User Manual

DA1458x Software Developer's Guide

UM-B-050

Abstract

This document describes the steps required to develop BLE applications on the SmartBond™ DA1458x Product Family software platform, specifically for the DA14580/581/583 devices, as supported by the new v5.x SDK series. It guides the developer through a number of pillar examples, acquainting in the developing of BLE applications on the DA1458x 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 DA14580/581/583 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
SPOTA	Software Patching Over The Air
SPOTAR	Software Patching Over The Air Receiver
SUOTA	Software Updating 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]   DA14580 Data sheet, Dialog Semiconductor.

[4]   DA14581 Data sheet, Dialog Semiconductor.

[5]   DA14583 Data sheet, Dialog Semiconductor.

[6]   UM-B-051, DA1458x Software Platform Reference, User Manual, Dialog Semiconductor.

[7]   AN-B-010, DA14580 using SUOTA, Application Note, 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 DA1458x System on Chip (SoC) family of integrated circuit (IC) devices, consisting of the DA14580/581/583, 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™ DA1458x Product Family of devices which are based on the DA1458x System on Chip (SoC) platform.

Developers that are new to the DA1458x System on Chip (SoC) platform are advised to first read Ref. [6], 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. [6], 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 DA1458x Bluetooth® Low Energy devices as well as on how to best utilize the capabilities offered by Dialog's DA1458x 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 DA1458x software architecture, utilizing SDK 5.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 DA1458x software architecture and APIs. It also explains through examples how one can use the peripherals that the DA1458x 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. Ttherefore it is assumed in this document that the developer is familiar with the DA1458x development kits and the µVision Keil software development environment. Depending on the development kit variant, the developer should refer to UM-B-048, Getting Started with DA1458x Development Kits - Basic, User Manual [1], for the basic kits, and UM-B-049, Getting Started with DA1458x 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 DA1458x 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 DA1458x Software Development Kit (SDK) in its latest v5.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 DA14580/581/583 development kit, consisting of:

- **Booter** is used for downloading for downloading hex files to DA14580/581/583 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 DA14580 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 DA1458x on the DK in real time over USB.
- **Sleep Mode Advisor** is used to help users understand how much power their application dissipates in Deep Sleep and Extended 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 (DA14583).

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 PC based software tool to control the link layer of the DA14580/581/583, 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, refer to the "Help Document" which can be found under the "Help" drop down menu of the Connection Manager application.

5 Blinky: Your First DA1458x 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 DA14580/581/583 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 DA1458x and LED blinking.

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

`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	DA1458x DK-Basic	DA1458x 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 following output from program will be visible in the console.
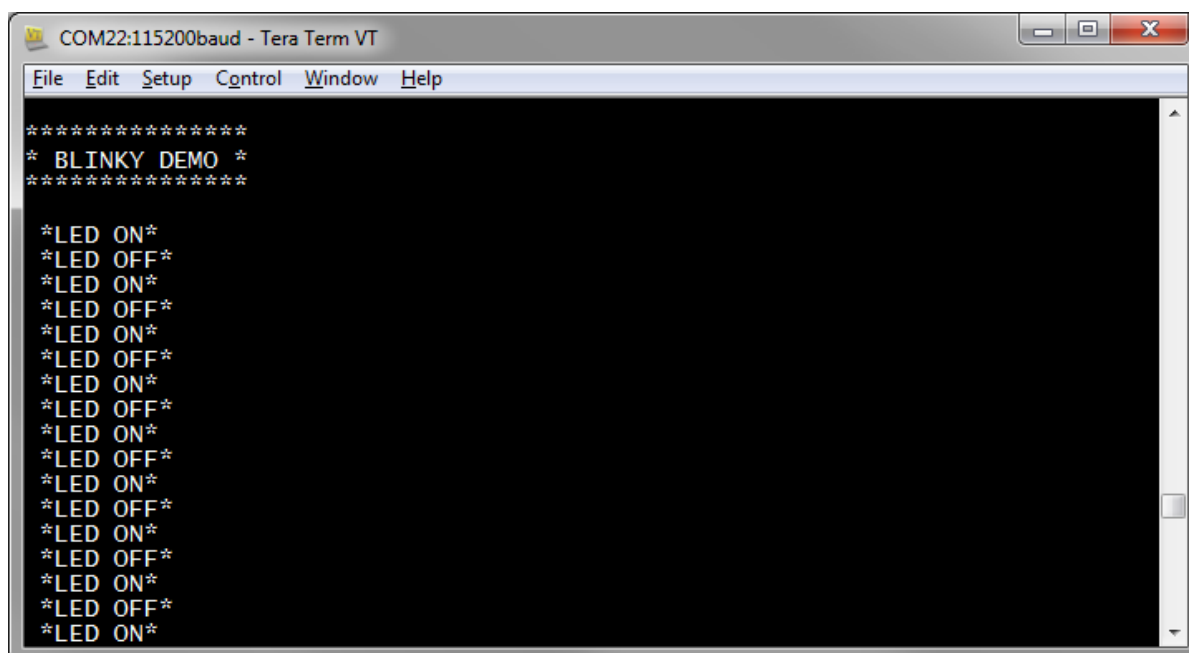
Figure 1: Blinky Example Output Console

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 Patching Over The Air Receiver (SPOTAR) 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.
- Deep Sleep mode after 3 minutes of inactivity.
- Push button.
- Stop Alert indications.
- Exit Deep Sleep 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 DA14580/581/583) 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.
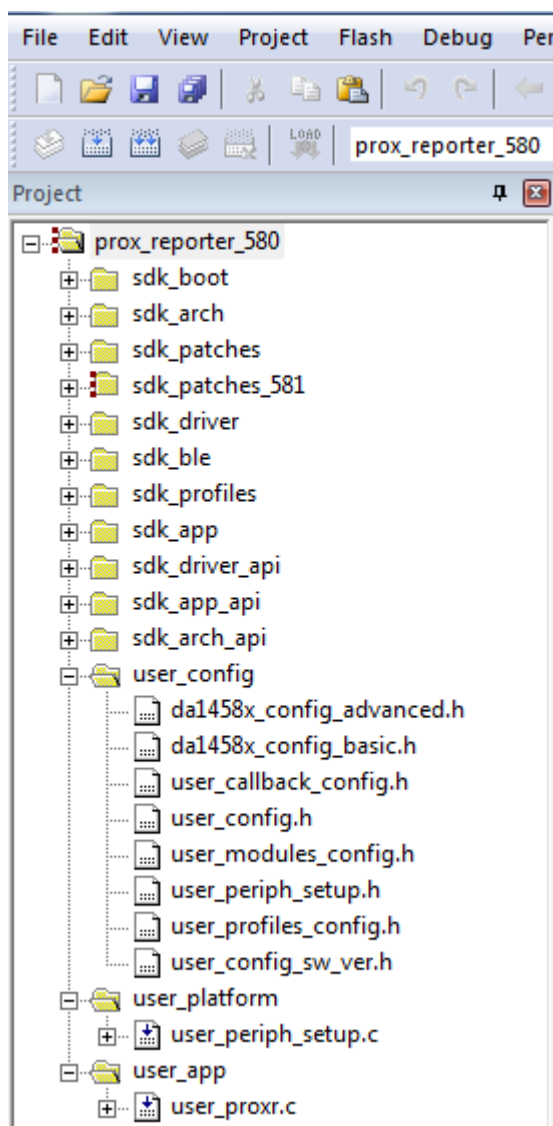
Figure 2: Proximity Reporter Keil Project Layout

