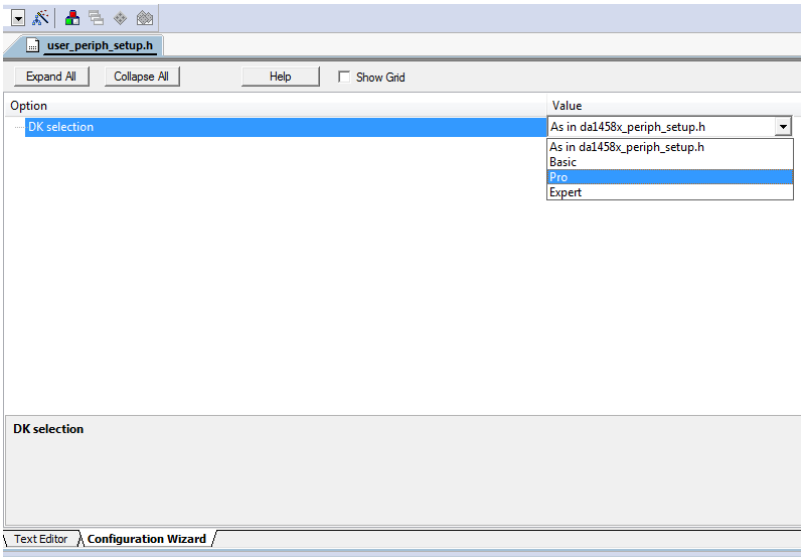


Figure 63: Development Kit Selection for the Pillar 7 Application

After the proper selection of the target processor and development kit, the application is ready to be built.

8.8.10 Interacting with BLE Application
8.8.11 LightBlue iOS

The LightBlue iOS application can be used to connect an iPad/iPod/iPhone device to the application. In such a case the iPad/iPod/iPhone acts as a BLE Central and the application as a BLE Peripheral. Figure 64 shows the result when the iPad/iPod/iPhone device manages to connect to the DA14585/586 (the application's advertising device name is **DIALOG-ALL-IN**). Please note that during the device interrogation you can be asked to accept pairing or enter passkey. It depends on security settings that are currently defined in `user_config.h`.

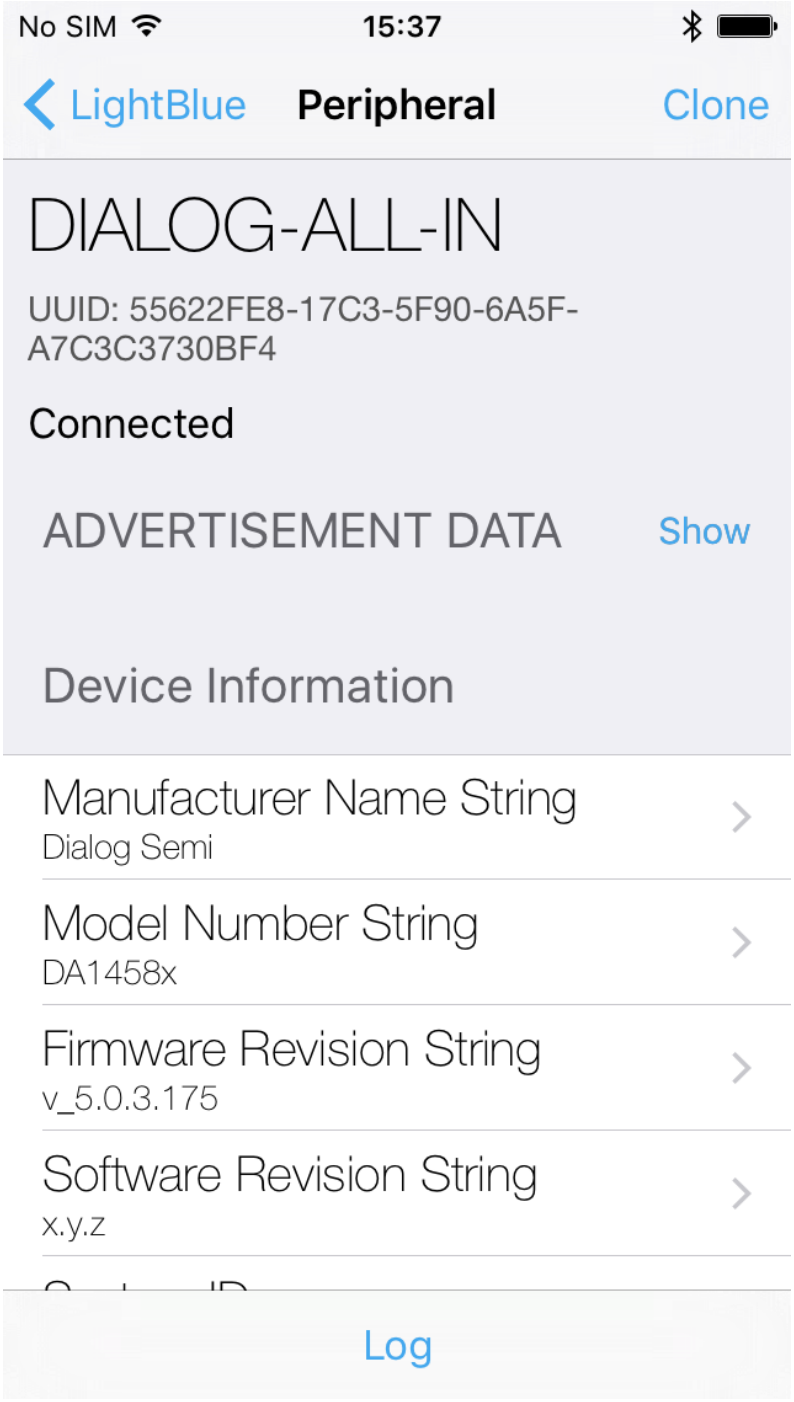


Figure 64: LightBlue Application Connected to Pillar 7 Application

9 Software Upgrade

9.1 Software Update Over The Air (SUOTA)

9.1.1 Introduction

The BLE platform allows the user to update the software of the device wirelessly. This process is called Software Update Over The Air (SUOTA), and is simple enough to be performed by the end user.

This section describes how to update the image on a DA14585/586 device that supports Dialog's Software Update Over The Air (SUOTA) proprietary service.

In this section, the following project will be used:

`<sdk_root_directory>\projects\target_apps\ble_examples\prox_reporter`

The DA14585/586 SDK provides software (for both Central & Peripheral roles) for software update over the air (SUOTA). Over the air update refers simply to a software update that is distributed over a Bluetooth Smart link.

Figure 65: SUOTA feature

This functionality can be achieved by using the SUOTA service.

Table 21: SUOTA profile enabled location in the SW

Parameter	Variable/macro	Source file
SUOTA profile	`#include "suotar.h"`	`user_profiles_config.h`

The SUOTAR profile is implemented in the following files:

`sdk\ble_stack\profiles\suota\suotar\suotar.c`

`sdk\ble_stack\profiles\suota\suotar\suotar_task.c`

To run SUOTA, a non-volatile memory (SPI flash or I2C EEPROM) must be hooked up to the DA14585/586.

SUOTA is instantiated as a GATT Primary Service.

The service exposes a control point to allow a peer device to initiate software update over the air and define two roles:

- The "SUOTA Initiator" which transmits the new software image. It is the GATT client for the SUOTA service (GAP Central Role)
- The "SUOTA Receiver" which receives the new software image, stores the image into the external FLASH/EEPROM device and runs the new image. It is the GATT server for SUOTA service (GAP Peripheral Role).

The proximity reporter (internal processor solution) uses the SUOTA profile to:

- Receive a new software image that is sent by the Central over the Bluetooth Smart link.
- Validate the new image and send informative status updates to the Central.
- Store the new image into an external non-volatile memory (FLASH/EEPROM devices).
- Restart the system.

A dual image bootloader detects and executes the active (latest valid) image. For more information about the secondary bootloader, please refer to [6].

Two different schemes are provided when SUOTA is used:

- **Scheme 1**: The secondary bootloader is stored in the external non-volatile memory.
- **Scheme 2:** The secondary bootloader is burnt into the internal OTP.

The main differences between these 2 schemes are outlined in the table below:

Table 22: Pros and cons of the different schemes

	Scheme 1	Scheme 2
Advantages	OTP can stay blank Useful for development purposes and/or when very low power consumption is not a requirement for the final product.	Fastest boot-up time. Guarantee to boot up anytime.
Disadvantages	In case the external memory is in power down mode and a software reset (*e.g.*: Watchdog) is triggered, the DA14585/586 will not boot up properly. The battery has to be removed and replaced.	OTP must be burnt.

9.2 Security, external memory and product header parameters

9.2.1 Security when SUOTA is used

The best solution to ensure that an application cannot be hacked or bricked by somebody is to:

- Use the bonding procedure straight after the connection is established
- A physical action on the device is needed to delete the bonding information.

The bonding device information is stored in an external non-volatile memory. A push button is used to delete the current bonding information. When no bonding information is stored in the external memory, the device can bond with a new central.

By default, the SUOTA profile uses the `Security Mode 1 Level 1: No security` level. This can be easily changed in the `app_suotar_enable()` function by adding the following code:

```
#if (BLE_APP_SEC)
  req->sec_lvl = PERM(SVC, UNAUTH); // Security enabled
#else
  req->sec_lvl = PERM(SVC, ENABLE);
#endif // BLE_APP_SEC
```

Figure 66: Security of the service enabled

The security modes and levels which can be used are outlined below:

- Security Mode 1 Level 1: No security (default mode used when SUOTA profile is activated)
- Security Mode 1 Level 2: Unauthenticated pairing with encryption
- Security Mode 1 Level 3: Authenticated pairing with encryption

The table below describes where the security levels can be changed in the software.

Table 23: Different security levels defined in the SW

Parameter	Variable/macro	Source file
Different security levels	gapm_write_att_perm	gapm_task.h

9.2.2 Choice of the external memory to store the images

The choice of the external memory can be easily selected from the secondary boot loader project as outlined below. By default, a SPI memory type is defined in the `bootloader.h` file.

Table 24: Choice of the external memory

Parameter	Variable/macro	Source file
Use of the SPI Flash	SPI_FLASH_SUPPORTED	bootloader.h
Use of the I2C EEPROM	EEPROM_FLASH_SUPPORTED	bootloader.h

For more information about how to handle external memories, please refer to [7].

9.2.3 Address of the product header

By default, the address of the product header is at `0x1F000`. In case, the address of the product header has to be changed, two files as outlined below must be changed.

Table 25: Address of the product header defined in the secondary bootloader project

Parameter	Variable/macro	Source file
Address of the product header	PRODUCT_HEADER_POSITION	bootloader.h
Address of the product header	PRODUCT_HEADER_POSITION	app_suotar.h

IMPORTANT NOTE
The product header can be stored at any addresses. By default, it is stored at the address `0x1F000` of the external flash. This can be easily changed as shown below. For more info about the product header, see the section 5 or 6.

9.3 Hardware needed

9.3.1 Central side

- The tablets and phones which are running iOS.
- The tablets and phones which are running Android with following versions:
  - Android devices that run version 5.0.0 and above always stall during the image upload.

- Android devices that run 4.4.4 work fine.
- The SUOTA app is not very stable with devices that run 4.4.2. It works most of the time, but stalling problems during image upload have been seen.

9.3.2 Peripheral side

9.3.2.1 Using Dialog's reference designs

- For the Beacon reference design, please refer to [8], section 4.
- For the Proximity Tag reference design, please refer to [9], section 5.

9.3.2.2 Using Dialog's PRO Development Kit

The picture below shows the right jumper's positions to program the external flash via the JTAG interface. This jumper setting will allow the DA14585/586 to boot from the external flash too.

Figure 67: development kit HW configuration

- For more information about the pro reference design, please refer to:http://support.dialog-semiconductor.com/resource/pro-all-documents-development-kit-pro-manual-gerber-bom-schematics

9.3.2.3 Using Dialog's BASIC Development Kit

The picture below shows the right jumper position to program the external flash via the JTAG interface. This jumper setting will also allow the DA14585/586 to boot from the external flash.

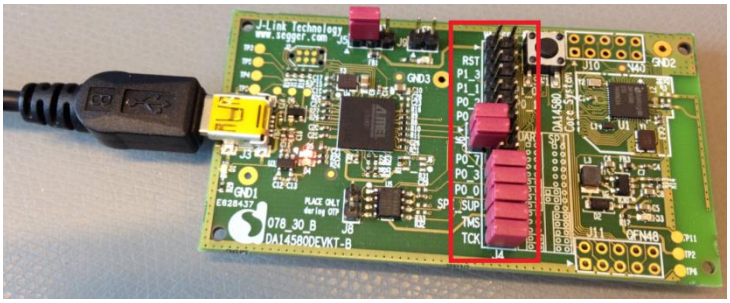


Figure 68: BASIC Development kit HW configuration

- For more information about the basic reference design, please refer to [10].

9.4 Running SUOTA with scheme 1

The system configuration of scheme 1 is described below:

- SPI/EEPROM flash only (no OTP is used).
- The dual image bootloader is stored at address `0x0` Image #1 is stored at address `0x8000`.
- Image #2 is stored at address `0x13000`.

- The product header is stored at address 0x1F000.
- Production settings are stored after the product header.