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 DA14580/581/583 advanced configuration settings.
- `da1458x_config_basic.h`, holds DA14580/581/583 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:
  - `proxr.h`, includes the Immediate Alert, Link Loss and Tx Power services.
  - `diss.h`, includes the Device Information service.
  - `bass.h`, includes the Battery service.
  - `spotar.h`, includes the Software Patching Over The Air Receiver 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 the 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             = default_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_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_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,
};
```

- 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_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).

- 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).

DA1458x Software Developer's Guide

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`.
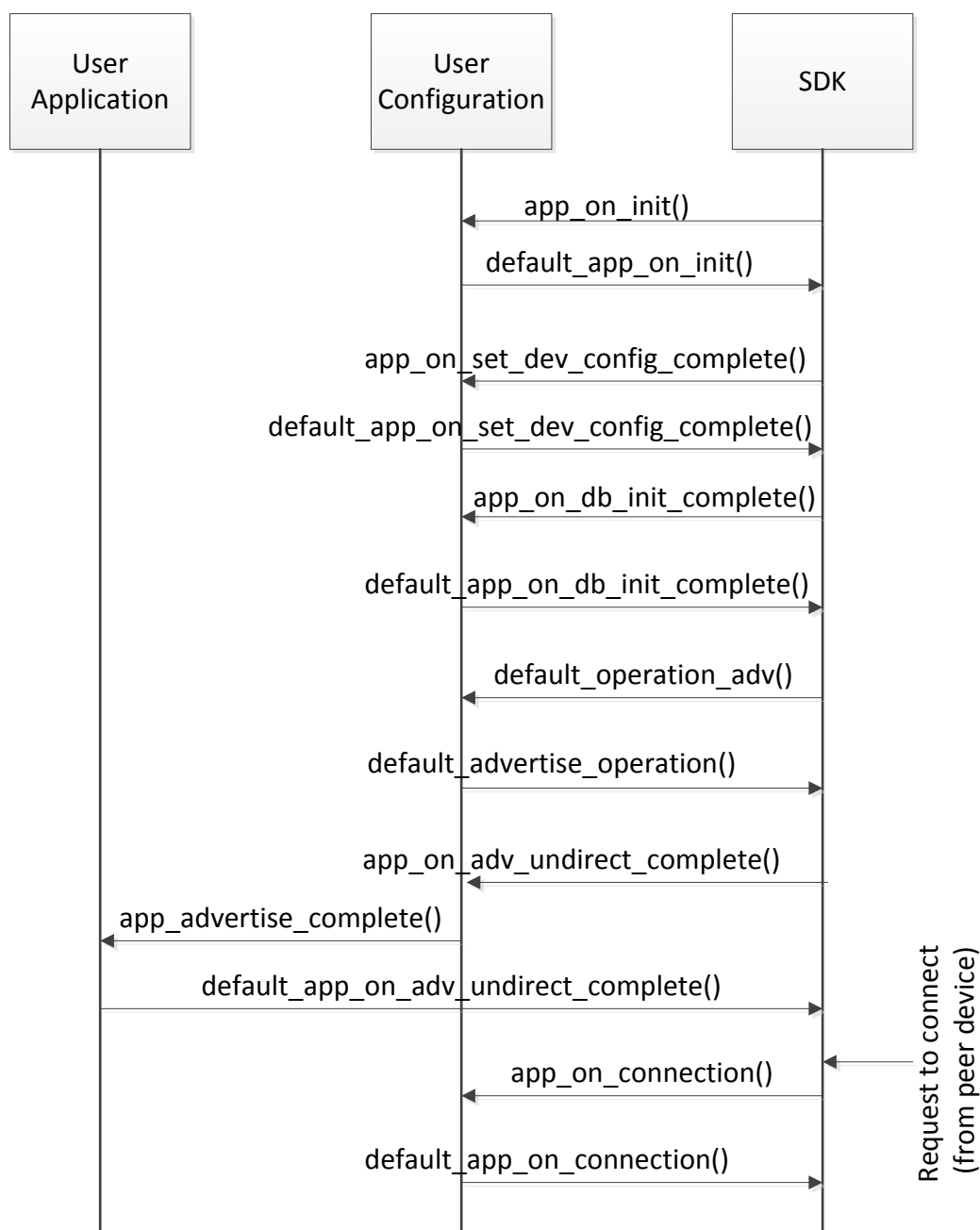
Figure 3: Proximity Reporter - User Application Code Flow

6.9 Building the Project for Different Targets and Development Kits

The Proximity Reporter application can be built for three different target processors, DA14580, DA14581 and DA14583.

The selection is done via the Keil tool as depicted in Figure 4.
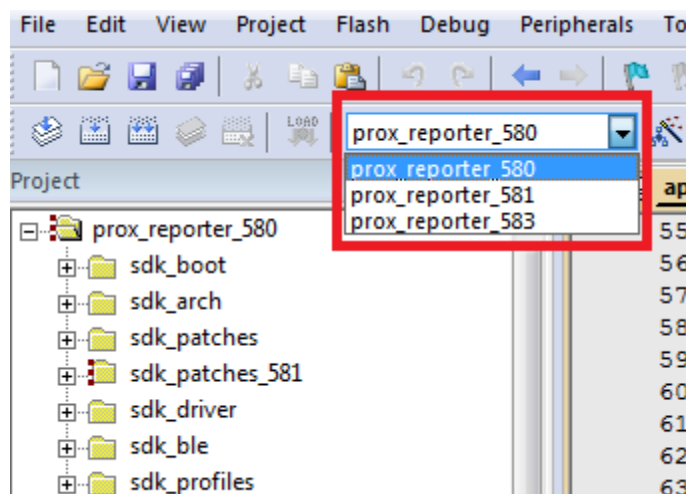
Figure 4: Building the Project for Different Targets

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 5.
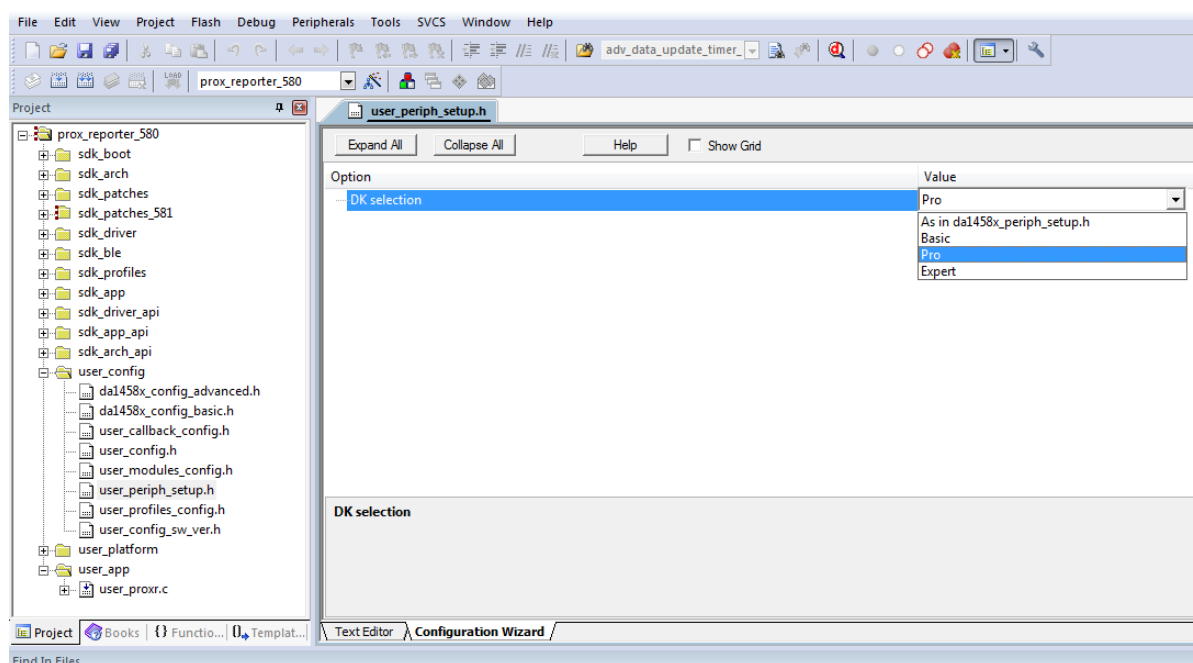
Figure 5: Development Kit Selection for Proximity Reporter Application

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

DA1458x Software Developer's Guide

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 6 shows the result when the iPad/iPod/iPhone device manages to connect to the DA14580/581/583 (the application's advertising device name is **DIALOG-PRXR**).
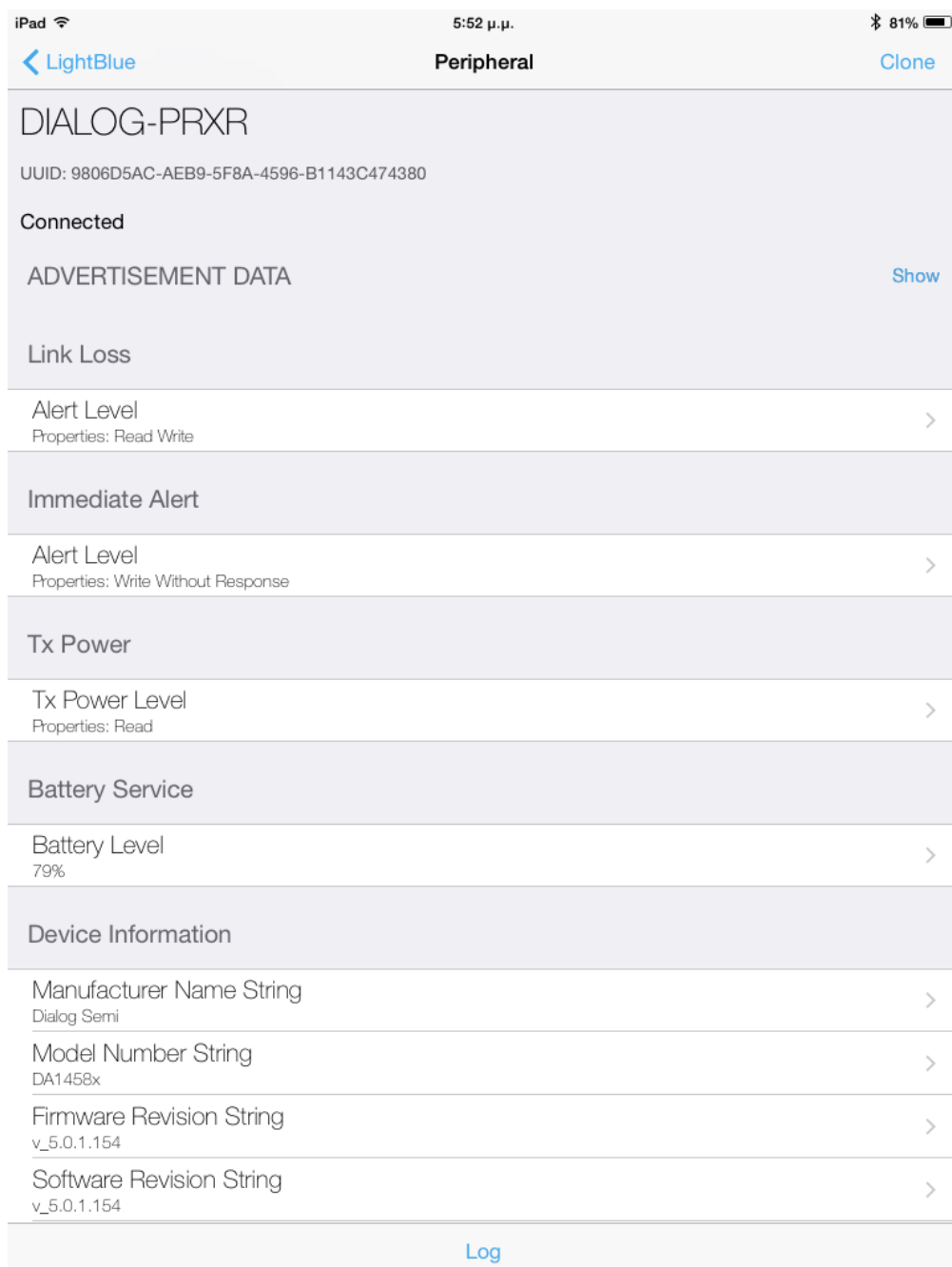
Figure 6: LightBlue Application Connected to Proximity Reporter Application

7   Peripheral Example Applications

The DA1458x 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 DA1458x.

7.1   Introduction

The peripheral example applications demonstrate the DA1458x's peripheral connectivity capabilities such as interfacing to SPI Flash and I2C EEPROM memories as well as using on chip peripherals 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 DA1458x Software Development Kit.

The implementation of the drivers is located in the `sdk\platform\driver` directory. To use the DA1458x 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 DA1458x Development Kits as described in Ref. [1] and Ref. [2].

Prior to running a desired peripheral example, the user has to configure the DA1458x development board accordingly, depending on the required hardware configuration.

Each example contains a subsection which describes the GPIO assignment of the DA1458x development board.

7.4 Configuring the UART Interface on a DA1458x DK

The DA1458x UART2 interface will be used for the UART interaction terminal.

7.5 DA1458x DK-Basic

On a DA1458x 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.

7.6 DA1458x DK-Pro

On a DA1458x 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.

7.7 Using a Serial Port Terminal with a DA1458x 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.

7.8 Connecting to a DA1458x DK-Basic

When the DA1458x 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 7) must be used in the serial port terminal application.
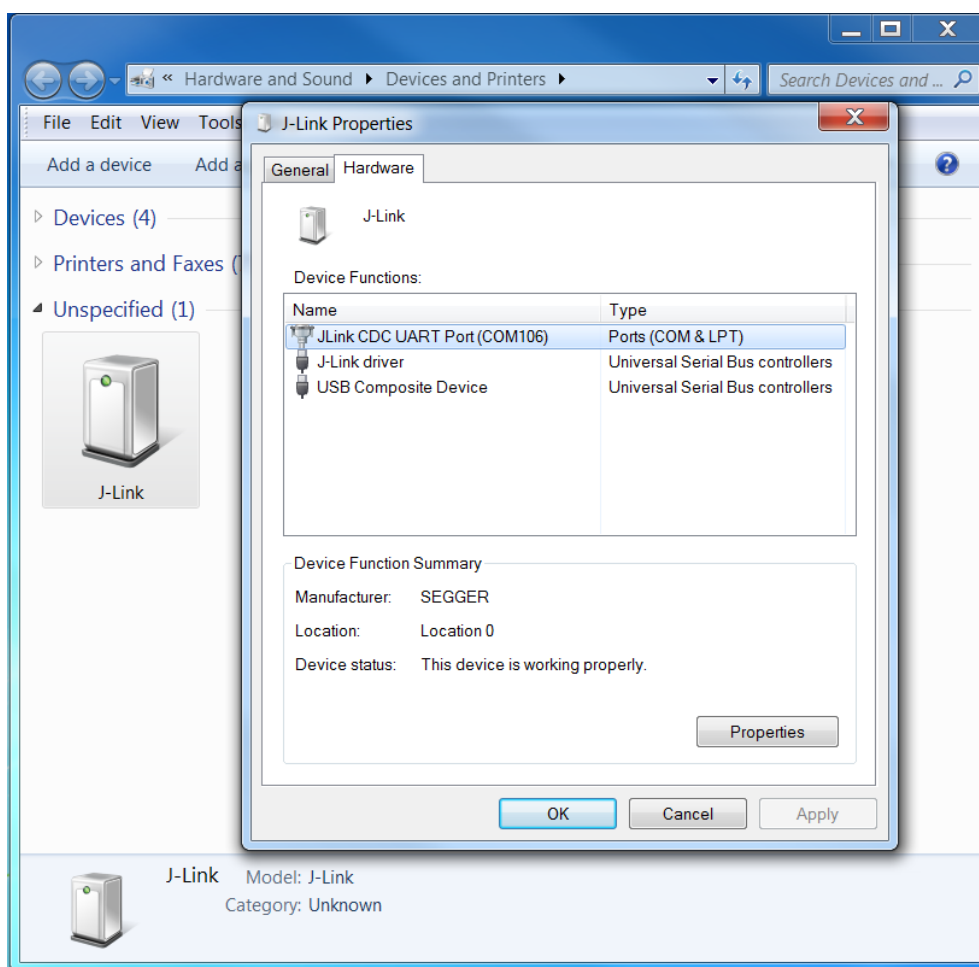
Figure 7: DA1458x DK - Basic Virtual COM Port

7.9   Connecting to a DA1458x DK-Pro

When the DA1458x 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 8) to provide it to a terminal console application.
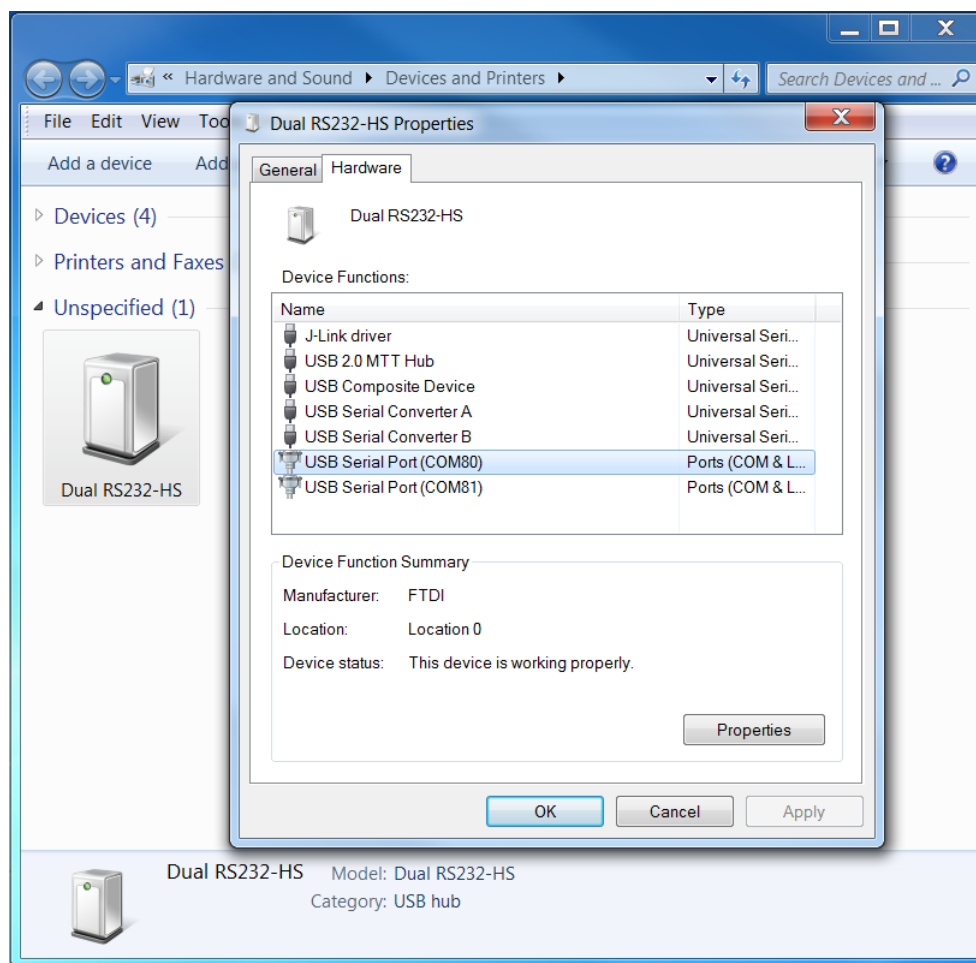
Figure 8: DA1458x DK-Pro Virtual COM Port

7.10 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 `projects\target_apps\peripheral_examples\uart` SDK directory.

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

`projects\target_apps\peripheral_examples\uart\Keil_5\uart.uvprojx`

7.11 Hardware Configuration

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

Table 2: UART Example Jumper Configuration

GPIO	Function	DA1458x DK-Basic	DA1458x 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

7.12 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 9. The uart_test function uses printf_byte and printf_string functions to print the message on the console.
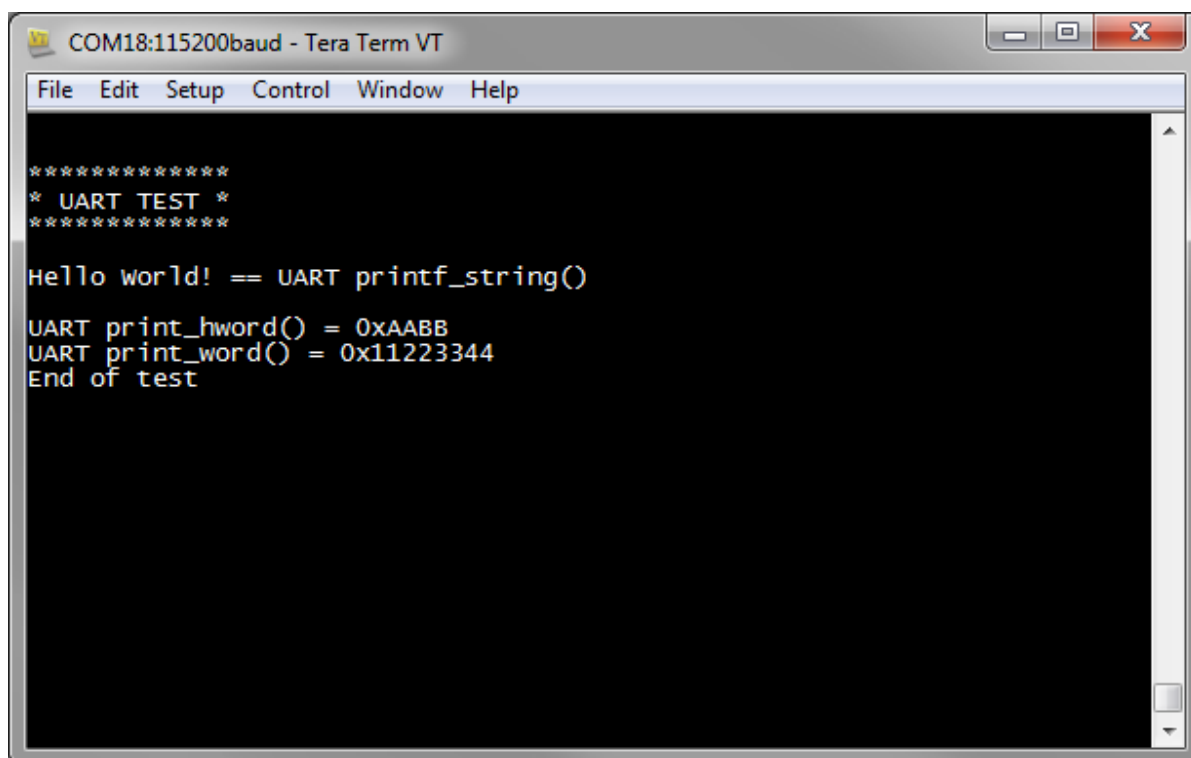
Figure 9: UART Simple Example

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

`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_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.`

7.13   UART2 Asynchronous Example

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

The project is located in the `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:

`projects\target_apps\peripheral_examples\uart2_async\Keil_5\uart2_async.uvprojx`

7.14   Hardware Configuration

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

Table 3: UART2 Example Jumper Configuration

GPIO	Function	DA1458x DK-Basic	DA1458x 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

7.15   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.
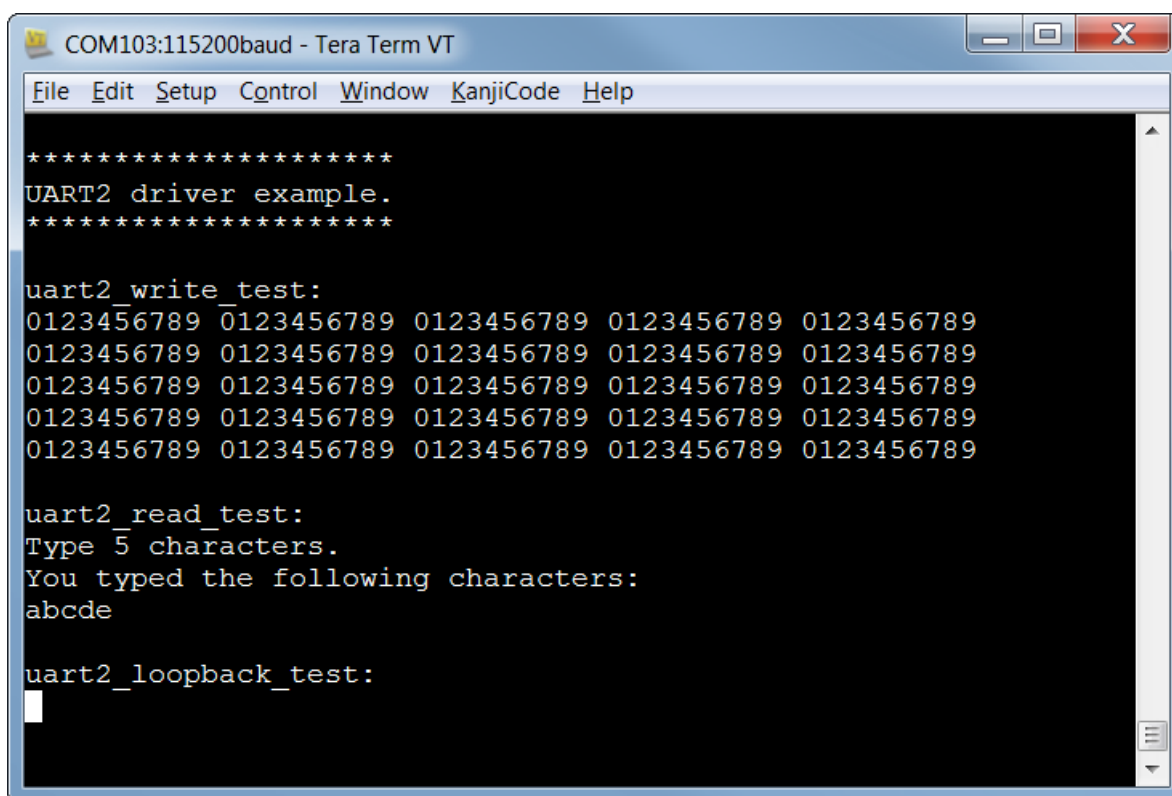
Figure 10: UART2 Example Console Output: Write Test

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 11.

Figure 11: 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 12):

Figure 12: UART2 Example Console Output: Loopback Test

7.16 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 `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:

`projects\target_apps\peripheral_examples\spi\spi_flash\Keil_5\spi_flash.uvprojx`

7.17 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	DA1458x DK-Basic	DA1458x 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	DA1458x DK-Basic	DA1458x 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)

7.18 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 13).

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