The memory architecture of scheme 1 is shown below:

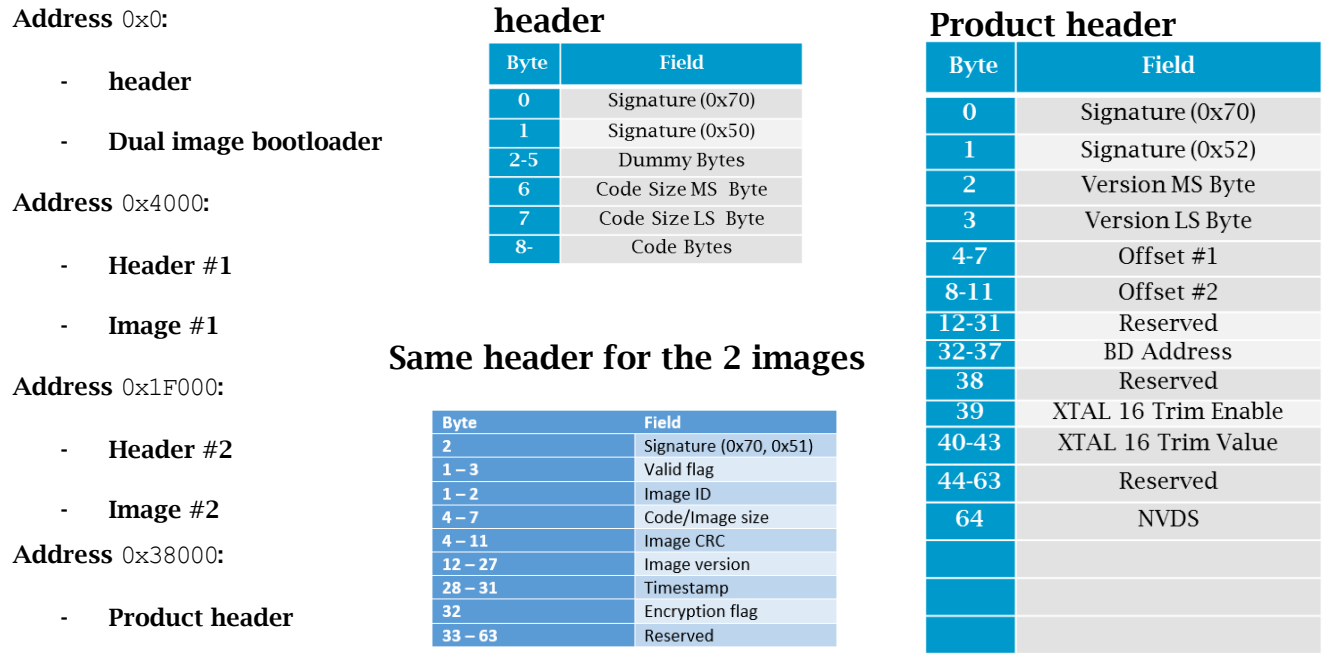


Figure 69: Memory architecture of scheme 1

Booting sequence:

- Boot according from SPI Flash or EEPROM
- The dual image bootloader will:
  - Retrieve the image addresses by reading the product header.
  - Find the last updated (active) image.
  - Load the active image to SRAM and execute the application
- SUOTA for firmware update:
  - Update specific image bank or update an older image

9.5 Running SUOTA with scheme 2

The system configuration of scheme 2 is shown below:

- SPI/EEPROM flash & OTP are used.
- The dual image bootloader is burnt into the OTP.
- Image #1 is stored at address 0x8000.
- Image #2 is stored at address 0x13000.
- The product header is stored at address 0x1F000.
- Production settings are stored after the product header or in OTP.

The memory architecture of Scheme 1 is shown below:

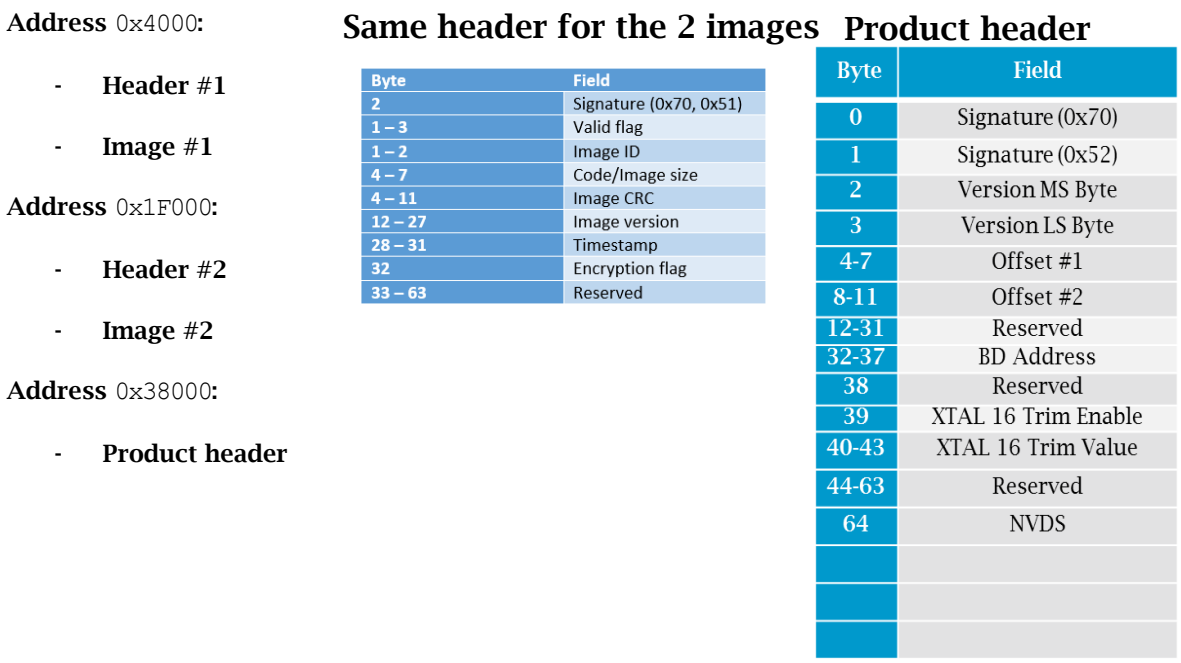


Figure 70: Memory architecture of Scheme 2

Booting sequence:

- System boots in normal mode with a faster booting time than Scheme 1.
- The dual image bootloader will:
  - Retrieve the image addresses by reading the product header.
  - Find the last updated (active) image.
  - Load the active image to SRAM and execute the application
- SUOTA for firmware update:
  - Update specific image bank or update an older image

9.6 Use of PYTHON tool to create the binary files, images & programming the flash

9.6.1 Step by step process

To create the binary files, the following steps must be carried out.

1. Download Python 3.5 from www.python.org.
2. Download DA1458x_SDK_6.x.x.zip from the Dialog Semiconductor support website.
3. Download DA1458x_SUOTA_Multipart_Binary_Generator.zip from the Dialog Semiconductor support website from tutorial section.

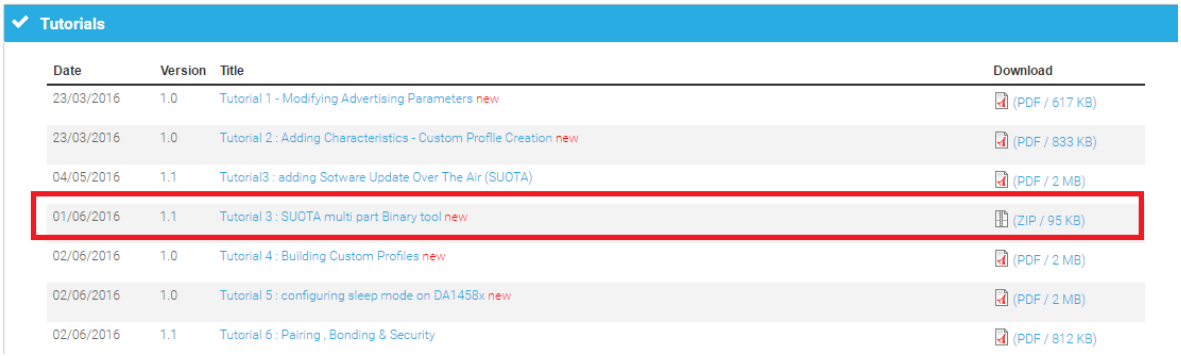


Figure 71: DA1458x_SUOTA_Multipart_Binary_Generator tools in support website

4. Install python and unzip DA14585/586_SDK_6.x.x.zip and DA1458x_SUOTA_Multipart_Binary_Generator.zip in a suitable location.
5. Figure 71: DA1458x_SUOTA_Multipart_Binary_Generator tools in support website
6. Open in KEIL IDE 5 the `..\projects\target_apps\ble_examples\prox_reporter\Keil_5\prox_reporter.uvprojx` from DA14585/586_SDK_6.x.x.
7. Change the default BD_ADDRESS.

   @File da1458x_config_advanced.h

   Example:

   #define CFG_NVDS_TAG_BD_ADDRESS {0x19, 0x00, 0x00, 0x00, 0x00, 0x19}
8. Define DLG_SUOTAR module in your application code.

   @File user_modules_config.h */

   Example:

   #define EXCLUDE_DLG_SUOTAR (0) /* included */
9. Define suotar.h

   @File user_profiles_config.h

   Example:

   #define CFG_PRF_SUOTAR
10. Turn off sleep mode

    @File user_config.h

    Example:

    const static sleep_state_t app_default_sleep_mode = ARCH_SLEEP_OFF;
11. Change the device advertising name.

    @File user_config.h

    Example:

    #define USER_DEVICE_NAME ("SUOTA-1")
12. Change the software version

    @File sdk_version.h

    Example:

    #define SDK_VERSION "v_6.0.1.207"

 # define SDK_VERSION_DATE "2017-03-13 12:06 "

 # define SDK_VERSION_STATUS "REPOSITORY VERSION v_6.0.1.207"

13. Build the project and *rename* `..\out_585\prox_reporter_585.hex` to *fw_1.hex*
14. Rename *sdk_version.h* to *fw_1_version.h*
15. Create a folder with the name '*temp*'
16. Copy *fw_1_version.h* and *fw_1.hex* to the folder '*temp*'.
17. Change the device advertising name.

    @File user_config.h

    Example:

    #define USER_DEVICE_NAME ("SUOTA-2")
18. Change the software version

    @File sdk_version.h

    Example:

    #define DA14580_SW_VERSION "v_5.0.4.1"

    #define DA14580_SW_VERSION_DATE "2016-06-14 16:01 "

    #define DA14580_SW_VERSION_STATUS "REPOSITORY VERSION"
19. Build the project and rename `..\out_585\prox_reporter_585.hex` to *fw_2.hex*

    Rename *sdk_version.h* to *fw_2_version.h*.
20. Copy *fw_2_version.h* and *fw_2.hex* to the folder '*temp*'.
21. Build the project `utilities\secondary_bootloader\secondary_bootloader.uvprojx` and copy `\Out\secondary_bootloader.hex` to the folder named input.
22. Copy and paste the *temp* folder contents inside `DA1458x_SUOTA_Multipart_Binary_Generator` *input* folder.

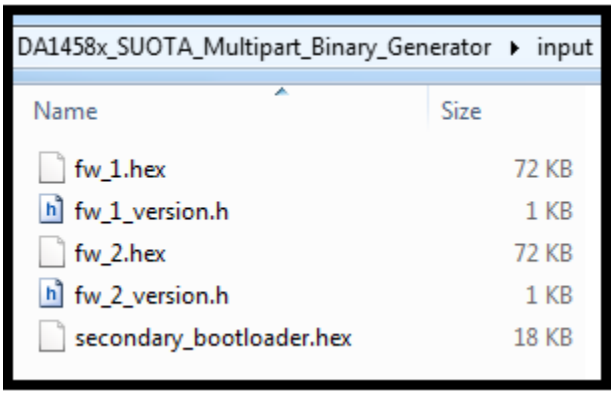


Figure 72: Content of input folder

23. Run command prompt and change the directory to *DA1458x_SUOTA_Multipart_Binary_Generator* folder.
24. Execute *project_multipart_binary_v2.py* from command prompt.

```
\DA1458x_SUOTA_Multipart_Binary_Generator>python project_multipart_binary_v2.py
############## FREE SCRIPT FOR CREATING DA1458x MULTI PART IMAGE   ###################
###################### M ALAM and contributors ######################################
#################### version     :: sw_v_0000.00002 ###################################
###################### Developed in 2016 March ######################################
hex2bin v1.0.10, Copyright (C) 2012 Jacques Pelletier & contributors

Lowest address   = 00000000
Highest address  = 0000665B
Pad Byte         = FF
8-bit Checksum = 91
hex2bin v1.0.10, Copyright (C) 2012 Jacques Pelletier & contributors

Lowest address   = 00000000
Highest address  = 0000665B
Pad Byte         = FF
8-bit Checksum = 93
hex2bin v1.0.10, Copyright (C) 2012 Jacques Pelletier & contributors

Lowest address   = 00000000
Highest address  = 0000184B
Pad Byte         = FF
8-bit Checksum = 43


INFO:: fw_image_1.img  is successfully created with no data encryption.
INFO:: fw_image_2.img  is successfully created with no data encryption.


Creating image 'output\fw_multi_part_spi.bin'...
[00000000] AN-B-001 SPI header
[00000008] Bootloader
[00001854] Padding (FF's)
[00008000] 'output\fw_image_1.img'
[0000e69c] Padding (FF's)
[00013000] 'output\fw_image_2.img'
[0001969c] Padding (FF's)
[0001f000] Product header
fw_multi_part_spi.bin  is created successfully.
##################################################################################################
```

Figure 73: Python script in action

24. Check the *output folder* and you will find *fw_multi_part_spi.bin* is created.

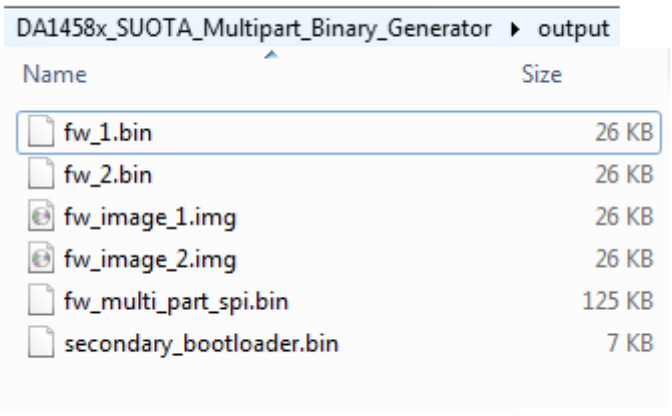


Figure 74: Generated output folder

Note: Proceed to 9.6.5 to download SUOTA in your peripheral device.

9.6.2 Details about the python script

Inside the script you will find a section "USER DATA CONFIGURATION SECTION". It ends with "USER DATA CONFIGURATION SECTION ENDS". A user is only concerned to change input values in this section. Below you will find all the details necessary to modify variables and work with the SUOTA:

Variable name	FW_1
Default value	"fw_1"
Description	Set the name of firmware 1 hex file and represents the fw_1.hex file from the input folder.

Variable name	FW_1_SW_VER
Default value	"fw_1_version"
Description	Set the name of firmware 1 header file with file extension and represents the fw_1_version.h file from the input folder.

Variable name	FW_2
Default value	"fw_2"
Description	Set the name of firmware 2 hex file without file extension and represents the fw_2.hex file from the input folder.

Variable name	FW_2_SW_VER
Default value	"fw_2_version"
Description	Set the name of firmware 2 header file without file extension and represents the fw_2_version.h file from the input folder.

Variable name	BOOT_2ND_LOADER
Default value	"secondary_bootloader"
Description	Set the name of secondary bootloader file without file extension and represents the secondary_bootloader.hex file from the input folder.

Variable name	BOOT_2ND_LOADER_IN_OTP
Default value	False
Description	If set to false then multi partition binary file is not burnt in OTP; therefore secondary bootloader will be stored in external memory. Check section 8.3 for detailed memory map.

Variable name	BOOT_2ND_LOADER_IN_OTP
Default value	False
Description	If set to false then multi partition binary file is not burnt in OTP; therefore secondary bootloader will be stored in external memory. Check section 8.3 for detailed memory map. If set to true then multi partition binary file is to be burnt in OTP; therefore secondary bootloader will NOT be stored in external memory. Check section 8.4 for detailed memory map.

Variable name	EXTERNAL_MEMORY
Default value	"spi"
Description	Set 'spi' (lowercase only) to create multi partition binary file to program in spi external memory. Set 'eeprom' (lowercase only) to create multi partition binary file to program in eeprom external memory. Check section 8.3 and 8.4 for detailed memory map and tricks to make eeprom to work.

Variable name	MEM_LOC_FOR_FW_1_IMG
Default value	"0x8000"
Description	Set memory location off-sets for firmware 1 image

Variable name	MEM_LOC_FOR_FW_2_IMG
Default value	"0x13000"
Description	Set memory location off-sets for firmware 2 image

Variable name	MEM_LOC_FOR_PRODUCT_HEADER
Default value	"0x1F000"

Description	Set memory location off-sets for product header

Variable name	IMG_1_ENC
Default value	False
Description	Set True or False; if True then output image of firmware 1 will be created with default encryption key and init vector values. IMG_ENC_KEY_DEF = "06A9214036B8A15B512E03D534120006" IMG_ENC_INIT_VEC_DEF = "3DAFBA429D9EB430B422DA802C9FAC41"

Variable name	IMG_2_ENC
Default value	False
Description	Set True or False; if True then output image of firmware 2 will be created with default encryption key and init vector values. IMG_ENC_KEY_DEF = "06A9214036B8A15B512E03D534120006" IMG_ENC_INIT_VEC_DEF = "3DAFBA429D9EB430B422DA802C9FAC41"

Variable name	IMG_1_ENC_MANUAL
Default value	False
Description	Set True or False; if True then output image of firmware 1 file will be created with user defined encryption key and init vector values;

Variable name	IMG_2_ENC_MANUAL
Default value	False
Description	Set True or False; if True then output image of firmware 2 file will be created with user defined encryption key and init vector values;

Variable name	IMG_ENC_KEY_1
Default value	""
Description	If IMG_1_ENC is set True; AND IMG_1_ENC_MANUAL is set True then a set 32 byte user defined number Format: IMG_ENC_KEY_1 = "AAAABBBBCCCCDDDDEEEEFFFF00001111"

Variable name	IMG_ENC_INIT_VEC_1
Default value	""
Description	If IMG_1_ENC is set True; AND IMG_1_ENC_MANUAL is set True then a set 32 byte user defined number Format: IMG_ENC_INIT_VEC_1 = "AAAABBBBCCCCDDDDEEEEFFFF11112222"

Variable name	IMG_ENC_KEY_2
Default value	""
Description	If IMG_2_ENC is set True; AND IMG_2_ENC_MANUAL is set True then a set 32 byte user defined number Format: IMG_ENC_KEY_2 = "AAAABBBBCCCCDDDDEEEEFFFF00001111"

Variable name	IMG_ENC_INIT_VEC_2
Default value	""
Description	If IMG_2_ENC is set True; AND IMG_2_ENC_MANUAL is set True then a set 32 byte user defined number Format: IMG_ENC_INIT_VEC_2 = "AAAABBBBCCCCDDDDEEEEFFFF11112222"

9.6.3 Creation of the fw_multi_part_spi.bin for the SPI memory using Scheme 1

Scheme 1 has the OTP blank. Therefore, the secondary bootloader will have to be stored in the external memory.

Using Scheme 1, the file fw_`multi_part_spi.bin` which will be programmed into the SPI memory has the outline architecture shown below:

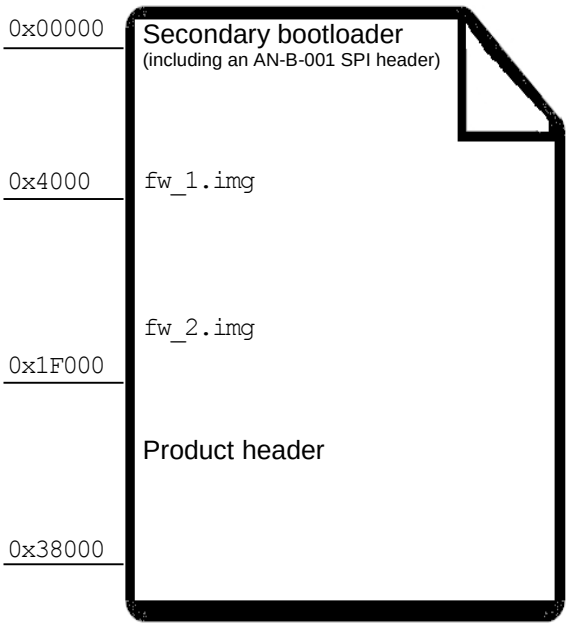


Figure 75: `multi_part.bin` file

In the python script simply set `BOOT_2ND_LOADER_IN_OTP` to False (False is also the default value) to make this scheme 1 to be activated. `EXTERNAL_MEMORY` = 'spi' (spi is also the default value) will use spi flash external memory to store the secondary bootloader, fw_1, fw_2 and product header.

IMPORTANT NOTE
In case, the EEPROM is used as an external memory, then assign EXTERNAL_MEMORY = 'eeprom' and BOOT_2ND_LOADER_IN_OTP = False then secondary bootloader to be stored in eeprom. fw_`multi_part_eeprom.bin` will be created instead of fw_`multi_part_spi.bin in output folder`

9.6.4 Creation of the multi_part.bin for the SPI memory using Scheme 2

Scheme 2 has the secondary bootloader burnt in the OTP. Therefore, the secondary bootloader will not be stored in the external memory.

Using Scheme 2, the file fw_`multi_part_spi.bin` which will be programmed into the SPI memory has the outline architecture shown below:

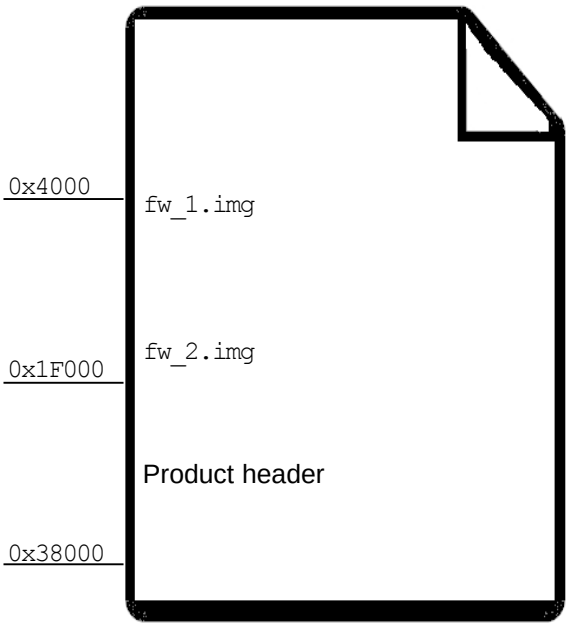


Figure 76: multi_part.bin file

In the python script simply set `BOOT_2ND_LOADER_IN_OTP` to True (False is also the default value) to make this scheme 2 to be activated. `EXTERNAL_MEMORY` = 'spi' (spi is also the default value) will use spi flash external memory to store the `fw_1`, `fw_2` and product header.

IMPORTANT NOTE
In case, the EEPROM is used as an external memory, then assign EXTERNAL_MEMORY = 'eeprom' and BOOT_2ND_LOADER_IN_OTP = True then secondary bootloader to be stored in OTP. fw_multi_part_eeprom.bin will be created instead of fw_multi_part_spi.bin in output folder

9.6.5 Preparing the SPI memory: erasing the SPI memory

First of all, make sure you have selected the JTAG connection from the SmartSnippet window as shown below:

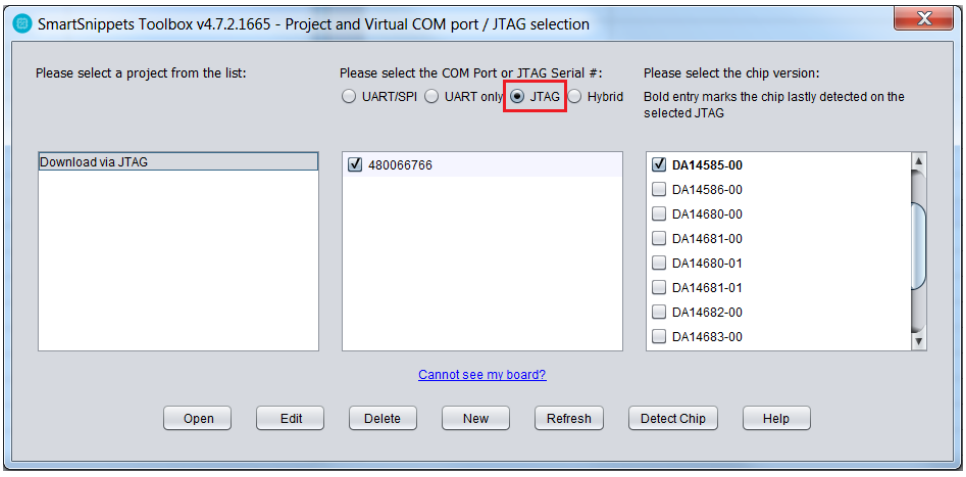


Figure 77: Select JTAG connection

If the external memory is an SPI flash device, proceed as follows:

1. Click on the FLASH tab on the left side of the SmartSnippets windows
2. Select the `multi_part.bin` file to be downloaded into the external memory
3. Press the 'Connect" button
4. Press the 'ERASE; button

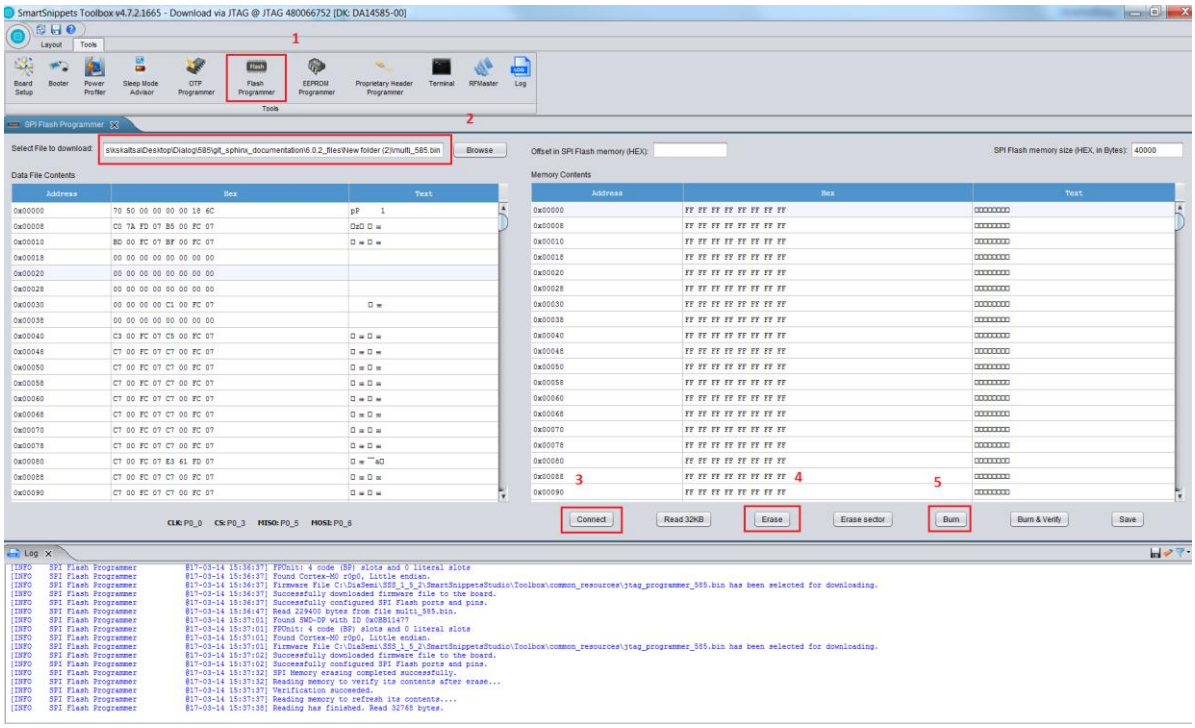


Figure 78: ERASE operation

5. Press the 'Burn' button

After pressing the 'burn' button, the windows shown in Figure 79 (below) will appear:

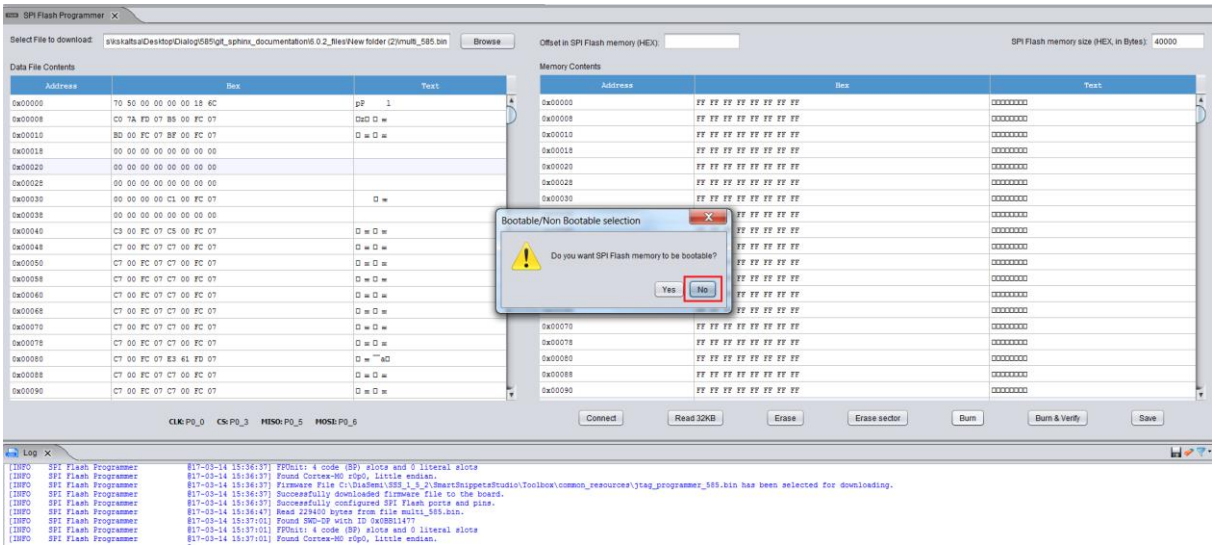


Figure 79: NON-bootable mode

- Now press NO in the bootable/Non-bootable pop-up window.