```
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's 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, one again the whole memory area is configured as unprotected and a full memory array erase is performed.

Figure 13: 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. [6].

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

```
projects\target_apps\peripheral_examples\\spi\spi_flash\src\main.c.
```

DA1458x Software Developer's Guide

7.19 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 `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:

`projects/target_apps/peripheral_examples/i2c/i2c_eeprom/Keil_5/i2c_eeprom.uvprojx`
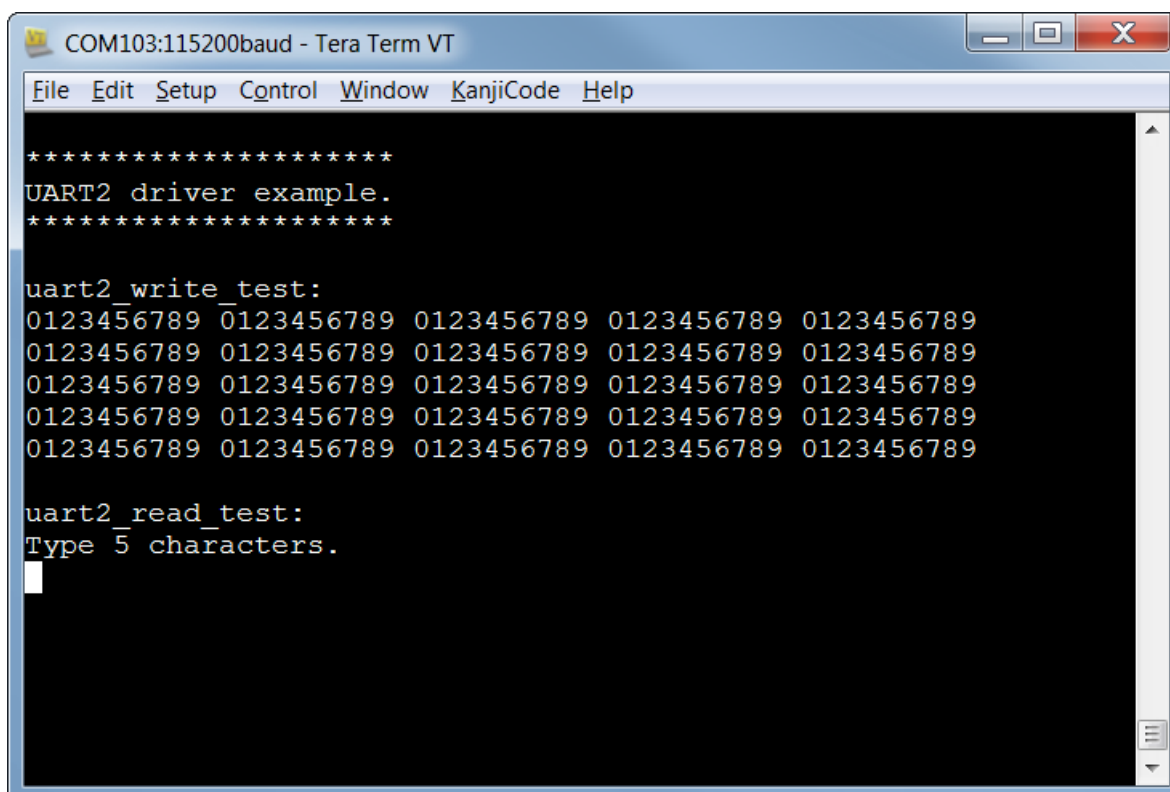
7.20 Hardware Configuration

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

Table 6: I2C EEPROM Example Jumper Settings

GPIO	Function	DA1458x DK-Basic	DA1458x 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 DA1458x 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`.

7.21 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 14.