You must now **reset the DevKit** and verify that it has started advertising. The name **SUOTA-1** should be displayed from the SUOTA application (This application is available from both the App store (iOS version) & the Google Play store (Android version)).

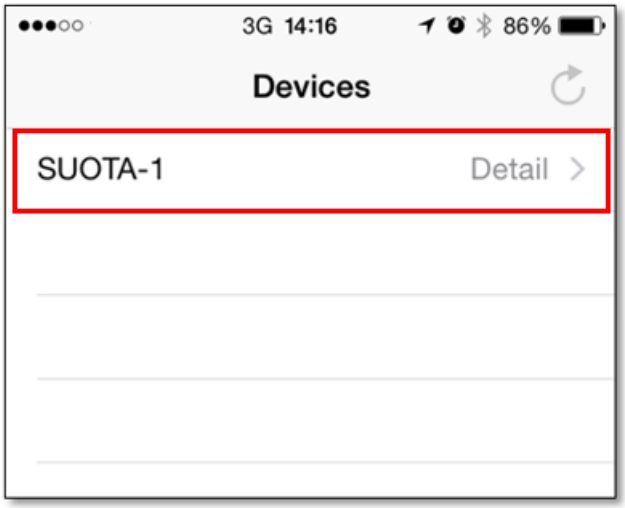


Figure 80: Detection of the DA14585/586 advertisements

9.7 Running SUOTA from an iOS platform

IMPORTANT NOTE
If the iOS SUOTA app cannot connect to an advertising DA14585/586 device make sure any old DA14585/586 devices that the iOS device has paired in the past are "forgotten" (settings->Bluetooth->click on the "**i**" next to the device name and select "Forget This Device").

First, make sure the Dialog SUOTA app is installed (available from the App Store).

Start iTunes, connect the iOS device to the PC via USB and:

1. Go to the 'Apps' section
2. Scroll down to 'File Sharing' (see 2a, below) and click on SUOTA app (2b)
3. Drag and drop the image files to the 'SUOTA documents' section from the mkimage folder.

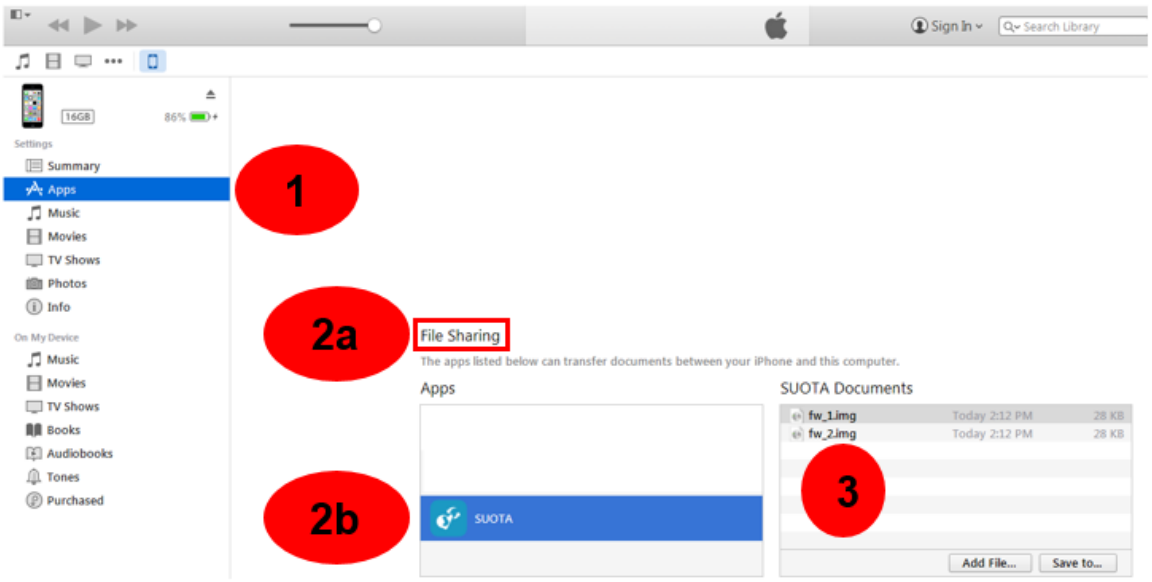
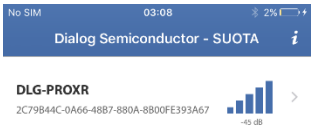
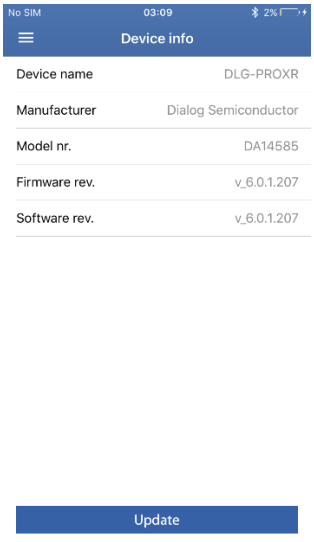
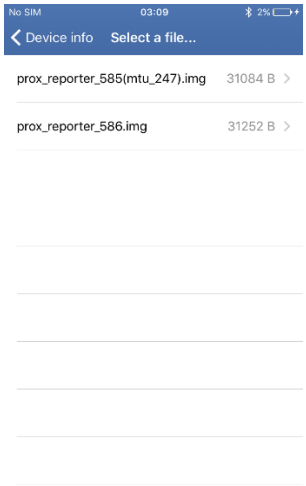


Figure 81: Copying images into the SUOTA app

Then:

1. Start the SUOTA application on the iOS device
2. The DA14585/586 should advertise at this point and the device name should be detected by the application. If not, click on the clockwise arrow to initiate scanning.
3. Click on the SUOTA-1 device to connect and see the DIS info screen. Verify that the "Firmware rev." field has the same value as the DA1458X_SW_VERSION string set during image creation.
4. After clicking on the "Update" button, the file selection screen appears. Select prox_reporter_585(mtu_247).img or prox_reporter_586.img to update.

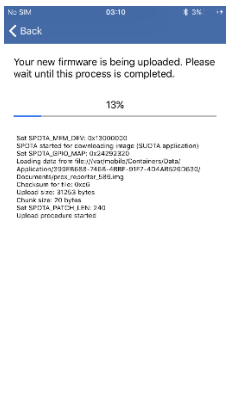
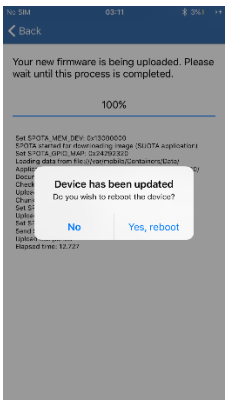
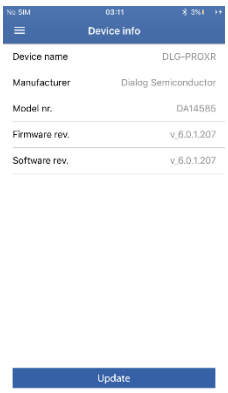
Table 26: First 4 steps to use SUOTA with iOS

Step 1	Step 2	Step 3	Step 4
			

5. After the file selection, the memory parameters configuration screen is shown. In this screen, the default GPIO settings for SPI FLASH configuration are pre-set. Also, the "Image Bank" is set by default to "Oldest" and the "Block size" to "240".

6. As soon as the "Send to device" button is pressed, the log screen appears with a status bar.
7. When the image is uploaded successfully, reboot the device in order to start advertising as SUOTA-2
8. The DA14585/586 should advertise at this point and the SUOTA-2 device should be detected by the application. Click on the device to connect and verify the "Firmware rev." value.

Table 27: Next 4 steps to use SUOTA with iOS

Step 5 (DA14585)	Step 5 (DA14586)	Step 6	Step 7	Step 8
				

IMPORTANT NOTE: AVOID THE SAME IMAGE ERROR

When the user tries to update an image that has the same software version **and** the same timestamp as the new image, a "Same Image Error" message is displayed on the iOS screen.

To avoid this error during a demo do one of the following:

- If two images are used then always update both memory banks with the same image. For example, in this demo description, the prox_reporter_585(mtu_247).img was used for both image banks when creating the multi_part.bin (step 11). When the SUOTA app was used to upload prox_reporter_586.img, only one of the memory banks has been updated. The other one still holds prox_reporter_585(mtu_247).img. To make sure that the remaining prox_reporter_585(mtu_247).img is updated with prox_reporter_586.img, upload prox_reporter_586.img again. If you want to switch back to prox_reporter_585(mtu_247).img, then upload prox_reporter_585(mtu_247).img twice to replace both image banks. By uploading the same image twice (replacing the old images in both memory banks), the "Same Image error" is eliminated.
- Create and use three images and sequentially upload one after the other. By doing this it is guaranteed that "Same Image Error" will not happen.
- Note that in normal use the "Same Image Error" rarely happens. Customer will normally create a new image to update an old one. However, in the case of a demo, the same files are used to switch from one image to another and back, so it is possible that a "Same Image Error" might occur if the two memory banks implementation is not well understood.

9.8 Running SUOTA from an Android platform

IMPORTANT NOTE: Make sure that the SmartTag device is not paired with another device

When the SmartTag device is in Advertising mode, the user can 'forget' a bonded central device by keeping the button pressed for 3 seconds, until a tone is heard. This indicates that the security information has been deleted from the SPI FLASH memory and a new central device can then pair with the SmartTag device.

When deleting the bonding data from the SmartTag SPI FLASH, when it is paired to an Android device, the SmartTag device needs to be removed from the list of paired devices of the Android device (this is usually done via menu *Settings -> Bluetooth -> Forget Device*).

1. Install and start the application on the Android device:

   After successful installation, the following icon should appear under the installed applications menu. Click on the icon to start the application.

2. Initial menu, scan for advertising devices:

   Press the  icon to initiate scanning. Assuming the SmartTag device is advertising, the device name and the Bluetooth Device address of the device will be displayed as shown in Figure 77.

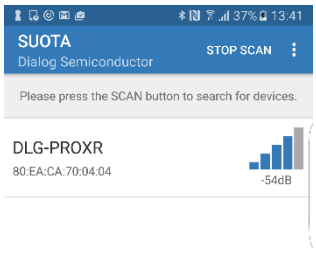
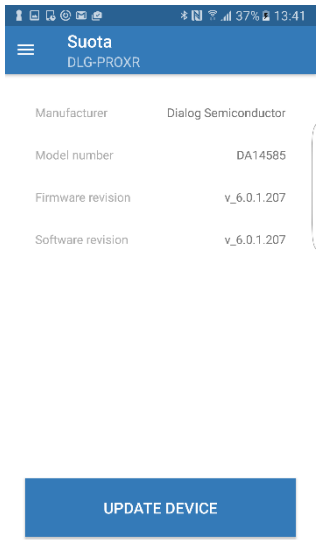
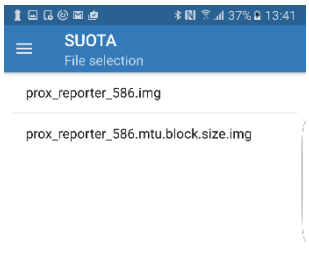
3. Connect to the SmartTag device:

   Click on the SmartTag device to connect. Upon successful connection to the SmartTag device, the DIS information will be displayed on the screen as shown in Table 28.

4. Update SmartTag software image:

   Click on the 'Update device' button and a list of files will appear on the screen. In order for the file to appear in this 'File Selection' screen it has to be copied to the 'Suota' directory of the Android device. **Connect the Android device via USB to the PC where the SmartTag images were created and copy these images under the 'Suota' directory**. An example of the 'File Selection' screen is shown in the table below. Verify that the target image is listed on this screen.

Table 28: First four steps when using SUOTA with Android

Step 1	Step 2	Step 3	Step 4
			

5. Set the SUOTA parameters:

   As soon as the image is selected, the 'Parameter settings' screen appears. See Table 29 (below) at Step 5.

First, set the memory type. The image update procedure is only supported for non-volatile memory types of SPI (FLASH memory) and I2C (EEPROM). In this example SPI (FLASH) has been selected. Then select the Image (memory) bank:

1: Use the first bank with start address as indicated in the Product Header

2: Use the second bank with start address as indicated in the Product Header

0: Burn the image into the bank that holds the oldest image

Next, define the GPIO pins of the memory device.

If the device is the DA14585 the SPI FLASH GPIO configuration is as follows:

```
MISO => P0_5
MOSI => P0_6
CS => P0_3
SCK => P0_0
```

If the device is the DA14586 the SPI FLASH GPIO configuration is as follows:

```
MISO => P2_4
MOSI => P2_9
CS => P2_3
SCK => P2_0
```

Finally, scroll down to choose the block size. A few points should be considered when this size is set:

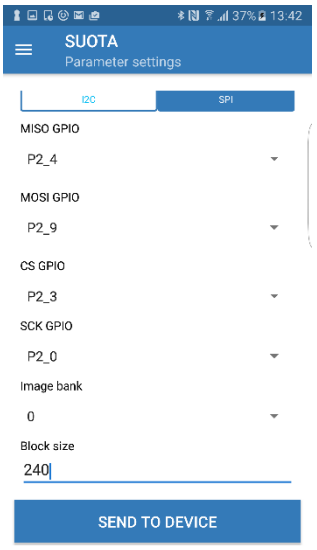
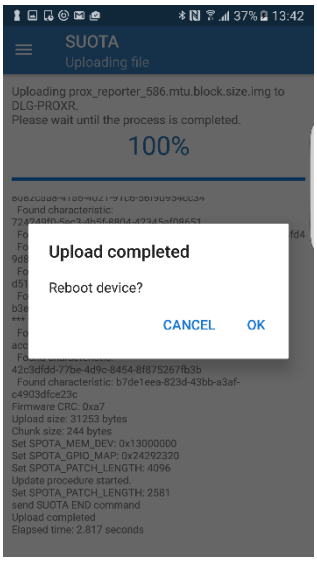
- It has to be larger than 64 bytes, which is the size of the image header.
- It must be a multiple of 20 bytes, which is the maximum amount of data that can be written at once in the `SUOTA_PATCH_DATA` characteristic.
- It should not be larger than the SRAM buffer in the SUOTA Receiver implementation, which holds the image data received over the BLE link before burning it into the non-volatile memory.