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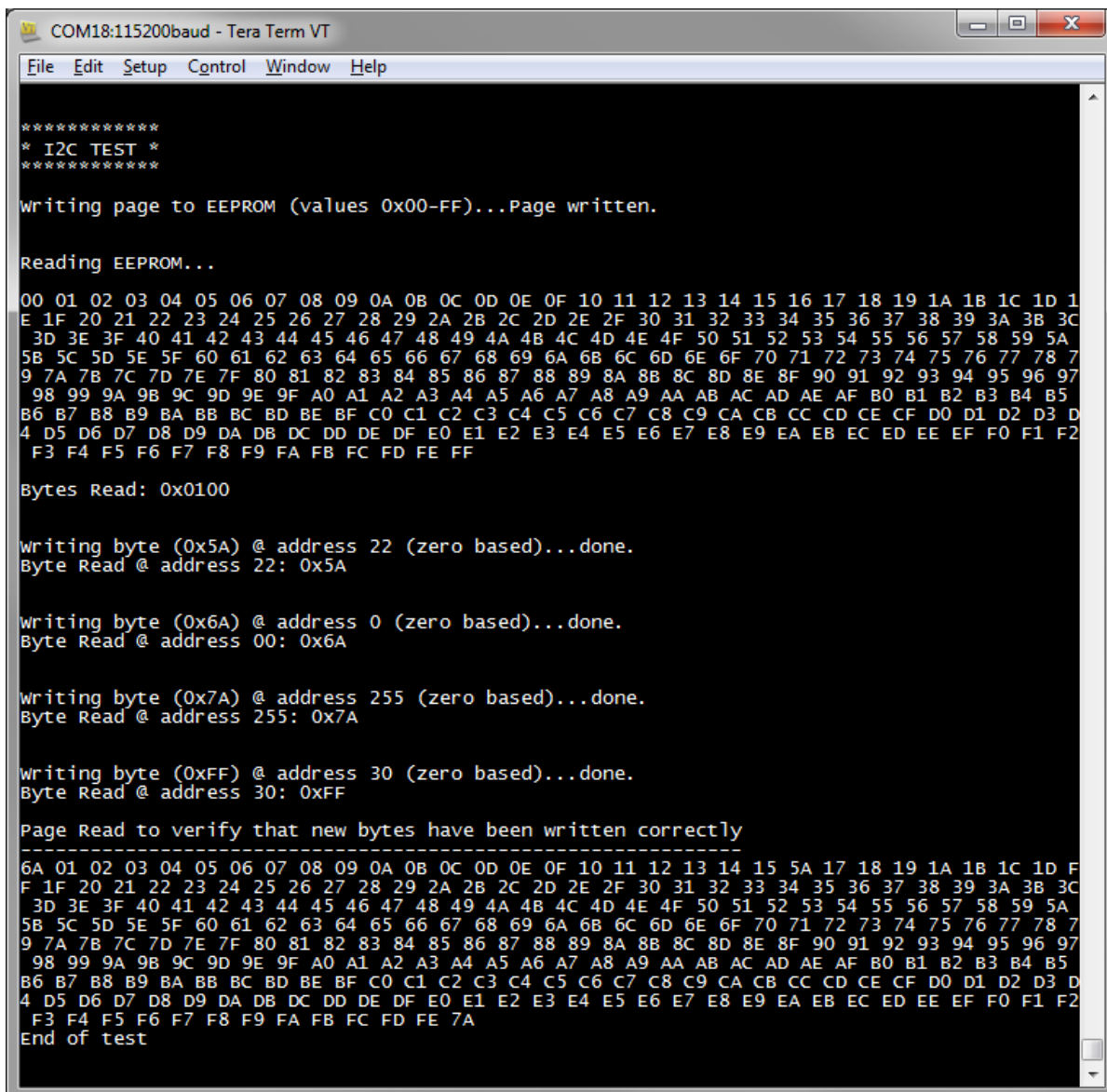
The user can also select the I2C module's parameters, such as slave address, speed mode, address mode and addressing scheme (1-byte/2-byte):

```
#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 14.

Figure 14: 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. [6].

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

`projects\target_apps\peripheral_examples\i2c\i2c_eeprom\src\main.c.`

7.22 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 `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`

7.23 Hardware Configuration

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

Table 7: Quadrature Decoder Example Jumper Settings

GPIO	Function	DA1458x DK-Basic	DA1458x 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.

7.24 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 15.
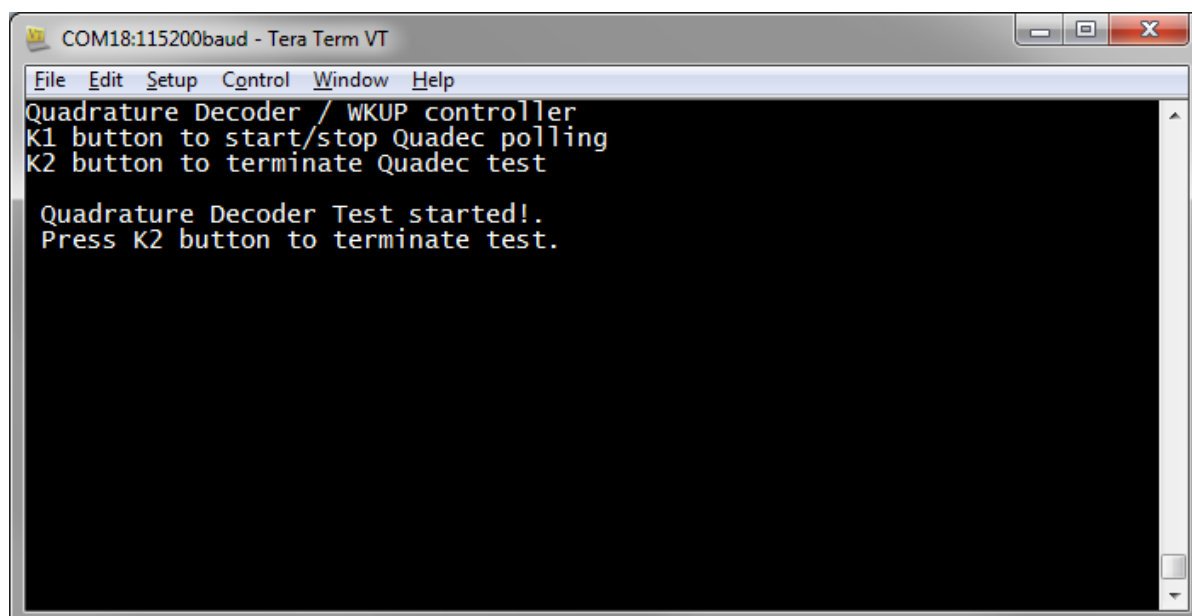
Figure 15: Quadrature Decoder Example

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 16).
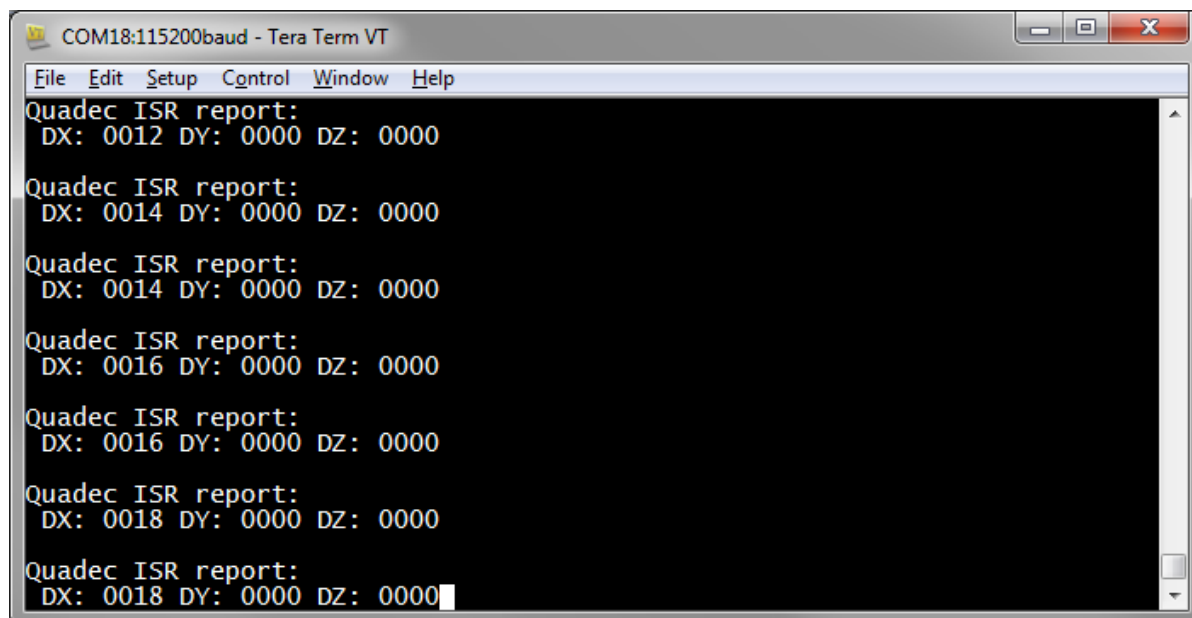
Figure 16: Quadrature Decoder ISR-Only Reports

If at any time the K1 button is pressed (the user should make sure that 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 17). If the quadrature decoder is activated, a mixture of ISR and polling generated reports are displayed (see Figure 18).
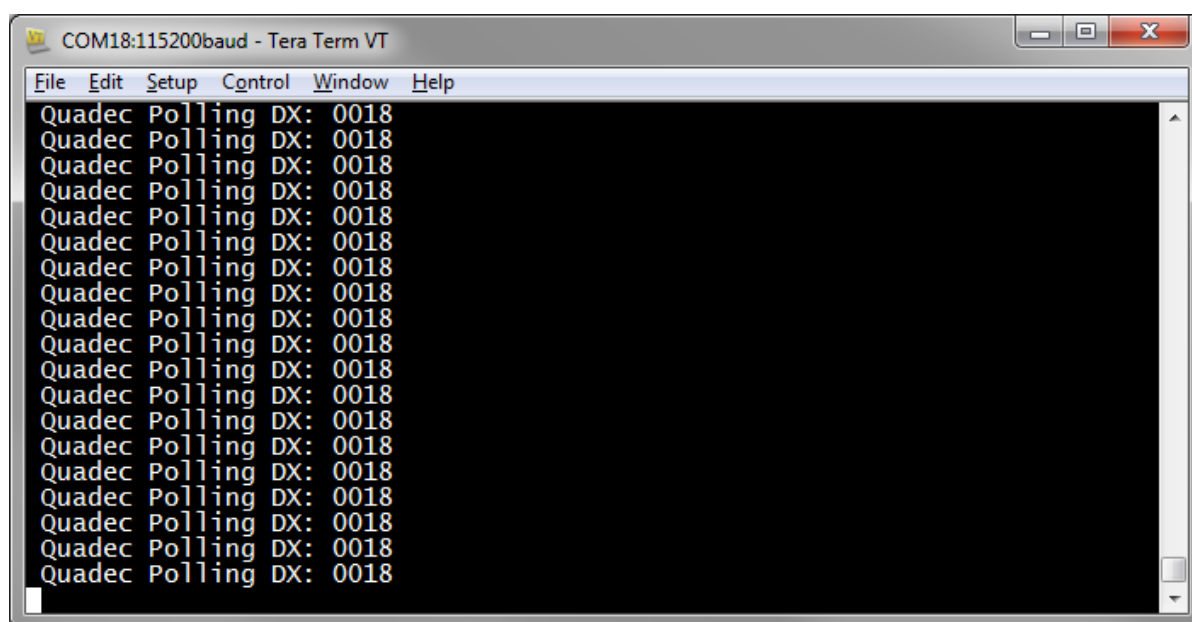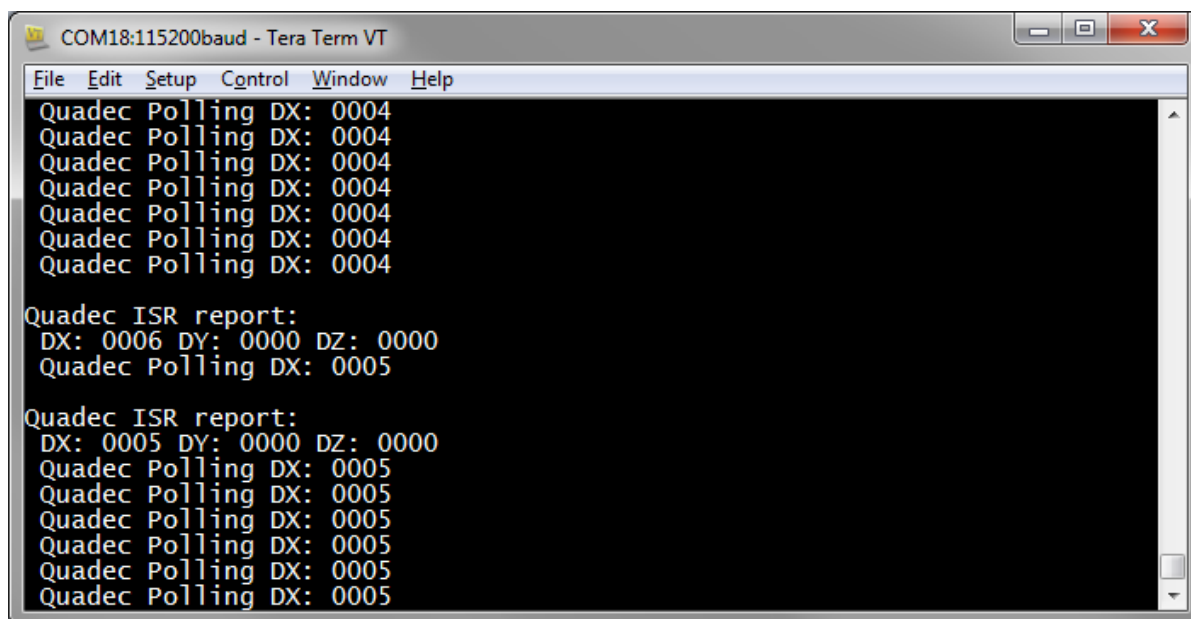
Figure 17: Quadrature Decoder Polling-Only Reports

Figure 18: 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:

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

7.25   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 projects\target_apps\peripheral_examples\systick SDK directory.

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

projects\target_apps\peripheral_examples\systick\Keil_5\systick.uvprojx

7.26   Hardware Configuration

This example does not use the UART terminal.

Table 8. Systick Example Jumper Settings

GPIO	Function	DA1458x DK-Basic	DA1458x DK-Pro
P1_0	LED	Connect J9.1 – J9.2	Connect J9.1 – J9.2

7.27   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:

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

7.28   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 `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:

`projects\target_apps\peripheral_examples\timer0\timer0_pwm\Keil_5\timer0_pwm.uvprojx`

7.29   Hardware Configuration

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

Table 9: Timer0 Example Jumper Settings

GPIO	Function	DA1458x DK-Basic	DA1458x 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).

7.30   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 19).
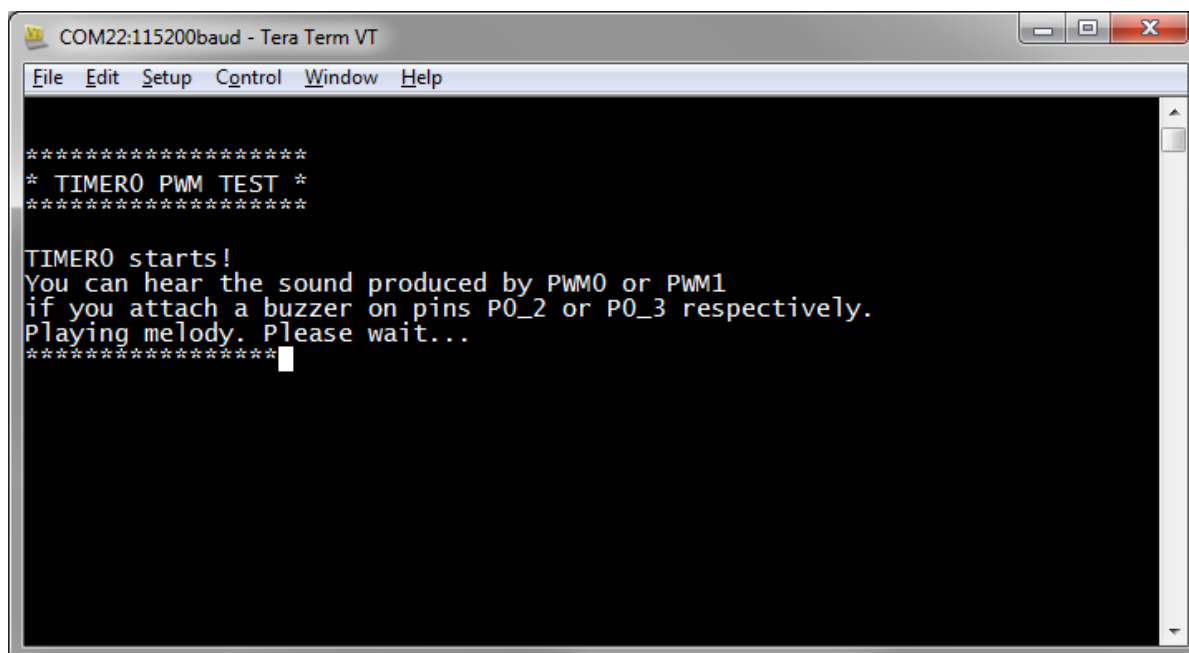
Figure 19: TIMER0 (PWM0, PWM1) Test Running

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:

```
projects\target_apps\peripheral_examples\timer0\timer0_pwm\src\main.c.
```

7.31 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 `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:

```
projects\target_apps\peripheral_examples\timer0\timer0_genera\Keil_5\timer0_general.uv
projx
```

7.32 Hardware Configuration

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

Table 10: Timer0 General Example Jumper Settings

GPIO	Function	DA1458x DK-Basic	DA1458x 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

7.33 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 20.
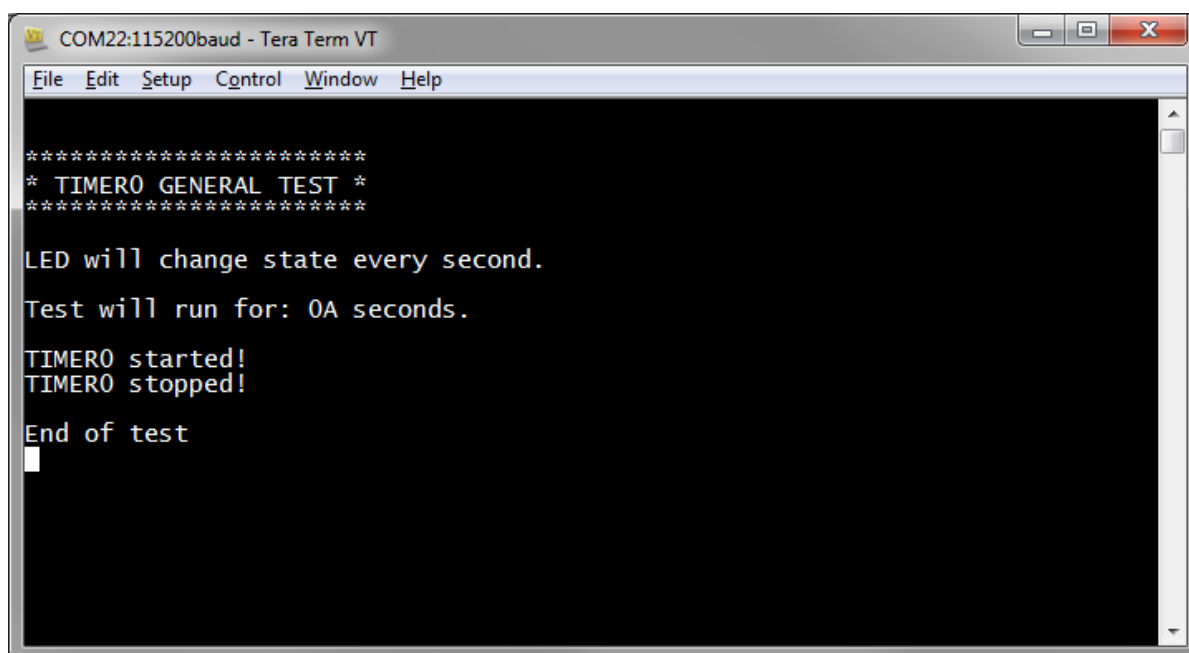
Figure 20: TIMER0 General Test Completed

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

```
projects\target_apps\peripheral_examples\timer0\timer0_general\src\main.c.
```

7.34 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 `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:

```
projects\target_apps\peripheral_examples\timer2\timer2_pwm\Keil_5\timer2_pwm.uvprojx
```

7.35 Hardware Configuration

The common UART terminal configuration described in 7.4 is required.

Table 11: TIMER2 Example Jumper Settings

GPIO	Function	DA1458x DK-Basic	DA1458x 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.

7.36 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 7.35 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 be changed in a loop function. The screen shown in Figure 21 will appear.
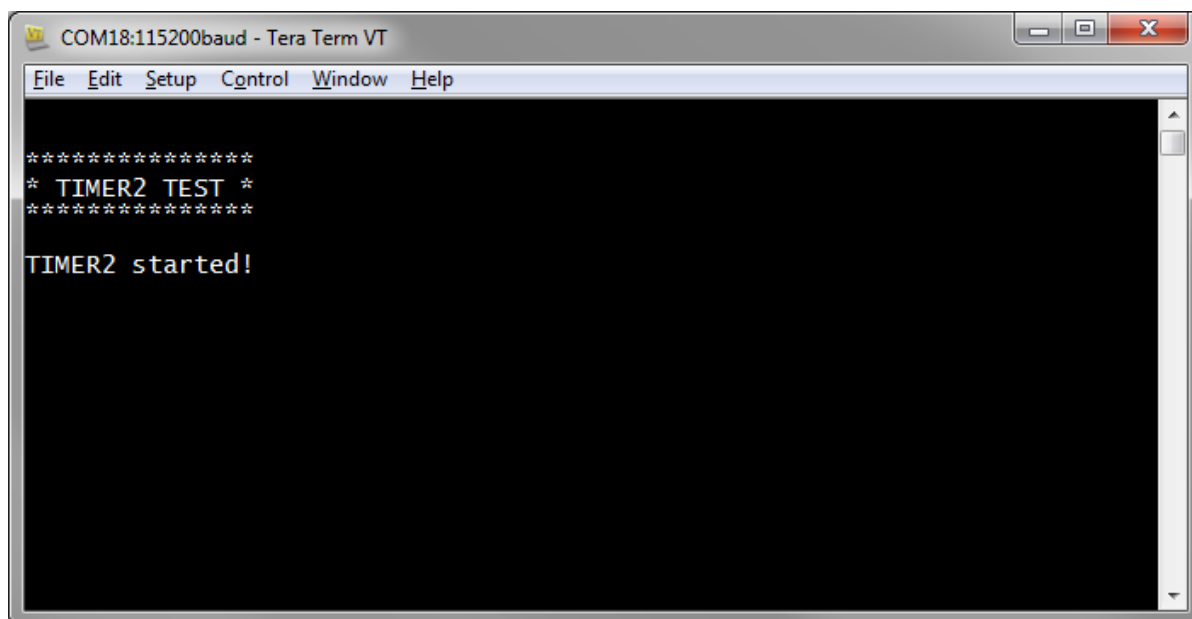
Figure 21: TIMER2 (PWM2, PWM3, PWM4) Test Running

After the test has been completed, the screen shown in Figure 22 will then appear.
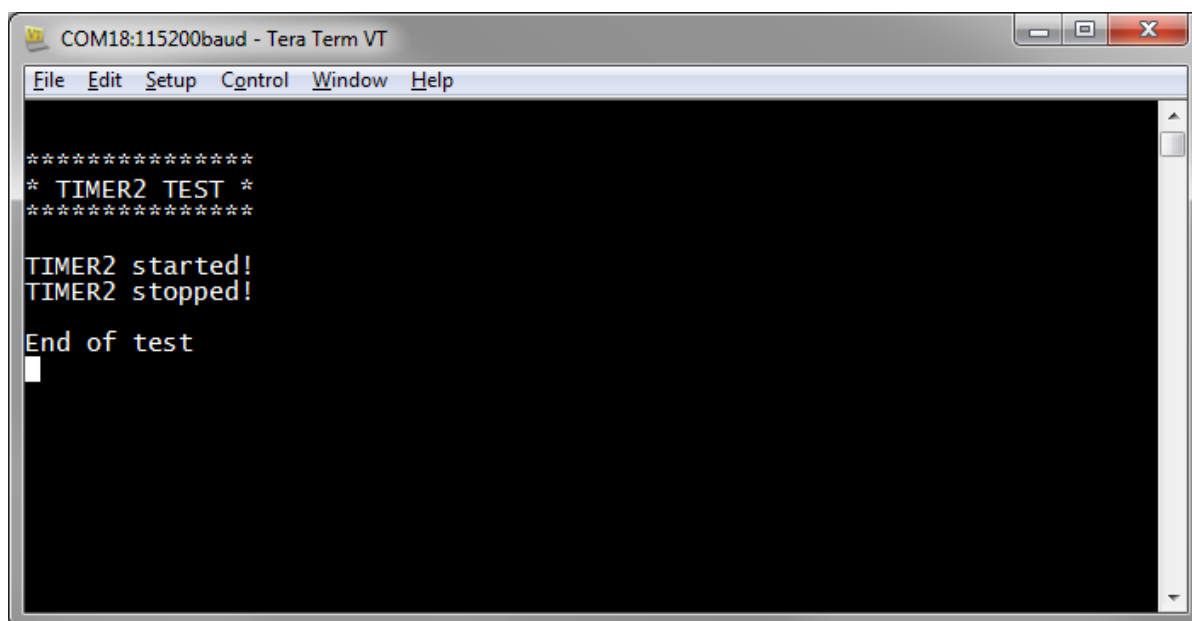
Figure 22: TIMER2 (PWM2, PWM3, PWM4) Test Completed

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

`projects\target_apps\peripheral_examples\timer2\timer2_pwm\src\main.c.`

7.37 Battery Example

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

The project is located in the `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:

`projects\target_apps\peripheral_examples\adc\batt_lvl\Keil_5\batt_lvl.uvprojx`

7.38 Hardware Configuration

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

**Table 12: Battery Example Jumper Settings**

GPIO	Function	DA1458x DK-Basic	DA1458x 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 DA1458x DK for CR2032 battery operation.

7.39 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 23).
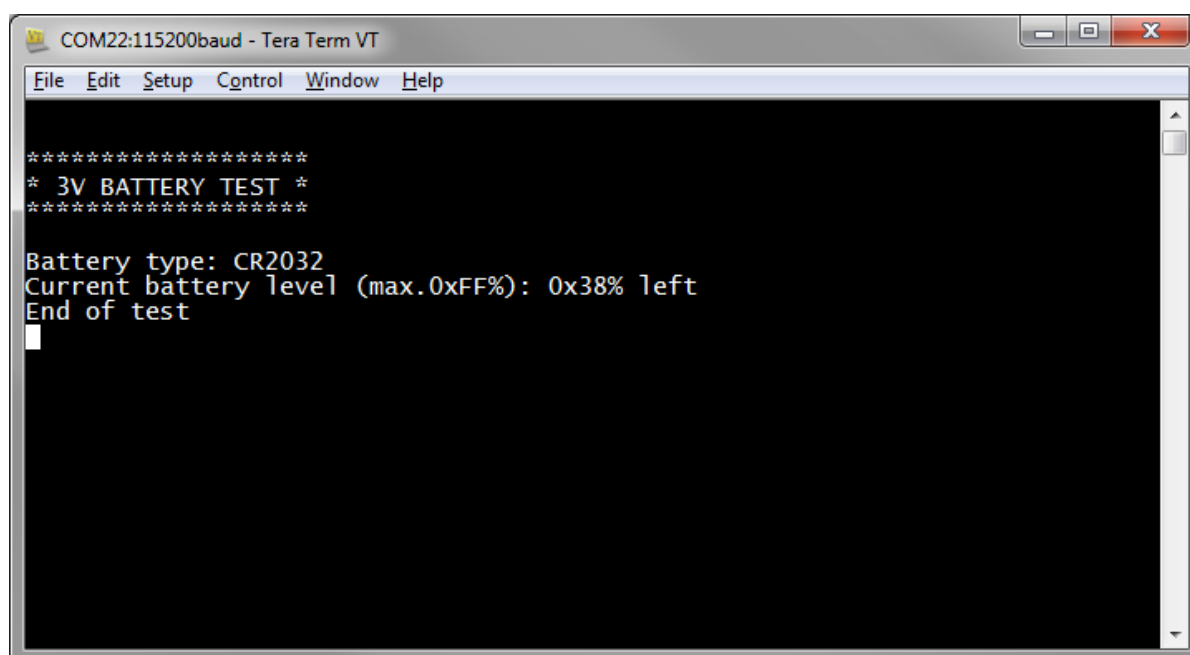
Figure 23: Battery Example

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 [3].

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

```
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 DA1458x SDK 5.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 24. 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 DA14580/581/583 software platform and devices.
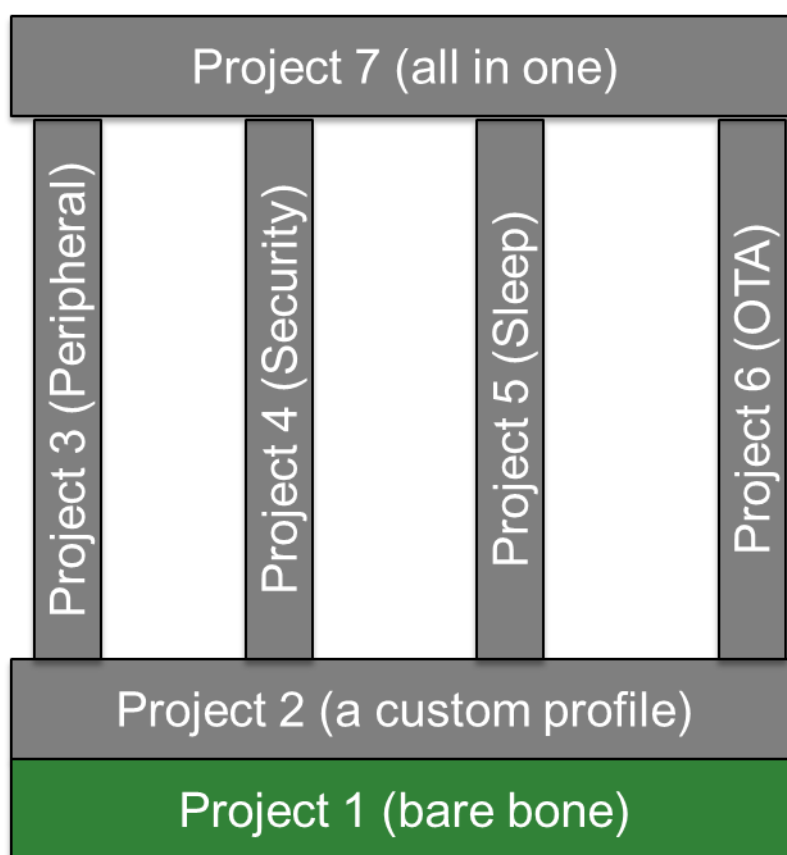
Figure 24: Pillar Example Projects

8.2   Pillar 1 (Bare Bone)
8.3   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.

8.4   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 every 10 s as follows:
  - Cancel on-going advertising operation
  - Change proprietary data (increment by 1 the 16-bit counter)
  - Restart advertising update timer
  - Start advertising

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

8.5   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.

8.6 Loading the Project

The Pillar 1 application is developed under the Keil v5 tool. The Keil project file is the:

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

Figure 25 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.
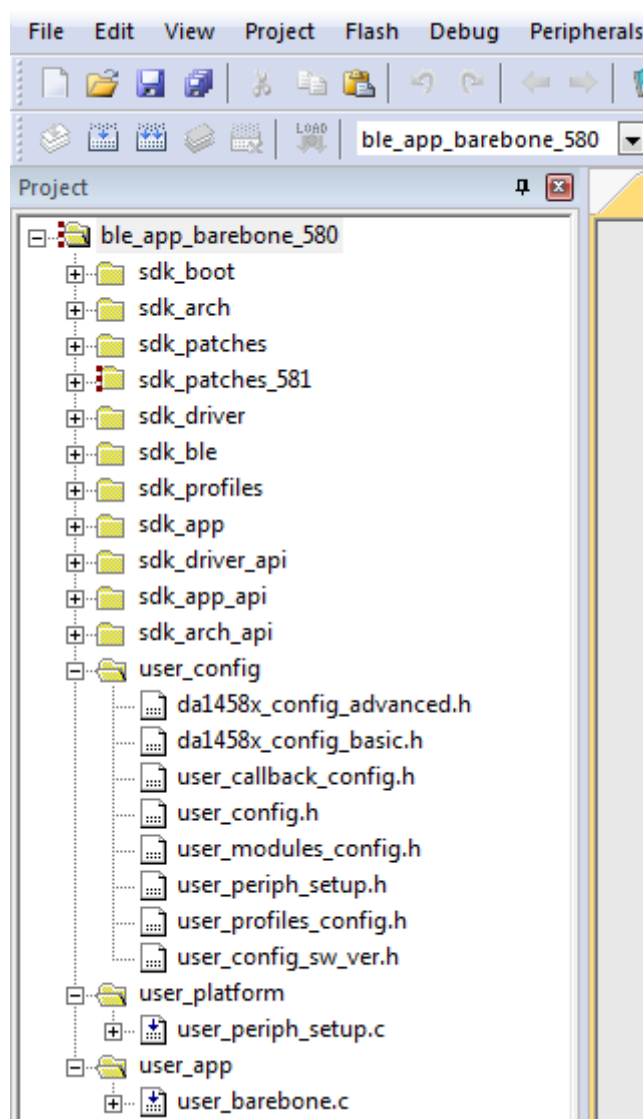
Figure 25: Pillar 1 Keil Project Layout

8.7 Going Through the Code
8.8 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 DA14580/581/583 advanced configuration settings.

- `da1458x_config_basic.h`, holds DA14580/581/583 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:

  - `diss.h`, includes the Device Information 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 Development Kit.

8.9 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,
#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 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`.

8.10   BLE Application Abstract Code Flow

Figure 26 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 26: Pillar 1 Application - User Application Code Flow

8.11 Building the Project for Different Targets and Development Kits

The Pillar 1 application can be built for three different target processors: DA14580, DA14581 and DA14583. The selection is done via the Keil tool as depicted in Figure 27.
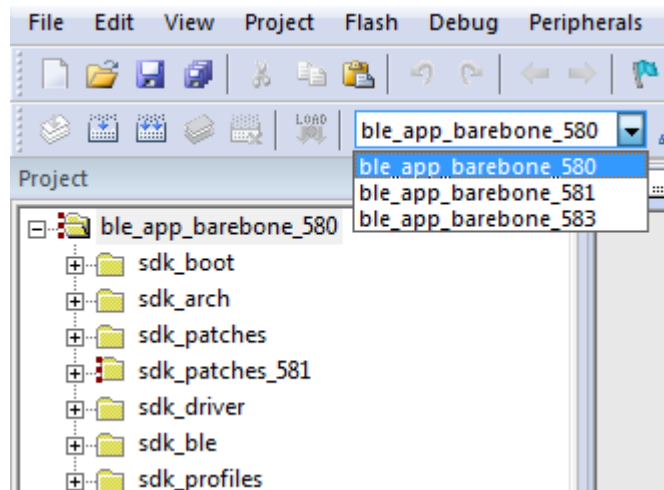
Figure 27: Building the Project for Different Targets

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 28.
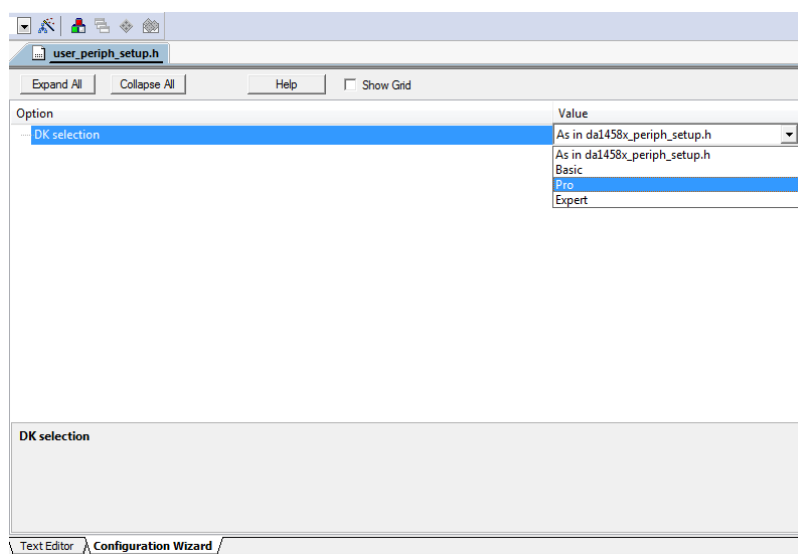
Figure 28: Development Kit Selection for Pillar 1 Application

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

8.12   Interacting with BLE Application
8.13   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 29 shows the result when the iPad/iPod/iPhone device manages to connect to the DA14580/581/583 (the application's advertising device name is **DIALOG-BRBN**).
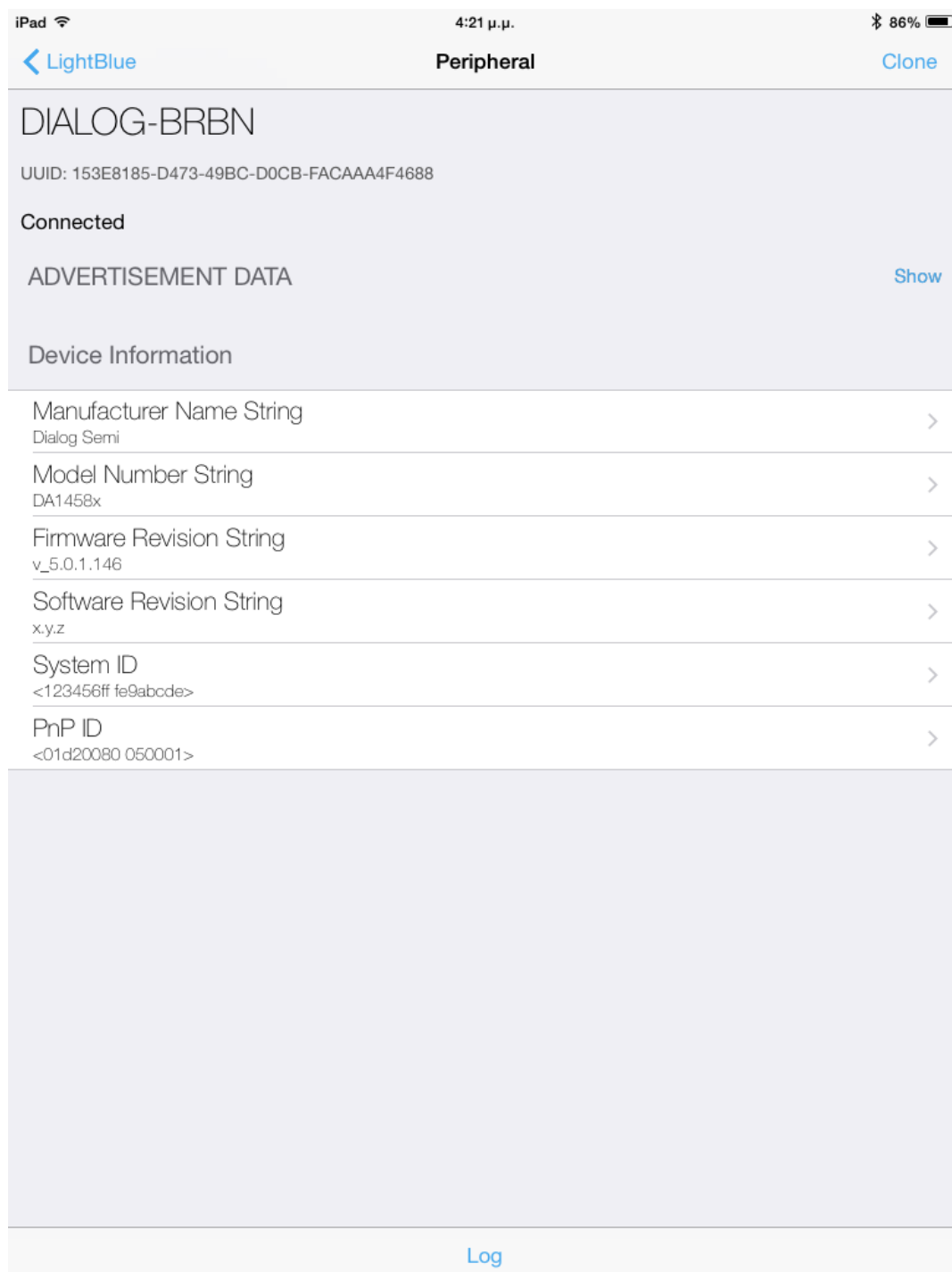
Figure 29: LightBlue Application Connected to Pillar 1 Application

8.14 Pillar 2 (Custom Profile)
8.15 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.

8.16 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:
   ○ 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.

8.17 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.

8.18 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 30 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.
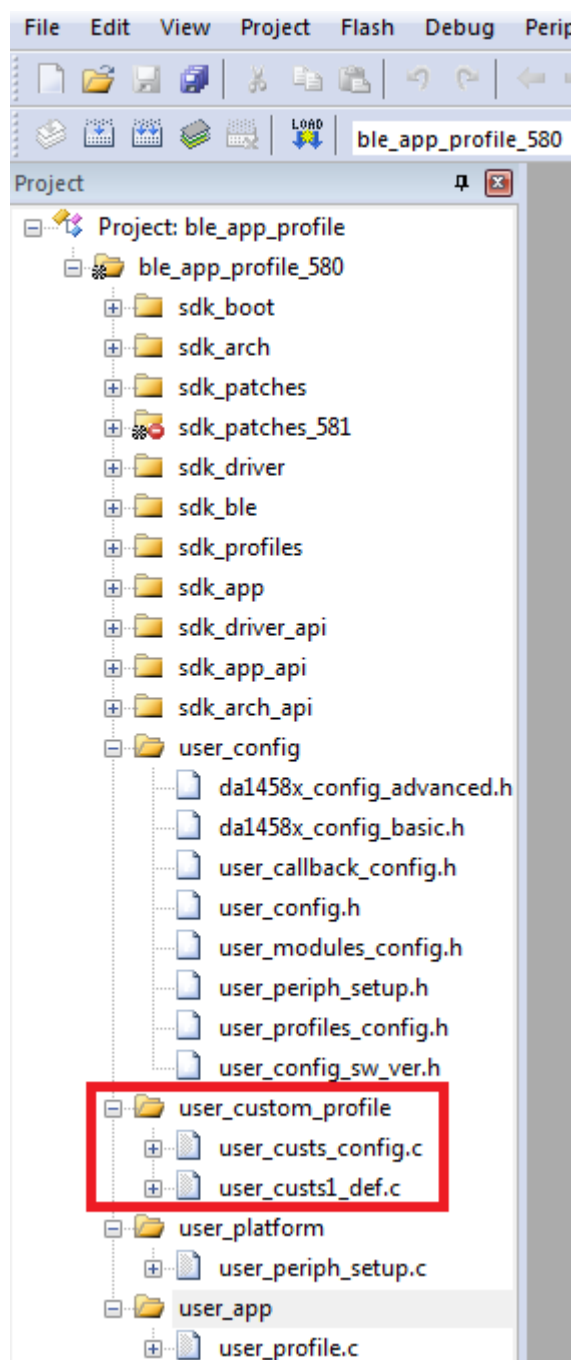
Figure 30: Pillar 2 Keil Project Layout

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

8.19 Going Through the Code
8.20 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 DA14580/581/583 advanced configuration settings.
- `da1458x_config_basic.h`, holds DA14580/581/583 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:
  - `diss.h`, includes the Device Information service.
  - `custs1.h`, 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.

8.21 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,
#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 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.