This example uses a block size of 240 bytes. After all the parameters have been set, the user can click on the 'Send to device' button at the bottom of the screen.

6. Reboot the device:

   As soon as the 'Send to device' button is clicked, a log screen appears that shows the image data blocks sent over the BLE link. In case an error occurs, a pop up indication will inform the user. When no error occurs and the SmartTag device has received and programmed the image successfully, the screen at Step 6 will appear, prompting the user to reset the SmartTag device.

Table 29: Steps 5 and 6 when using SUOTA with Android

Step 2	Step 3
	

7. Verify that the new software is running on the SmartTag device:

   Repeat steps 2 and 3 to verify that the DIS screen shows the firmware and software version of the new software.

9.9 Total time vs energy consumption to update a new image

9.9.1 A Real life example

The environment was as shown below in order to estimate how much time it takes to update the image:

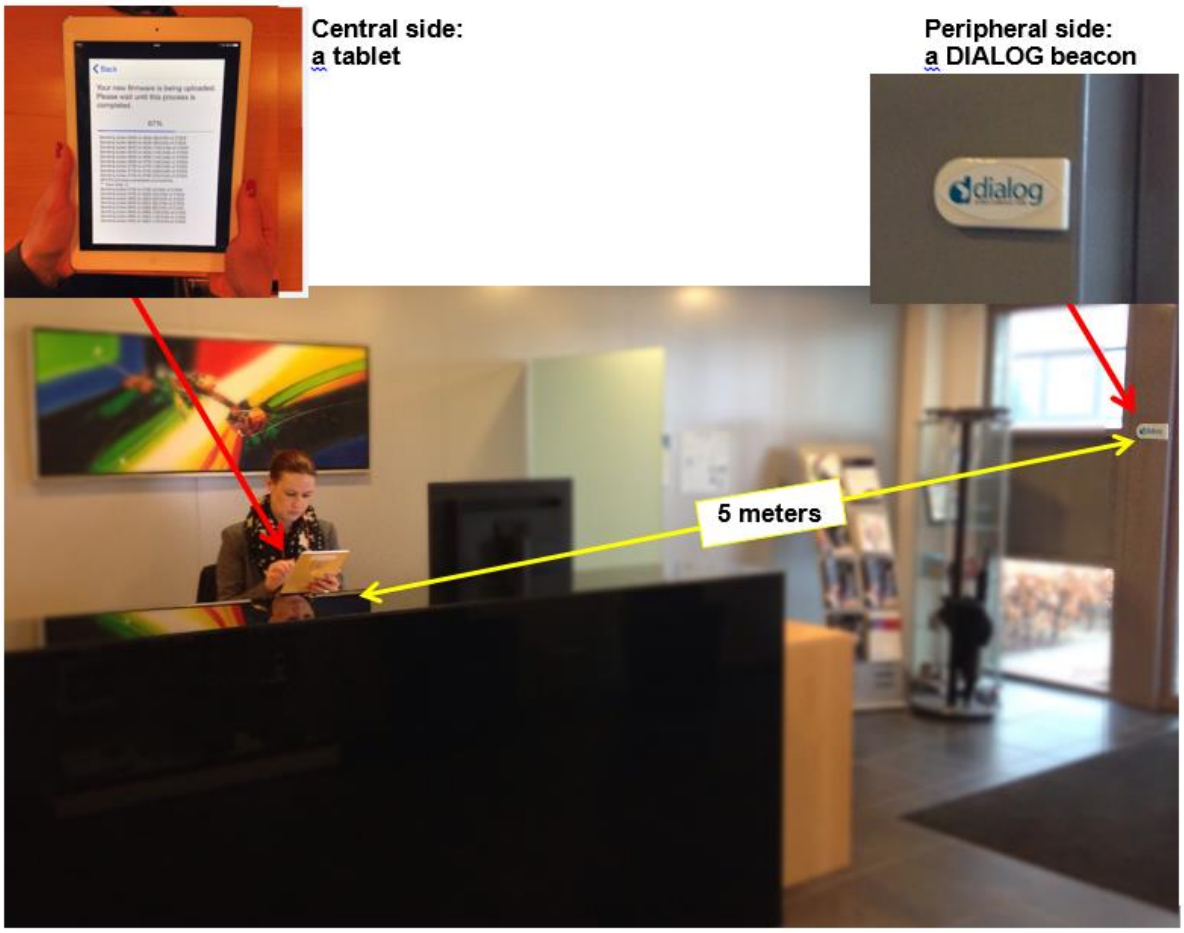


Figure 82: Real life example

- Range from the Peripheral to the Central = 5 meters.
- Average of 4 packets sent per connection interval
- Connection interval = 30 ms (set by the iOS central device)
- Image size = 27 kB
- MTU size = 20 bytes/packet, which is the default size in the software.

Table 30: Define the MTU size in the SW

Parameter	Variable/macro	Source file
MTU size	ATT_DEFAULT_MTU	attm_cfg.h

9.9.2 Total time to update a new image

The total time to update a new image using SUOTA is depending on many parameters:

- The range from the Central to the Peripheral;
- The environment;
- The average number of packets that the Central can send per connection interval;
- The connection interval set from the Central device;

- The MTU & images sizes.

The following equation gives a roughly estimate of the time needed to update the image:

$$rate\left(\frac{kbit}{s}\right) = 4\ (packets) \times 20\left(\frac{bytes}{packet}\right) \times 8\left(\frac{bits}{byte}\right) / \ connection\ interval\ (ms)$$

9.9.2.1 Result in practice

Figure 83: Time needed to update a 27 kB image

It takes 11.80 seconds to update a new image of 27 kB (including the erasing + writing operation into the flash).

9.9.2.2 Result in theory

The following formula must be applied:

$$rate\left(\frac{kbit}{s}\right) = 4\ (packets) \times 20\left(\frac{bytes}{packet}\right) \times 8\left(\frac{bits}{byte}\right)\ /\ 30\ (connection\ interval\ in\ ms)$$

$$rate\left(\frac{kbit}{s}\right) = 4 \times 20 \times 8/\ 30$$

$$rate\left(\frac{kbit}{s}\right) = 21.333\ kbit/sec$$

So, **21,333 bits** are sent **during 1 second**.

The **image size** is 27 KB = 27,648 bytes = **221,184 bits**.

Therefore, the total time needed to update a new image can be calculated as follow:

$$\Delta t = \frac{Image\ size\ (in\ bits)}{number\ of\ bit\ sent\ per\ second} = \frac{221\ 184}{21\ 333} = \ 10.4\ seconds$$

The total time needed to **update a 27 KB image** is around **10.4 seconds**. Also, the time to erase the flash plus the writing operation time should be added.

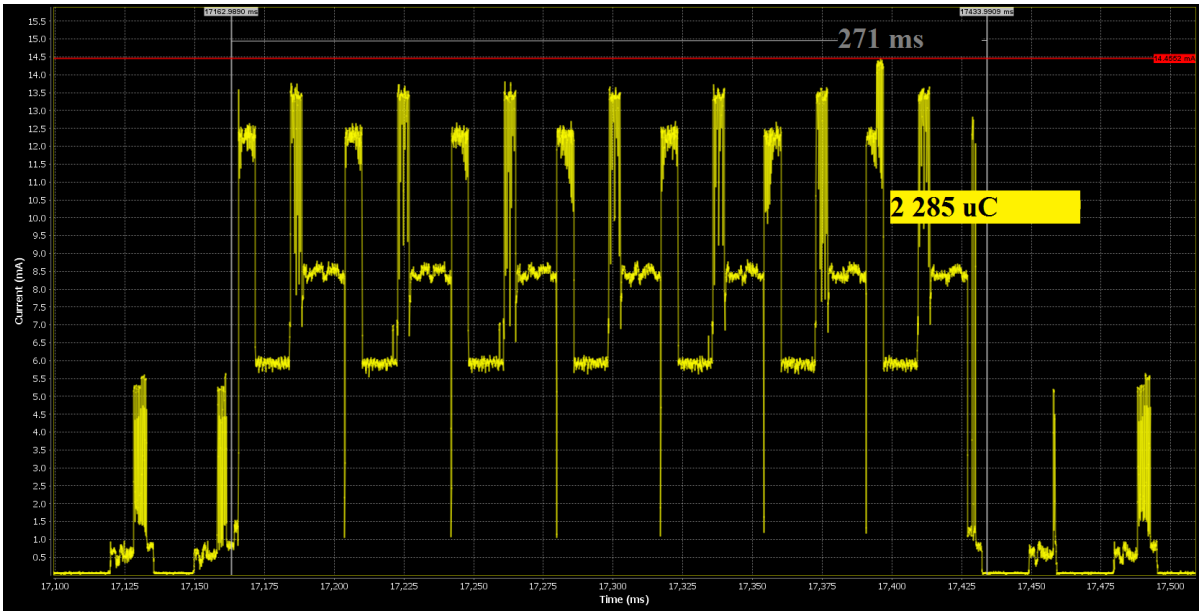
9.9.3 Energy consumption to update a new image

When SUOTA is running, the image is stored in the external flash memory. Obviously, depending on the external flash memory used, the energy consumption may vary. For more information about external flash/EEPROM, refer to [7].

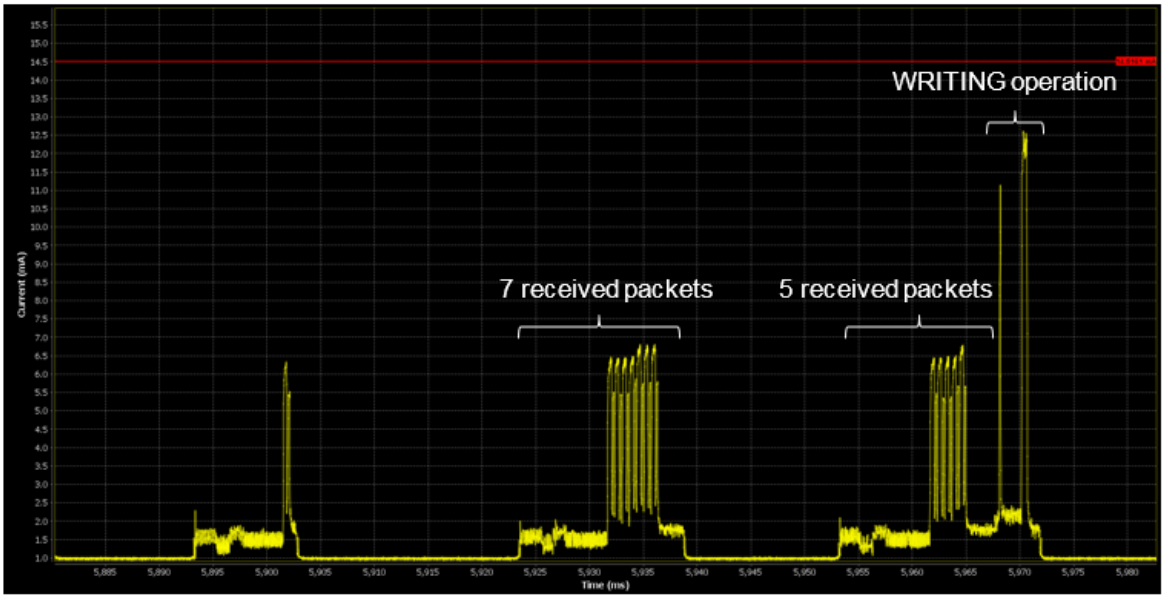
For this example, the MX25V1006E SPI flash memory from MACRONIX was used. The maximum peak current recorded reached 14.5 mA for 2.4 ms.

When SUOTA is running, the following steps occur:

1. **ERASE the Flash** (time needed: 271 ms, charge: 2 285 µC @3V).



2. **Every 12 packets (240 bytes) sent over the air, a WRITING operation is done.**
(time needed to receive 12 packets + writing operation: 92 ms, charge: 154 µC @3V).



9.10 Important notes

- SDK6.0.2 has [..*ble_examples**ble_app_ota**Keil_5**ble_app_ota.uvprojx]*. API *user_app_disconnect()*is defined to send a platform reset after SUOTA update if reset is requested by the Application layer. It is an optimal improvement made which does not exist in older SDKs.

10 Resolvable private address

10.1 Introduction

This section describes the steps required to develop BLE applications on the SmartBond™ DA14585/586 Product Family software platform, specifically for the DA14585/586 devices, as supported by the new v6.x SDK series. It guides the developer through a number of pillar examples, acquainting in the developing of BLE applications on the DA14585/586 software architecture and APIs.

This section describes the procedures that must be followed by a DA14585/586 host application in order to:

- Configure its local device address as a resolvable private address.
- Identify a bonded peer device using a resolvable private address in subsequent scanning or connection operations.

10.2 Random resolvable address mode

10.2.1 Set up device configuration

A resolvable private address is generated from an Identity Resolving Key (IRK). The IRK must be generated by the host application and provided to the BLE stack in the corresponding field of a GAPM_SET_DEV_CONFIG_CMD message.

If the device role is set to peripheral role (GAP_PERIPHERAL_SLV), the privacy bit in the flag field (GAPM_CFG_PRIVACY_EN) must be set.

10.2.2 Advertising

In order to use a resolvable random address during advertising, the host application must set *op.addr_src* to GAPM_GEN_RSLV_ADDR in the GAPM_START_ADVERTISE_CMD command. Field *op.renew_dur* controls the duration of the resolvable address before it gets regenerated and it is counted in units of 10 ms. Minimum valid value for *renew_dur* is 15000 (150 s). If the value of *renew_dur* is < 15000, the BLE stack will automatically set it to 15000.

The BLE stack will send a GAPM_DEV_BDADDR_IND event containing the generated random resolvable address to the host application upon receiving the GAPM_START_ADVERTISE_CMD command and each time a new resolvable random address is generated. The host application may store the generated random address for future usage. The random address generation process is active until the GAPM_START_ADVERTISE_CMD command is completed, i.e. until a peer connects to the device or the host application cancels advertising with a GAPM_CANCEL_CMD command.

A GAPM_START_ADVERTISE_CMD command with *op.addr_src* = GAPM_GEN_RSLV_ADDR always starts by generating a new resolvable random address. If a host application has a stored address and wishes to reuse the last generated resolvable random address in subsequent advertising commands, it can set the *op.addr_src* field to GAPM_PROVIDED_RND_ADDR and copy the stored address in *op.addr* to force the BLE stack to use this address.

10.2.3 Scan/connection

Similar to the advertising procedure, the host application in central role devices can set *op.addr_src* to GAPM_GEN_RSLV_ADDR in commands GAPM_START_SCAN_CMD and GAPM_START_CONNECTION_CMD to enable random resolvable private addressing. The address is generated by the BLE stack and returned to the host application in a GAPM_DEV_BDADDR_IND message.

The application can set *op.addr_src* to GAPM_PROVIDED_RND_ADDR in these two commands to reuse a previously generated random address.

10.3 Bonding procedure

A device using a resolvable private address should be able to distribute its local IRK to peer devices during the bonding procedure. Otherwise it will not be possible for its peer devices to identify the device when it updates its resolvable private random address.

The bonding procedure must be initiated by the central device with a GAPC_BOND_CMD command.

10.3.1 IRK distribution from a central device perspective

The host application initiates the bonding procedure using a GAPC_BOND_CMD command with:

- Set GAP_KDIST_IDKEY bit of *pairing.ikey_dist* to enable distribution of its own IRK
- Set GAP_KDIST_IDKEY bit of *pairing.rkey_dist* in GAPC_BOND_CMD to request peer IRK distribution

During the bonding procedure the central host application will receive a GAPC_BOND_IND message with *info* = GAPC_IRK_EXCH containing the peer's IRK in *data.irk*.

10.3.2 IRK distribution from a peripheral device perspective

During the bonding procedure initiated by the peer central device, the host application of the peripheral device receives a GAPC_BOND_REQ_IND with *request* = GAPC_PAIRING_REQ.

The application will respond with a GAPC_BOND_CFM with *request* = GAPC_PAIRING_RSP and

- Set GAP_KDIST_IDKEY bit of *data.pairing.rkey_dist* to enable the distribution of its own IRK
- Set GAP_KDIST_IDKEY bit of *data.pairing.ikey_dist* to request peer IRK distribution

The BLE stack will handle the distribution of the peripheral's IRK to the peer central device.

Later during the bonding procedure, the peripheral host application will receive a GAPC_BOND_IND message with *info* = GAPC_IRK_EXCH containing the peer's IRK in *data.irk*.

10.4 Resolving the address

A device can resolve a resolvable address by sending a GAPM_RESOLV_ADDR_CMD command to TASK_GAPM with the address to be resolved in *addr field* and the stored IRKs of bonded devices.

If the address is resolved by any of the provided IRKs, the TASK_GAPM will respond with a GAPM_ADDR_SOLVED_IND message containing the resolved address and the IRK resolving it. Otherwise, a GAPM_CMP_EVT event or GAPM_RESOLV_ADDR operation with status value GAP_ERR_NOT_FOUND will be sent.

10.5 Example

This section describes an example of a DA14585/586 device with a central role and a public address connecting to a DA14585/586 device with peripheral role and resolvable private random address. For better understanding of resolvable private address usage, the message flow of host applications running on both devices will be described for the following procedures:

1. Configure and start advertising with resolvable private address on peripheral device.
2. Bonding procedure.
3. Resolve peripheral device random address during scanning procedure.

10.5.1 Advertising with resolvable private address

The host application of the DA14585/586 in a peripheral role must generate the IRK before device configuration. This can be done in the *app_init()* function. The following member is added in *app_sec_env* to store the IRK data. The IRK size always has the maximum value: KEY_LEN = 16 (bytes).

```
struct gap_sec_key irk;
```

A sample function of pseudo random key generation is as follows:

```
void app_sec_gen_irk(void)
{
    // Counter
    uint8_t i;
    // Randomly generate the end of the LTK
    for (i = 0; i < KEY_LEN; i++)
    {
        app_sec_env.irk.key[i] = rand()%256;
    }
}
```

In a device configuration command, the application provides the stored local IRK and IRK size. It must also enable the peripheral privacy flag. Below follows a sample code with the required actions for this operation.

```
// set device configuration
struct gapm_set_dev_config_cmd* cmd = KE_MSG_ALLOC(GAPM_SET_DEV_CONFIG_CMD,
TASK_GAPM, TASK_APP, gapm_set_dev_config_cmd);

cmd->role = GAP_PERIPHERAL_SLV;
memcpy (cmd->irk.key, app_sec_env.irk.key, KEY_LEN);
cmd->flags = GAPM_CFG_PRIVACY_EN;
```

In a start advertising command, the application must set *addr_src* to GAPM_GEN_RSLV_ADDR. The *renew_dur* field must be set to the required lifetime of the random address before it gets regenerated.

```
struct gapm_start_advertise_cmd *cmd = KE_MSG_ALLOC(GAPM_START_ADVERTISE_CMD,
                                       TASK_GAPM, TASK_APP,gapm_start_advertise_cmd);

cmd->op.addr_src = GAPM_GEN_RSLV_ADDR;
cmd->op.renew_dur = <Random address lilftime>; // In units of 10ms. Min = 15000,  Max
= 65535
```

10.5.2 Bonding procedure

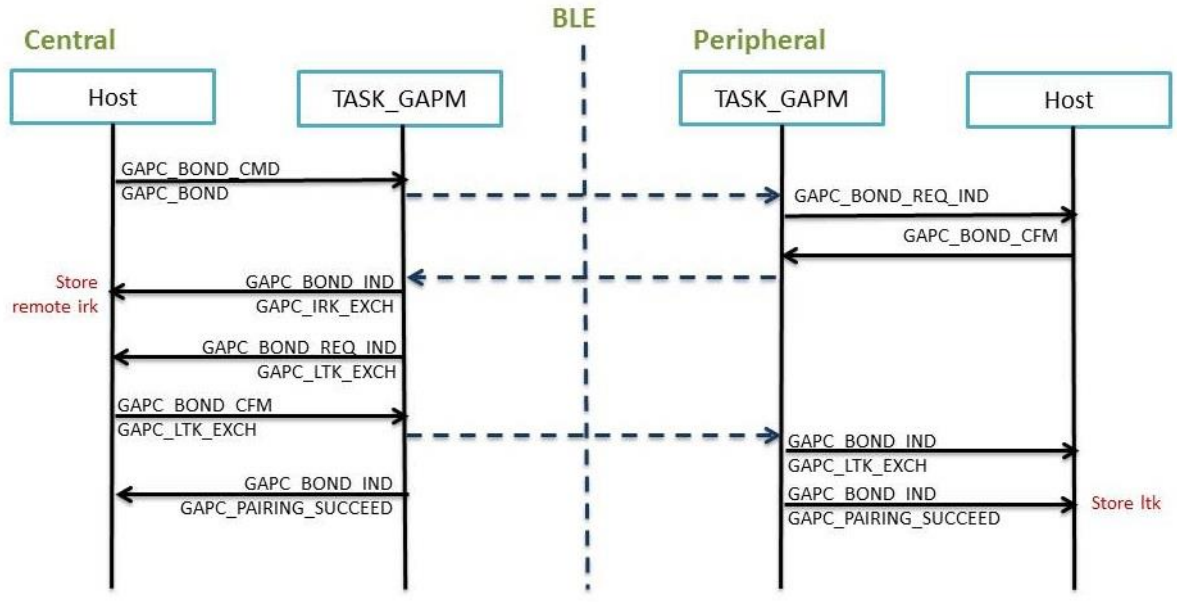


Figure 84: Message flow during bonding procedure

A central device can discover and connect to peripherals with a resolvable private address.

The central device can determine whether a discovered address is a resolvable random address by examining the combination of *adv_addr_type* in GAPM_ADV_REPORT_IND or *peer_addr_type* in GAPC_CONNECTION_REQ_IND with the value of the two most significant bits in the address. The address type of resolvable addresses is '0' for public addresses and '1' for random addresses. The two most significant bits of resolvable private addresses are equal to '01'.

For more details about resolvable private address format refer to [11], *Volume 3, 10.8.2.2 Resolvable Private Address Generation Procedure*.

After establishing a connection, the host application in a central device must initiate the bonding procedure with a peripheral to obtain the remote IRK. The bonding procedure is initiated with a GAPM_BOND_CMD command.

Other security parameters (such as OOB data presence, MITM, LTK distributor) can have various values depending on the application requirements. For this example we will assume that MITM is disabled, OOB data are not present and the LTK distributor will be the security initiator (central).

Below is a sample code enabling this setup:

```
struct gapc_bond_cmd * req = KE_MSG_ALLOC (GAPC_BOND_CMD, KE_BUILD_ID(TASK_GAPC,
app_env.conidx), TASK_APP, gapc_bond_cmd);
req->operation = GAPC_BOND;
req->pairing.sec_req = GAP_NO_SEC;
// OOB information
req->pairing.oob = GAP_OOB_AUTH_DATA_NOT_PRESENT;  // No OOB data present
// IO capabilities
req->pairing.iocap          = GAP_IO_CAP_NO_INPUT_NO_OUTPUT; // No I/O capabilities
// Authentication requirements
req->pairing.auth           = GAP_AUTH_REQ_NO_MITM_BOND; //Bonding, No MITM
// Encryption key size
req->pairing.key_size       = 16;
//Initiator key distribution
req->pairing.ikey_dist      = SMP_KDIST_ENCKEY; //Initiator distributes LTK
//Responder key distribution
req->pairing.rkey_dist      = SMP_KDIST_IDKEY; //Request receiver to distribute IRK
```

The host application in the peripheral device will receive a GAPC_BOND_REQ_IND message with *request* = GAPC_PAIRING_REQ, indicating that the central device has started the pairing (bonding) procedure. It must respond with a GAPC_BOND_CFM. Below is the sample code for the response:

```
struct gapc_bond_cfm* cfm = KE_MSG_ALLOC(GAPC_BOND_CFM, TASK_GAPC, TASK_APP,
gapc_bond_cfm);

cfm->request = GAPC_PAIRING_RSP; //pairing response
cfm->accept = true;
// OOB information
cfm->data.pairing_feat.oob           = GAP_OOB_AUTH_DATA_NOT_PRESENT; // No OOB data
present
// Encryption key size
cfm->data.pairing_feat.key_size      = KEY_LEN;
// IO capabilities
cfm->data.pairing_feat.iocap         = GAP_IO_CAP_NO_INPUT_NO_OUTPUT; // No I/O
capabilities
// Authentication requirements
cfm->data.pairing_feat.auth          = GAP_AUTH_REQ_NO_MITM_BOND; //Bonding, No MITM
//Security requirements
cfm->data.pairing_feat.sec_req       = GAP_NO_SEC;
//Initiator key distribution
cfm->data.pairing_feat.ikey_dist     = SMP_KDIST_ENCKEY; //Initiator distributes LTK
//Responder key distribution
```



```
cfm->data.pairing_feat.rkey_dist      = SMP_KDIST_IDKEY; //Request receiver to
distribute IRK
```

The host application in the central device will receive an indication that the remote IRK is distributed. The host must extract the IRK of the remote device (from the *irk* structure in the data union) and store it for use in resolving the peripheral device's random address for reconnection. Below is the sample code:

```
int gapc_bond_ind_handler(ke_msg_id_t msgid,
                          struct gapc_bond_ind *param,
                          ke_task_id_t dest_id,
                          ke_task_id_t src_id)
{
    switch (param->info)
    {
           //... other cases ...
           case GAPC_IRK_EXCH:
               // store param->data.irk in the application environment
               // in a variable of type struct gapc_irk
            break;
           //... other cases ...
    }
    return 0;
}
```

On the peripheral side, no IRK distribution is requested at the host application layer. The local IRK is stored in BLE stack layers and IRK distribution is handled in the BLE stack.

Since the central device is the distributor of LTK in this example, its host application will receive a GAPC_BOND_REQ_IND message with *request* = GAPC_LTK_EXCH and must distribute the LTK with the corresponding GAPC_BOND_CFM message.

The host application on the peripheral device will receive a GAPC_BOND_IND message with *request* = GAPC_LTK_EXCH, containing the distributed LTK. Both devices must store the LTK information for establishing secure links in subsequent connections.

The bonding procedure will be completed by the host applications on both devices by the reception of a GAPC_BOND_IND message with field *info* = GAPC_PAIRING_SUCCEED.

Message flow during bonding procedure on both devices is outlined Figure 84.

10.6 Resolving peripheral device random address during scanning

The central host application starts scanning as usual. Upon receiving an advertising report (GAPM_ADV_REPORT_IND) the application checks whether it is from a device with a resolvable random address:

```
int gapm_adv_report_ind_handler(ke_msg_id_t msgid,
                                struct gapm_adv_report_ind *param,
                                ke_task_id_t dest_id,
                                ke_task_id_t src_id)
{
       if(param->report.adv_addr_type == 1
              && ( (param->report.adv_addr.addr[5] & 0xC0) = 0x40) )
       {
              // it is a resolvable random address
       }
       return (KE_MSG_CONSUMED);
}
```

Next, resolve the address by issuing the command GAPM_RESOLVE_ADDR_CMD:

```
struct gapm_resolv_addr_cmd *cmd =
(struct gapm_resolv_addr_cmd *)KE_MSG_ALLOC_DYN(GAPM_RESOLVE_ADDR_CMD, TASK_GAPM,
```

```
                    TASK_APP, gapm_resolv_addr_cmd, N * sizeof(struct gap_sec_key) );
cmd->operation = GAPM_RESOLV_ADDR; // GAPM requested operation
cmd->nb_key = N;                   // Number of provided IRK
cmd->addr = <<addr>> ; /// Resolvable random address to solve
cmd->irk = <<array of stored IRKs>>; /// Array of IRK used for address resolution (MSB
-> LSB)

/// Resolve Address command
struct gapm_resolv_addr_cmd
{
    /// GAPM requested operation:
    ///  - GAPM_RESOLV_ADDR: Resolve device address
    uint8_t operation;
    /// Number of provided IRK (sahlle be > 0)
    uint8_t nb_key;
    /// Resolvable random address to solve
    struct bd_addr addr;
    /// Array of IRK used for address resolution (MSB -> LSB)
    struct gap_sec_key irk[__ARRAY_EMPTY];
};
```

The host application indicates that the address is resolved with a `GAPM_ADDR_SOLVED_IND` message, containing the resolved address and the IRK resolving it.

```
/// Indicate that resolvable random address has been solved
struct gapm_addr_solved_ind
{
    /// Resolvable random address solved
    struct bd_addr addr;
    /// IRK that correctly solved the random address
    struct gap_sec_key irk;
};
```

When no provided IRK resolves the address, a `GAPM_CMP_EVT` event or `GAPM_RESOLV_ADDR` operation with *status* = `GAP_ERR_NOT_FOUND` will be sent.