8.22 BLE Application Abstract Code Flow

Figure 31 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`.
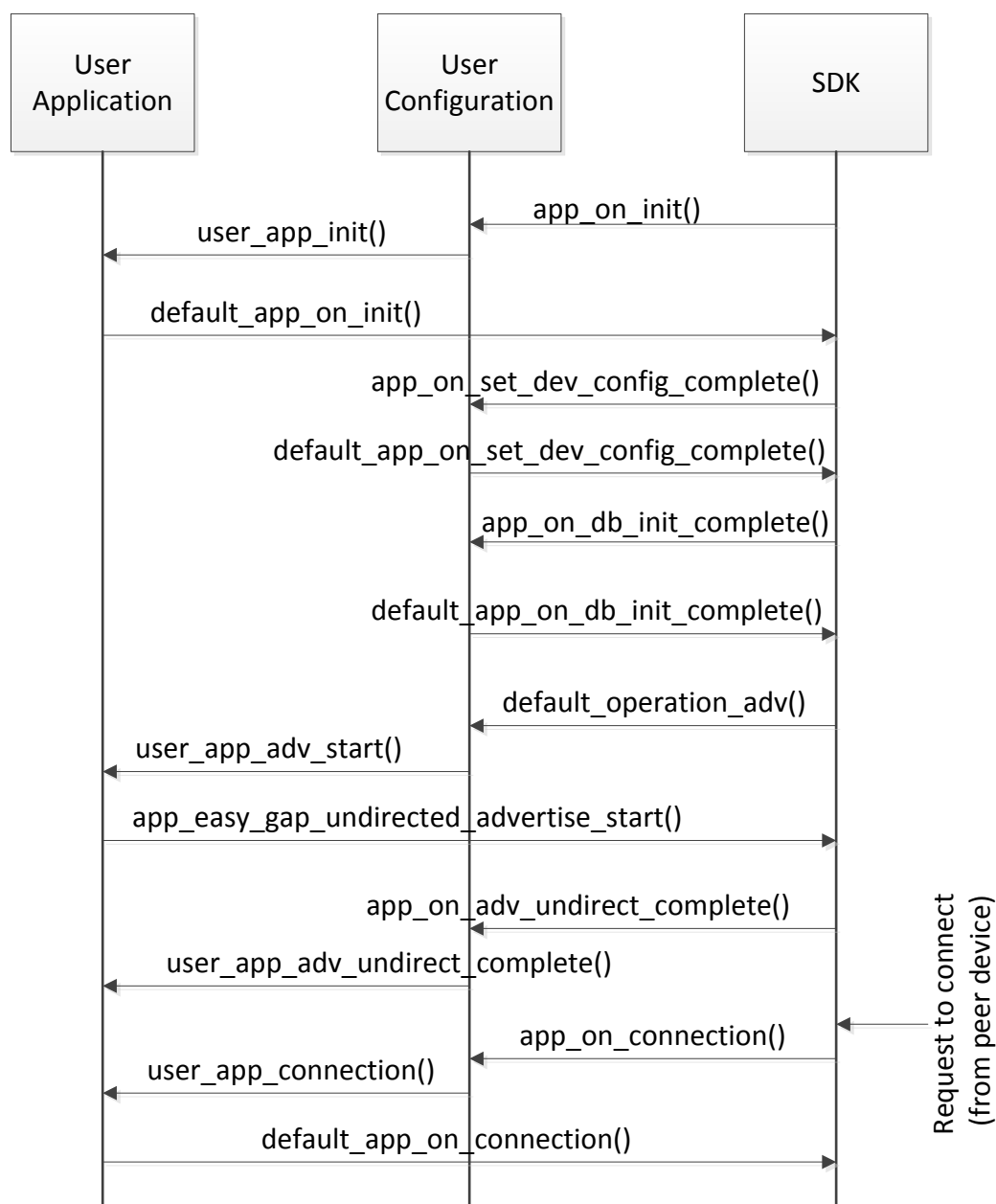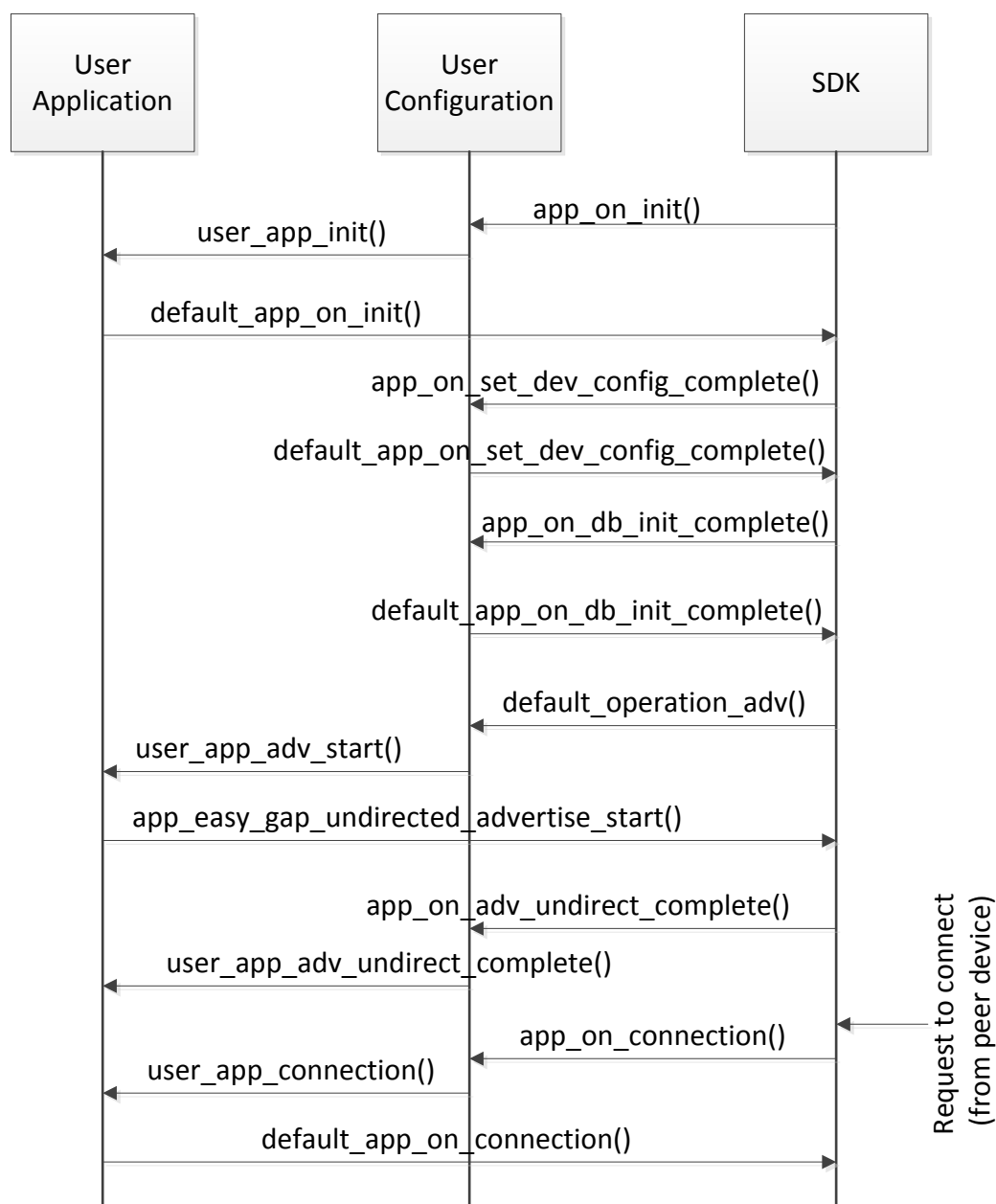

Figure 31: Pillar 2 Application - User Application Code Flow

8.23 Building the Project for Different Targets and Development Kits

The Pillar 2 application can be built for three different target processors: DA14580, DA14581 and DA14583. The selection is done via the Keil tool as depicted in Figure 32.
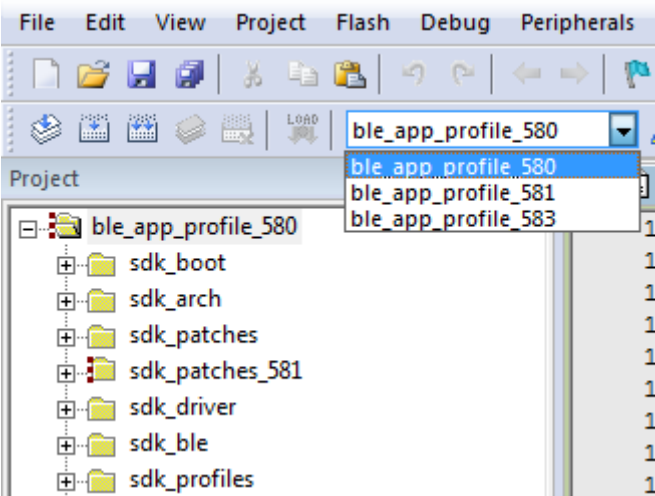
Figure 32: Building the Project for Different Targets

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 33.
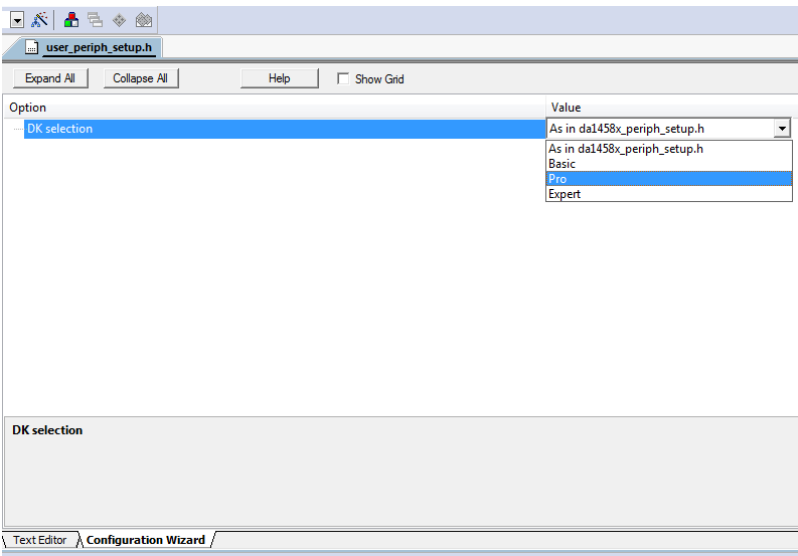
Figure 33: Development Kit Selection for Pillar 2 Application

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

8.24 Interacting with BLE Application
8.25 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 34 shows the result when the iPad/iPod/iPhone device manages to connect to the DA14580/581/583 (the application's advertising device name is **DIALOG-PRFL**).
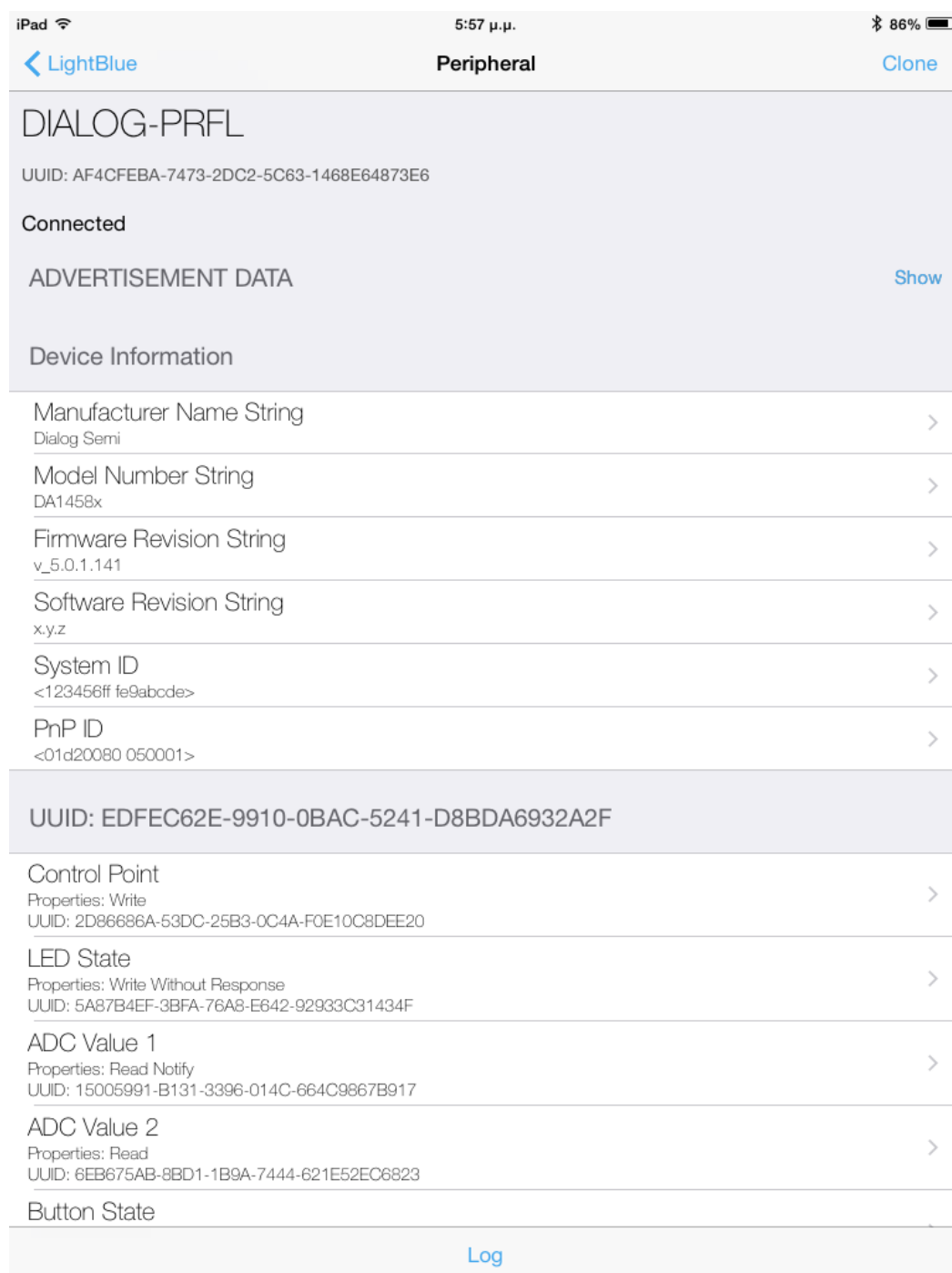
Figure 34: LightBlue Application Connected to Pillar 2 Application

8.26 Pillar 3 (Peripheral)
8.27 Application Description

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.

8.28 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.

8.29 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).

DA1458x Software Developer's Guide

The selected LED (port and pin number of the DA14580/581/583) 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.

8.30 Loading the Project

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

`projects\target_apps\ble_examples\ble_app_peripheral\ble_app_peripheral.uvprojx`.

Figure 35 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.
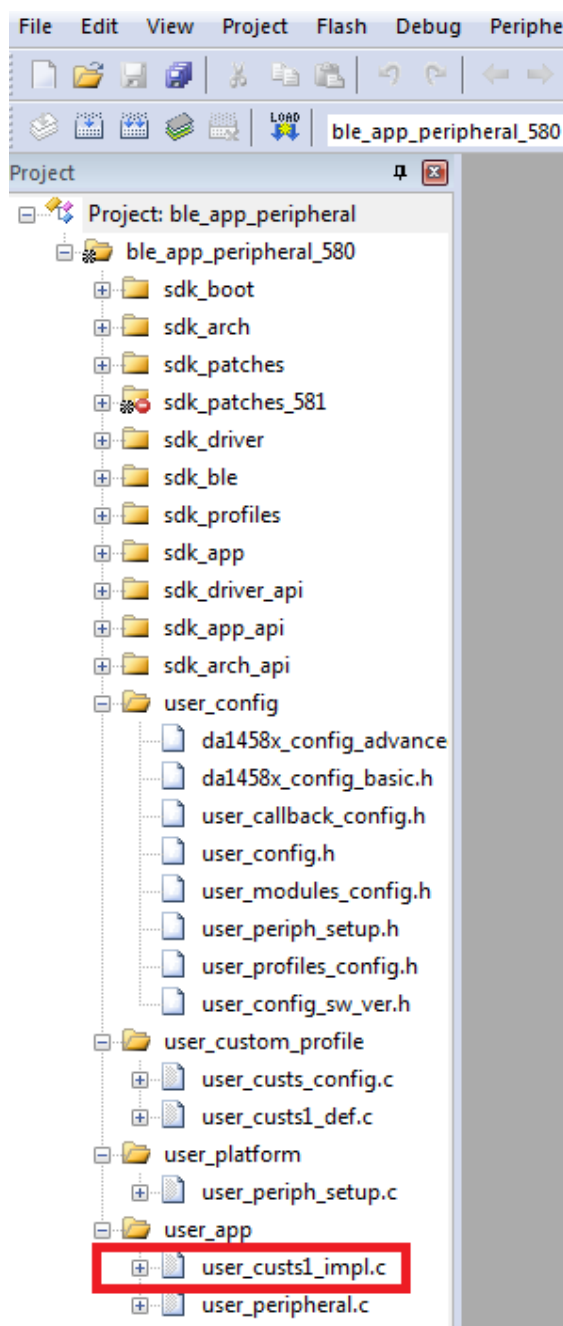
Figure 35: Pillar 3 Keil Project Layout

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

8.31 Going Through the Code
8.32 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 DA14580/581/583 advanced configuration settings.