11 External wake-up mechanisms

This document describes the external wake-up mechanisms that can be used in external processor based applications of the DA14585/586 Bluetooth® Smart SoC. Two wake-up mechanisms can be implemented: external processor to DA14585/586 and DA14585/586 to external processor. These can be used in either UART or SPI interface. An external processor can wake up the DA14585/586 using a GPIO signal as the interrupt line. This can be used as an alternative to the existing DA14585/586 auto wake-up process. Similarly, when the DA1585/586 wakes up due to internal kernel timer expiration or due to a pre-programmed BLE event, it can toggle a GPIO to wake up the external processor.

11.1 Introduction

In systems that use the DA14585/586 Bluetooth® Smart SoC together with an external processor, one could use special wake-up mechanisms to further extend the already ultra-low power features of the DA14585/586.

Two new features will be described here:

- The DA14585/586 sleeps forever and wakes up via an external processor signal (see Section 11.6 for more information).
- The external processor is in sleep mode and wakes up via the DA14585/586.

For the external processor to DA14585/586 wake-up feature, the DA14585/586 Wakeup Timer, as described in [3], [4] and [5] can be used. This particular DA14585/586 internal hardware block can wake up the system after a pre-programmed number of GPIO events. Any of the GPIO inputs can be selected and configured to generate a wake-up interrupt. A simple toggle of the pre-configured GPIO will wake up the system.

In this particular wake-up architecture, the DA14585/586 sleeps forever (see Section 11.6 for more information) and only wakes up when the external host wants it to. This external host processor can wake up the system by toggling a DA14585/586 GPIO assigned to a DA14585/586 wake-up interrupt and send the required message.

The following DA14585/586 signals can be used as wake-up signals:

- Any GPIO
- The DA14585/586 UART CTS signal, when the UART is being used as the transport layer.
- The DA14585/586 SPI_EN (CS) when SPI is being used as the transport layer.

When the last two signals are used the following applies: a DA14585/586 GPIO cannot have multiple functions at the same time. For example, it cannot act as a UART CTS signal and as a wake up interrupt simultaneously. Therefore, the software has to configure the GPIO functionality accordingly. Before going to sleep, the software has to configure the GPIO as a wake up interrupt. When the DA14585/586 wakes up, it should reconfigure it as UART CTS.

Sections 11.2, 11.3 and 11.6 provide a more detailed analysis of the external processor to DA14585/586 wake-up procedure. Three examples are provided: using any GPIO, using the UART CTS signal and using the SPI_EN signal.

Similarly, the DA14585/586 can wake up an external processor by toggling a GPIO signal. The DA14585/586 system could wake itself up to service BLE events, respond to internal kernel timer expirations and perform other tasks. When the DA14585/586 system wakes up, the UART or the SPI interface is automatically activated to enable the communication with the external processor and possibly send a message to it. If the external processor is in sleep mode, the DA14585/586 can toggle a GPIO to wake up the external processor.

Section 11.5 provides a more detailed analysis of the DA14585/586 to external processor wake-up mechanism.

11.2 Waking up the DA14585/586 using any GPIO

Any DA14585/586 GPIO can be used as wake-up interrupt for the DA14585/586. This generic implementation implies that the user is aware of the system hardware and that the GPIO used will not conflict with any other external hardware.

When the user decides to use any GPIO in either the UART or the SPI transport interface, it should be noted that the flow control of the transport layer should definitely be used. The flow control will provide a safe mechanism for communication synchronization. Only by using the flow control the host system will be sure that the DA14585/586 has woken up and is ready to receive messages.

Figure 86 illustrates the generic GPIO wake-up process with flow control and Figure 85 shows the required connections.

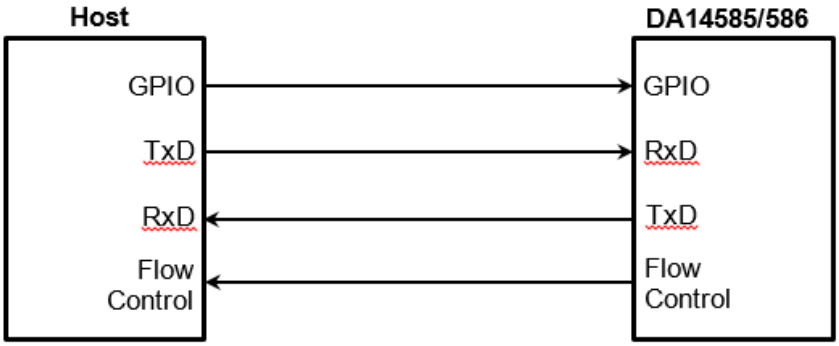


Figure 85: External CPU to DA14585/586 generic wake-up connections

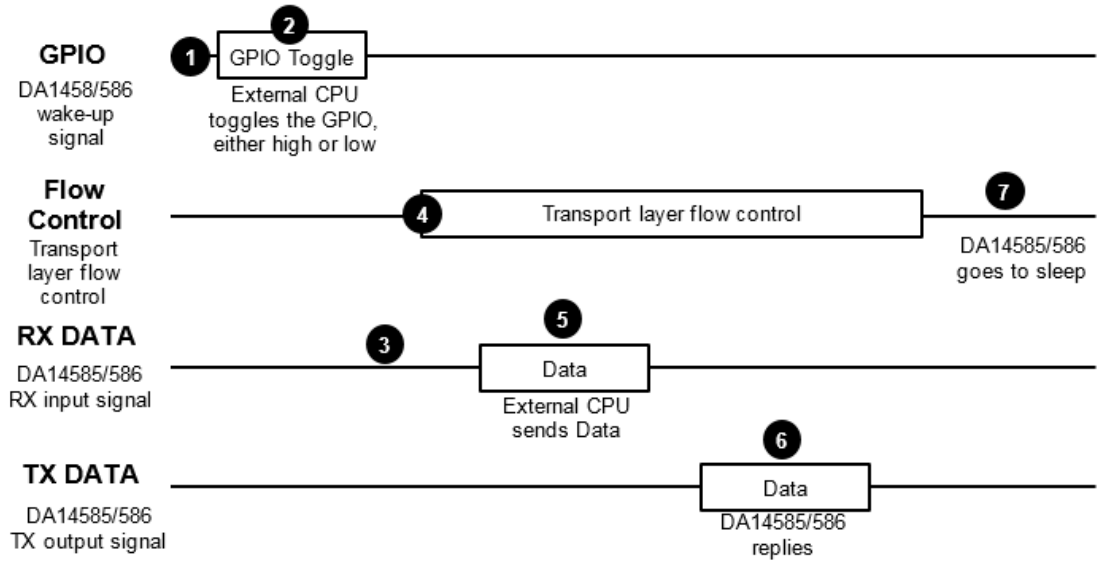


Figure 86: External CPU to DA14585/586 generic wake-up process

Details:

1. The DA14585/586 is in sleep mode. The GPIO input signal has simple GPIO functionality and is configured to have a wake-up interrupt, active either LOW or HIGH.
2. The external processor wants to send a message to the DA14585/586. It toggles the GPIO.
3. The external processor cannot send any data yet. The DA14585/586 is still in the wake-up process and therefore the external processor should wait for the flow control signal.

4. The DA14585/586 has woken up and asserts or de-asserts (depending on the transport layer used) the flow control signal. It is ready to receive messages.
5. The external processor detects the flow control signal and sends the data.
6. The DA14585/586 replies.
7. Actions on the DA14585/586 are performed, according to the received message and when finished it goes to sleep by appropriate flow control signaling.

11.3 Waking up the DA14585/586 using the UART CTS signal

In this external wake-up configuration using the UART CTS as a wake-up signal, a special requirement exists: the external processor should be able to modify its RTS signal functionality. That means that it should be able to toggle its RTS signal before sending anything via the UART interface. This may not be feasible in some processors, as the UART driver might be a fixed hardware block with dedicated signals whose functionality cannot be changed. In that case the process described in Section 11.2 can be used.

Figure 87 shows the connections needed and Figure 88 illustrate the external wake-up process. A more detailed description follows.

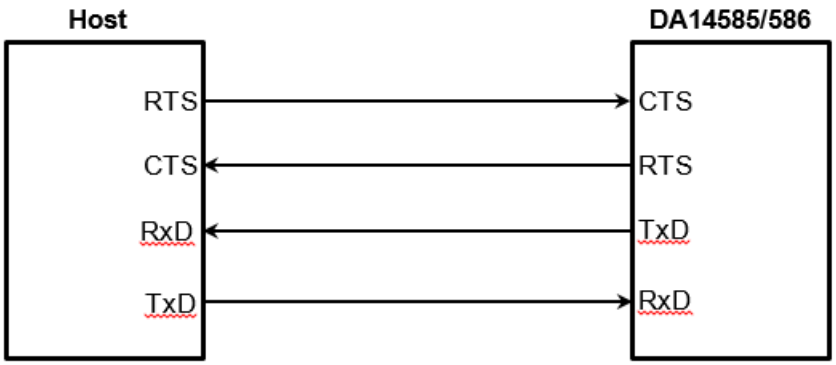


Figure 87: External CPU to DA14585/586 wake-up UART connections

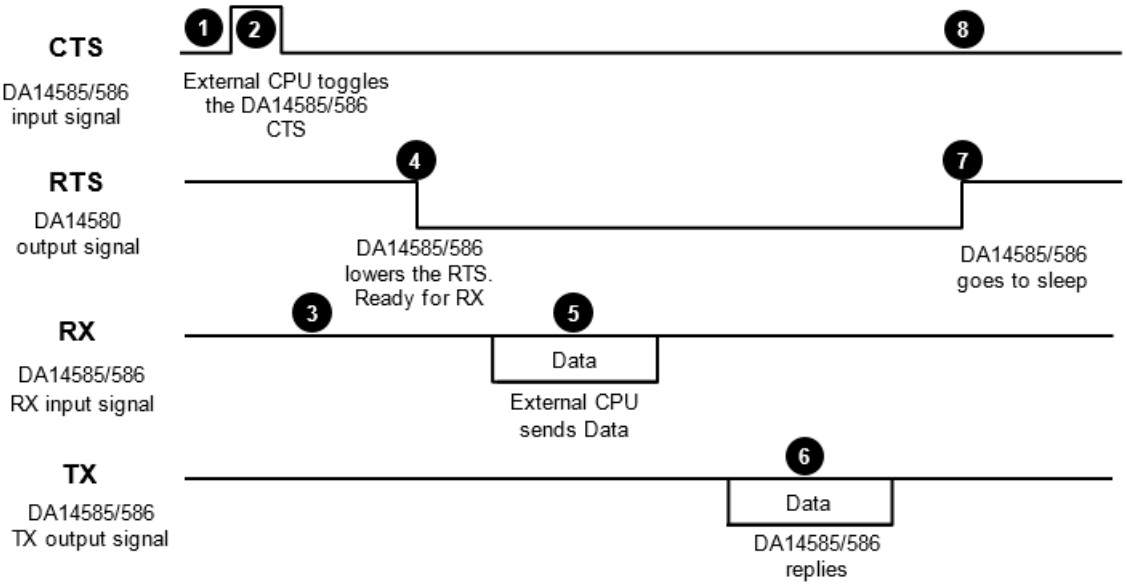


Figure 88: External CPU to DA14585/586 wake-up process via UART

Details:

1. The DA14585/586 is in sleep mode. The CTS input signal has simple GPIO functionality and is configured to have an active HIGH wake-up interrupt.
2. The external processor wants to send a message to the DA14585/586. It should take control of the CTS signal from its UART driver block and toggle it HIGH.
3. The external processor cannot send any data yet because the RTS signal is still HIGH. The DA14585/586 is still in the wake-up process.
4. The DA14585/586 has woken-up and sets the RTS signal LOW. It is ready to receive messages.
5. The external processor detects the LOW level of the RTS signal and sends the data.
6. The DA14585/586 replies.
7. Actions on the DA14585/586 are performed, according to the received message and when finished it goes to sleep, making RTS HIGH.
8. At the same time and just before entering sleep, the DA14585/586 changes the CTS signal from UART CTS functionality to simple GPIO functionality with external wake-up interrupt.

11.4 Waking up the DA14585/586 using the SPI_EN signal

The SPI external wake-up process uses a similar procedure to the UART described above. The same requirement exists, which implies that the external CPU should be able to control the SPI_EN (CS) SPI signal apart from any dedicated SPI driver it may have.

Figure 89 and Figure 90 are described in [12]. They illustrate an external wake-up process with the SPI using the SPI_EN as the wake-up signal.

Figure 89: External CPU to DA14585/586 wake-up SPI connections

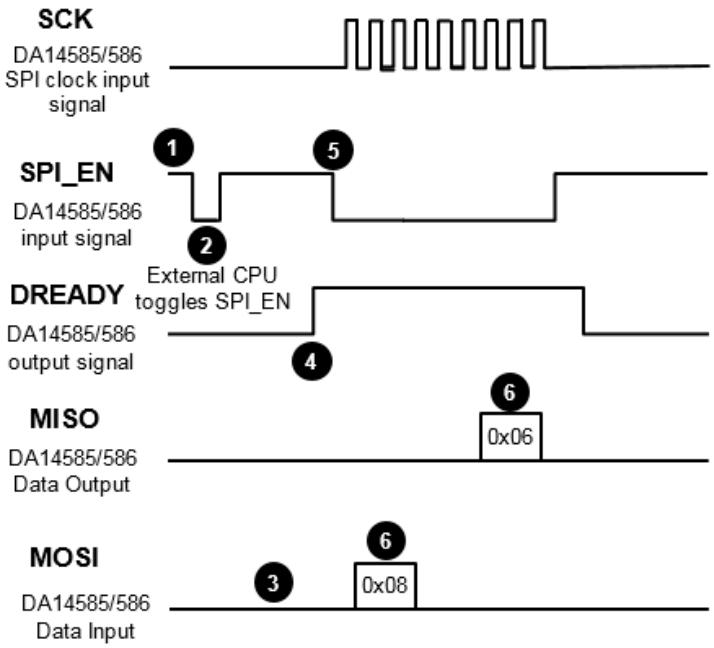


Figure 90: External CPU to DA14585/586 wake-up process via SPI

Details:

1. The DA14585/586 is in sleep mode. The SPI_EN input signal has simple GPIO functionality and is configured to have an active LOW wake-up interrupt.
2. The external processor wants to send a message to the DA14585/586. It should take control of the SPI_EN signal from its SPI driver block and toggle it LOW.
3. The external processor cannot send any data yet, since the SPI custom flow control is enabled as described in [12].
4. The DA14585/586 has woken up and sets the DREADY signal HIGH as described in [12].
5. The external processor detects the HIGH level on the DREADY signal and sets the SPI_EN LOW.
6. The MOSI and MISO bytes and the rest of the flow control process and SPI data exchange are described in [12].

11.5 Waking up an external processor

Figure 91 shows a block diagram of the connections needed for the DA14585/586 to wake up an external processor. Figure 92, illustrates the process. It shows how the DA14585/586 can wake up an external processor before sending a message over the transport layer interface. A simple DA14585/586 GPIO signal is used and is toggled according to the external processor requirements. Flow control should be used in either SPI or UART interface in order to ensure communication synchronization. The steps outlined next give a more detailed description of the process.

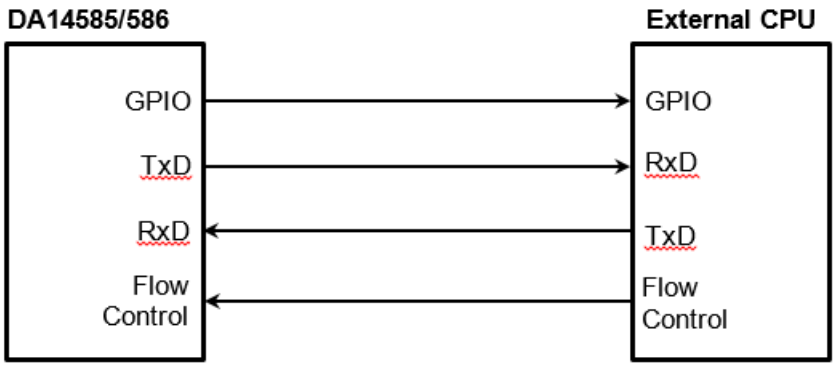


Figure 91: DA14585/586 to external processor wake-up connections

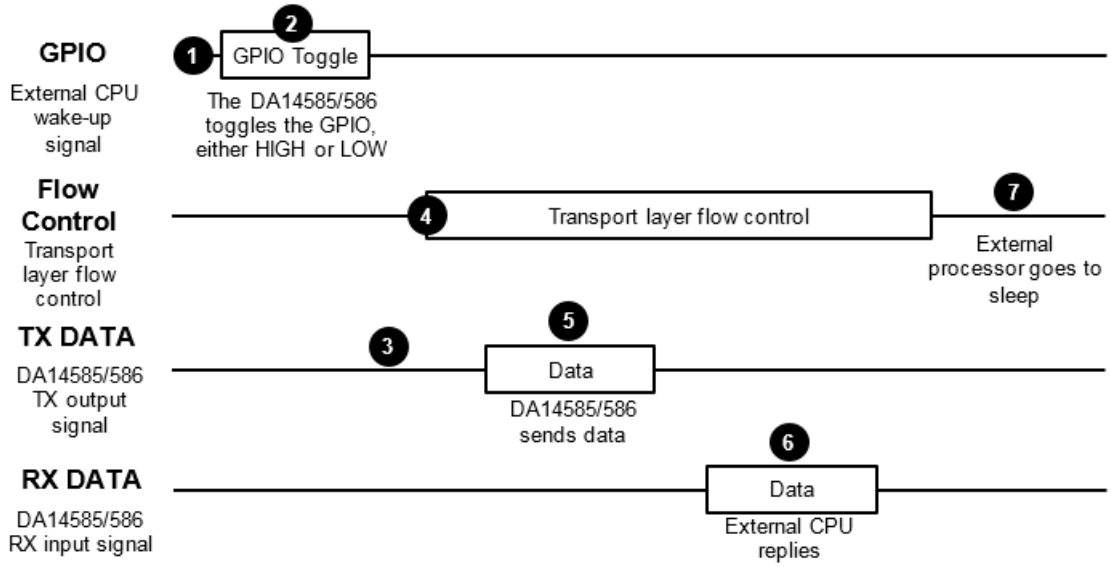


Figure 92: DA14585/586 to external processor wake-up process

Details:

1. The external processor is in sleep mode. The DA14585/586 GPIO signal, configured to act as the external processor wake-up signal, is set to output. It is initialized either to LOW or HIGH depending on the external processor wake-up signal requirements.
2. The DA14585/586 wants to send a message to the external processor. The DA14585/586 should toggle the GPIO to wake up the external processor.
3. The DA14585/586 cannot yet send any data. The external processor is still in the wake-up process and therefore the DA14585/586 should wait for the flow control signaling.
4. The external processor has woken up and it asserts or de-asserts (depending on the interface used) the flow control signal. It is ready to receive messages.
5. The DA14585/586 detects the level change of the flow control signal and sends the data.
6. The external processor replies.
7. Actions on the external processor are performed, according to the received message and when finished it goes to sleep by appropriate flow control signaling.

11.6 Sleep limitation

A sleep limitation currently exists in the software stack, which prevents the DA14585/586 to sleep forever in the external processor configuration and when the GTL interface is used. This limitation does not exist in the integrated processor configuration software.

The DA14585/586 must automatically wake itself up at least every 10 s, even when no external wake-up interrupt is triggered. This automatic wake up does not prevent the external wake-up process to be fully functional as described.

A workaround for this limitation will be given in a future release of this application note, as alternatives are still under investigation.

11.7 Software changes

The software changes needed to implement the external wake-up processes will be outlined next.

11.7.1 External processor to DA14585/586 wake-up process software

To enable the external processor to DA14585/586 wake-up process one should include the following definition:

```
/*External wake up mechanism*/
#define EXTERNAL_WAKEUP     1
```

This definition can be found in the following two files:

```
<sdk_root_directory>\sdk\platform\arch\arch.h
```

The external wake-up GPIO assignment can be passed as a function parameter when the function `ext_wakeup_enable` is called.

This is done inside the following file:

```
dk_apps\src\plf\refip\src\arch\main\ble\arch_main.c
```

```
#if ((EXTERNAL_WAKEUP) && (!BLE_APP_PRESENT)) // external wake up, only in external processor
designs
    ext_wakeup_enable(EXTERNAL_WAKEUP_GPIO_PORT, EXTERNAL_WAKEUP_GPIO_PIN,
EXTERNAL_WAKEUP_GPIO_POLARITY);
#endif
```

The above gives default GPIO wake-up configurations for either `UART CTS` or `SPI_EN` signals, depending on the communication transport layer used. One could modify the parameters of the `ext_wakeup_enable` function as desired and set any GPIO port and pin as the wake-up signal, irrespectively of the interface in use.

11.7.2 DA14585/586 to external processor wake-up process software

An example implementation exists in the throughput evaluation project. One could edit the following definitions to change the GPIO used as the wake-up signal.

```
#define EXT_WAKEUP_PORT     GPIO_PORT_0
#define EXT_WAKEUP_PIN      GPIO_PIN_7
```

These definitions can be found in the following file:

```
sdk_ref\dk_apps\src\modules\app\src\app_project\throughput_eval_peripheral_fh\system\p
eriph_setup.h
```

The activation of the wake-up process is enabled by defining the `CFG_EXT_PROCESSOR_WKUP` in the `da14580_config.h` configuration file.

11.7.3 DA14585/586 wake-up timing

The wake-up delay of the DA14585/586 is 4.5 ms to 5 ms. This delay is the same, irrespectively of the transport layer or the wake-up signal being used.

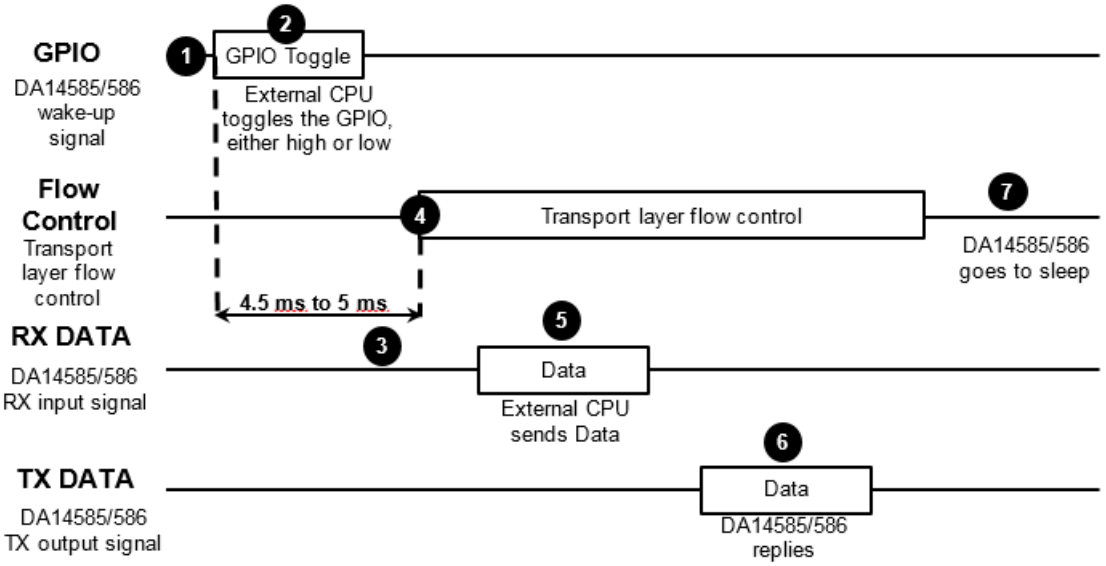


Figure 93: DA14585/586 wake up timing

Figure 93 shows the wake-up timing. From the start of the wake-up signal toggle (GPIO, UART CTS or SPI_EN signal) until the DA14585/586 is ready to receive messages, indicated by the flow control process, 4.5 ms to 5 ms are needed.

Appendix A Writing HEX file into OTP memory

The *SmartSnippets* OTP Programmer tool enables downloading the default firmware into the System RAM and writing a user-defined HEX or BIN file into the OTP memory. The tool can be used for: writing a secondary bootloader in the OTP memory of a DA14585/586.

Figure 94 shows the main screen of the OTP Programmer.

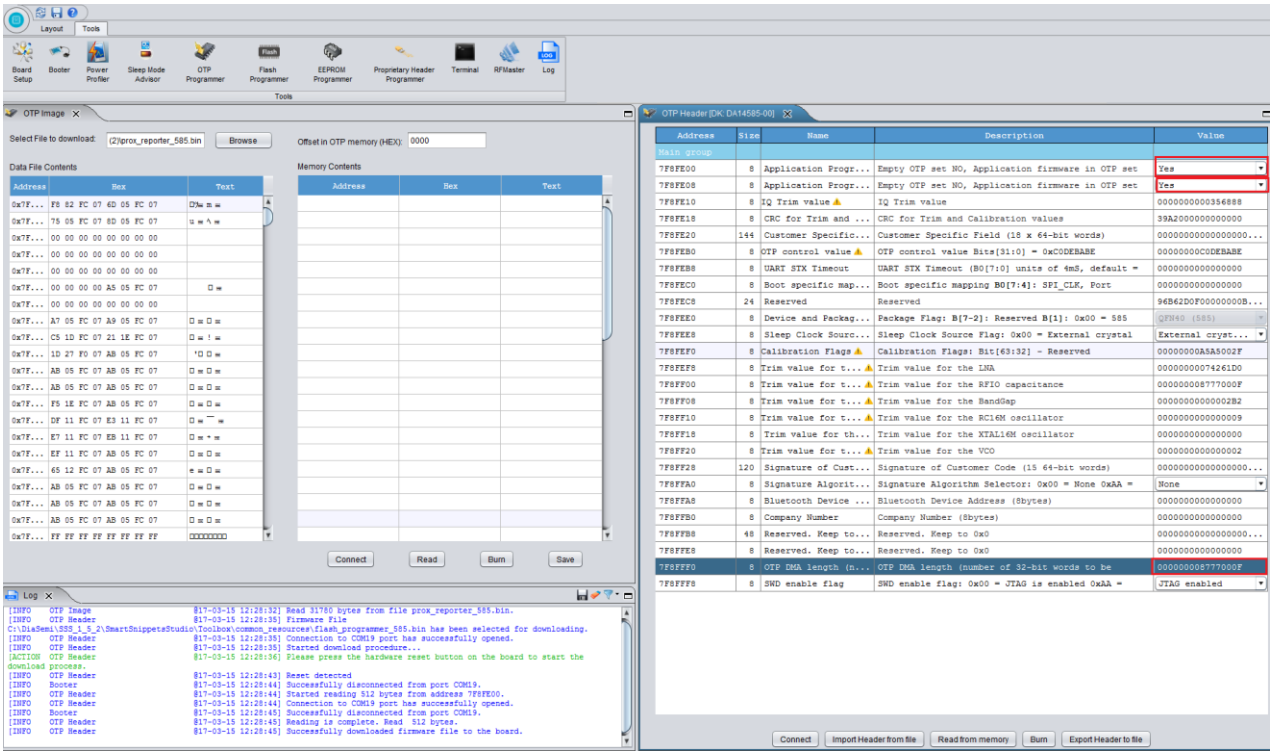


Figure 94: OTP Programmer

The following steps are required for writing the executable *secondary_bootloader.hex* into OTP memory using *SmartSnippets* v3.8 in UART mode:

1. Open *SmartSnippets* and select the chip version.
2. Open the "Board Setup" tool and select the appropriate UART configuration.
3. Open the "OTP Programmer" tool.
4. Write the secondary_*bootloader.hex* into the OTP memory at offset 0. Enable Application Flag 1 and Application Flag 2, set DMA Length (Length = size / 4) and write the OTP header.

Revision History

Revision	Date	Description
1.0	23-Dec-2016	Initial version. Applies to SDK 6.0.1 for DA14585/586.
2.0	24-Mar-2017	Update for the DA14585/586 SDK Release 6.0.2

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 5425 4567

Taiwan
Dialog Semiconductor Taiwan
Phone: +886 281 786 222

Web site:
www.dialog-semiconductor.com

Singapore
Dialog Semiconductor Singapore
Phone: +65 64 8499 29

Hong Kong
Dialog Semiconductor Hong Kong
Phone: +852 3769 5200

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