- `da1458x_config_basic.h`, holds DA14580/581/583 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:
  - `diss.h`, includes the Device Information service.
  - `custs1.h`, 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.

8.33 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;
```

8.34 BLE Application Abstract Code Flow

Figure 36 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`.
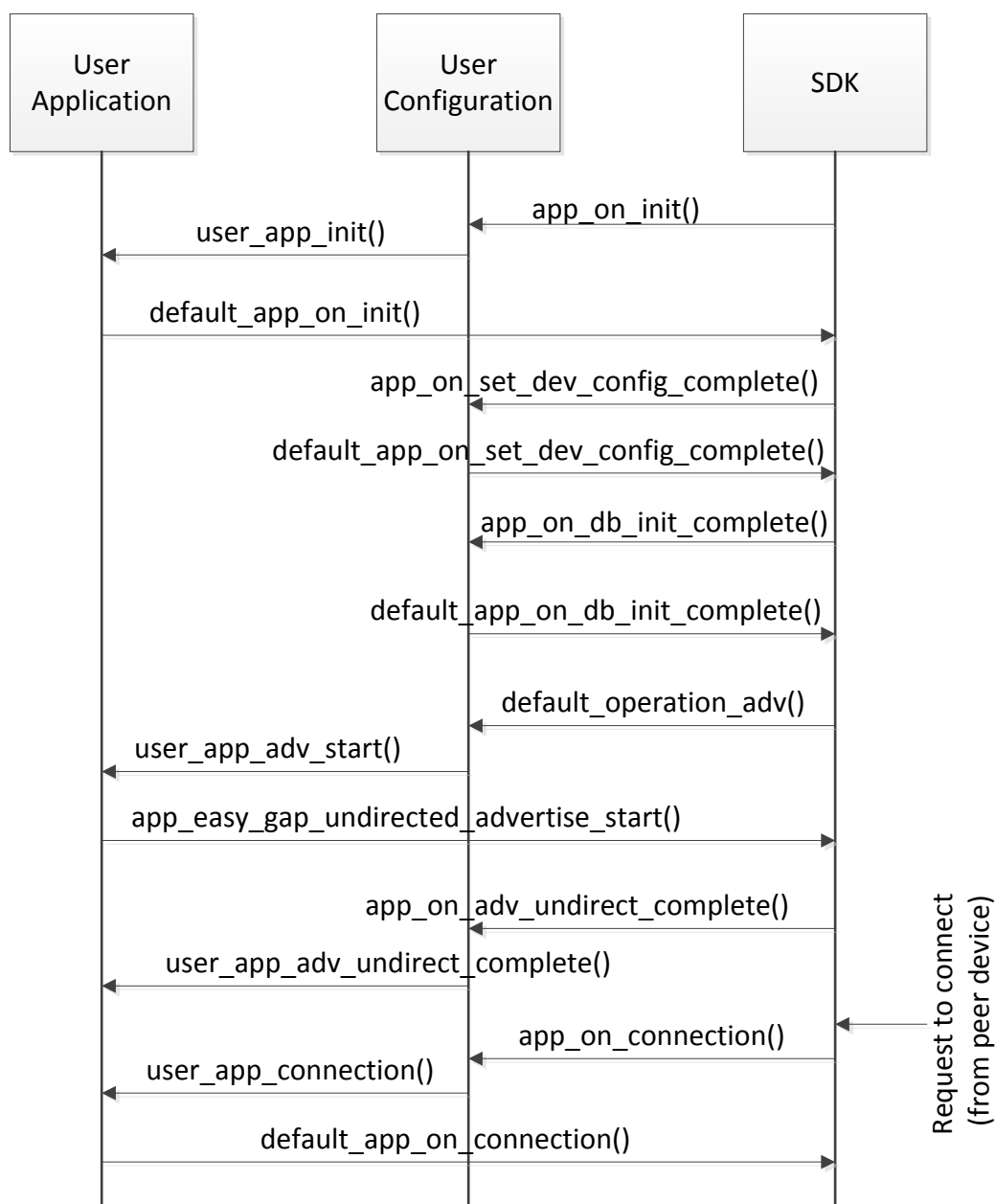
Figure 36: Pillar 3 Application - User Application Code Flow

8.35 Building the Project for Different Targets and Development Kits

The Pillar 3 application can be built for three different target processors: DA14580, DA14581 and DA14583. The selection is done via the Keil tool as depicted in Figure 37.
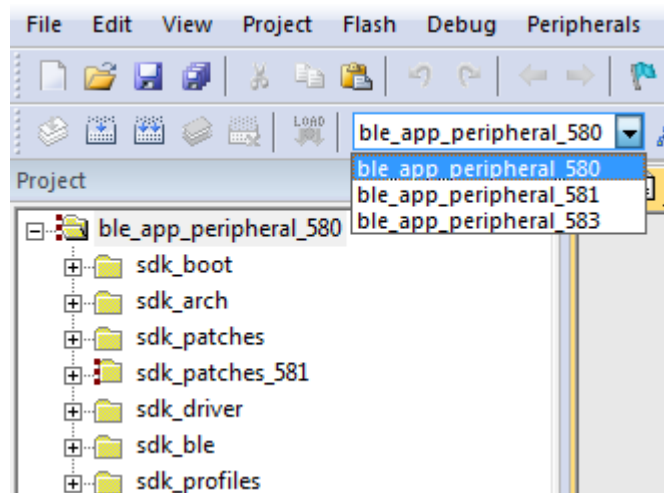
Figure 37: Building the Project for Different Targets

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 38.
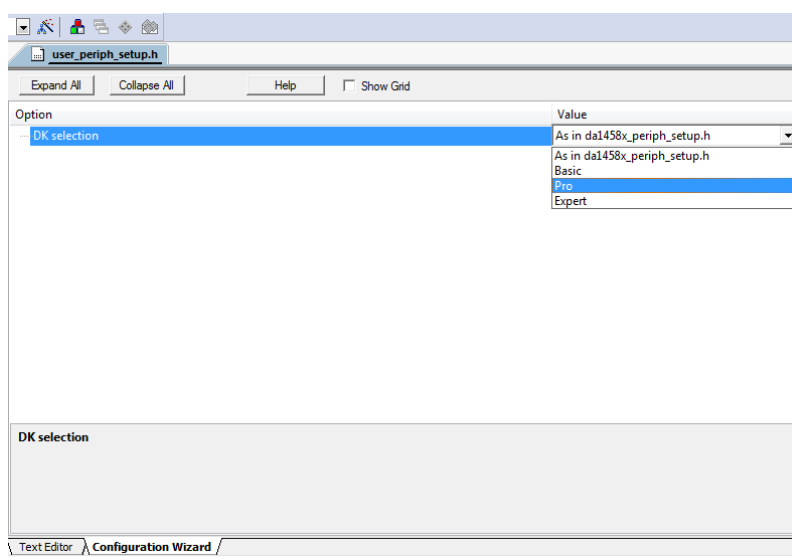
Figure 38: Development Kit Selection for Pillar 3 Application

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

8.36   Interacting with BLE Application
8.37   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 39 shows the result when the iPad/iPod/iPhone device manages to connect to the DA14580/581/583 (the application's advertising device name is **DIALOG-PRPH**).
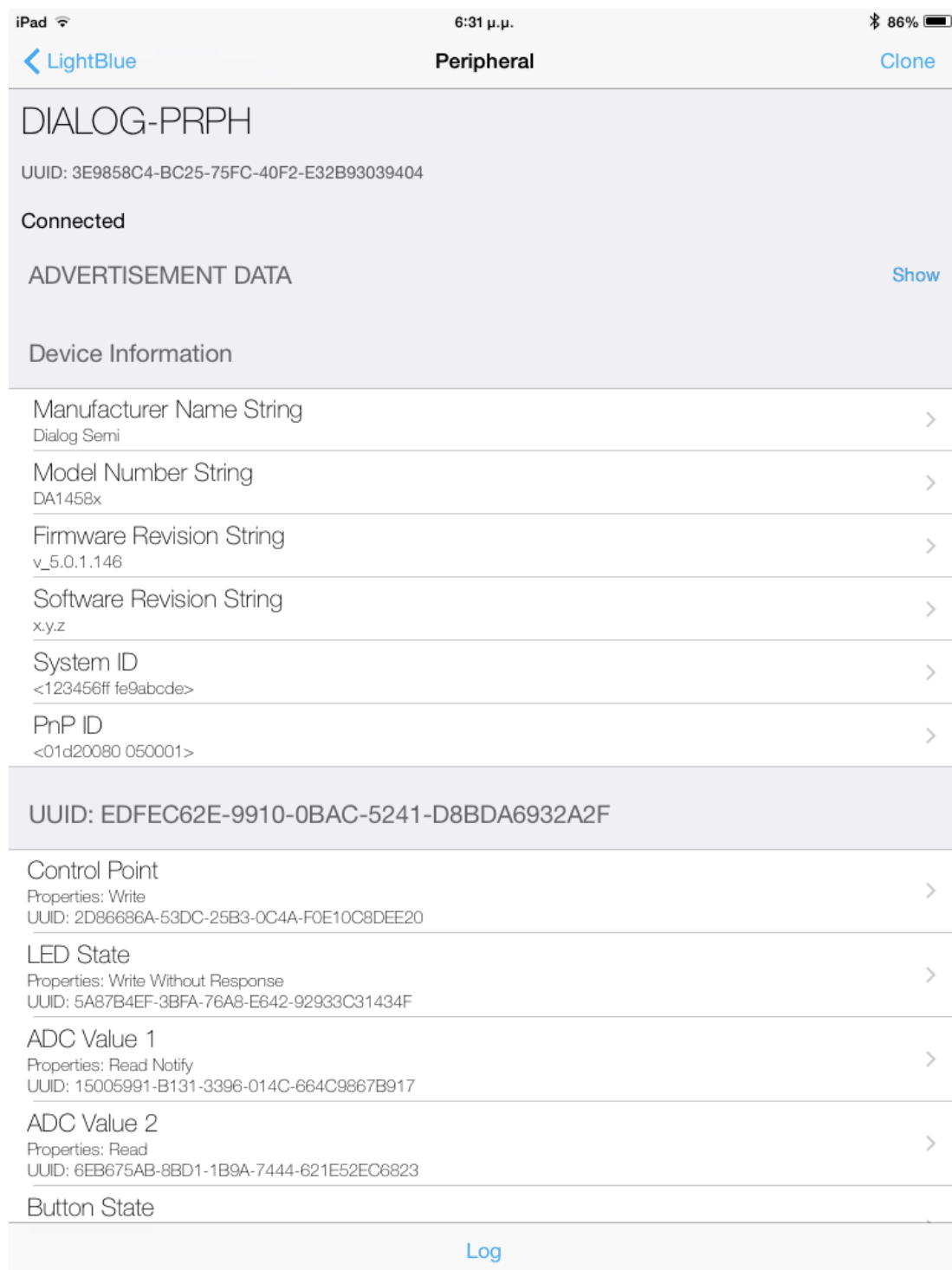
Figure 39: LightBlue Application Connected to Pillar 3 Application

8.38   Pillar 4 (Security)
8.39   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.

8.40   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.

8.41   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.

8.42   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.
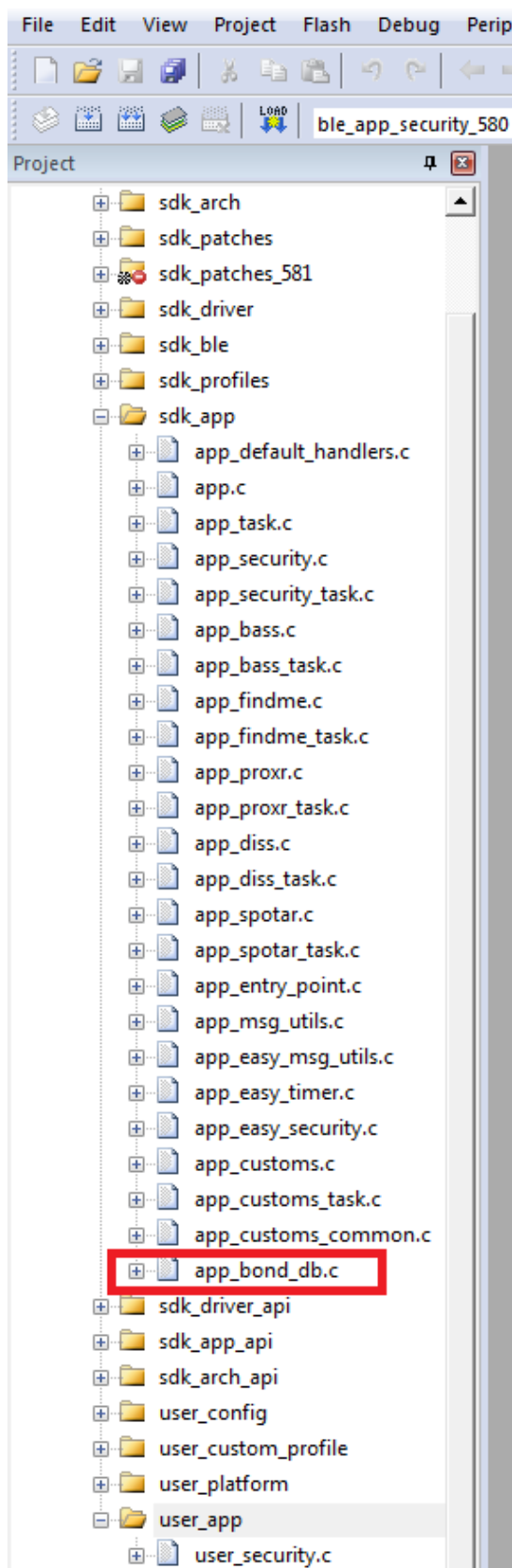
Figure 40: Pillar 4 Keil Project Layout

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

8.43 Going Through the Code
8.44 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 DA14580/581/583 advanced configuration settings.

- `da1458x_config_basic.h`, holds DA14580/581/583 basic 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:
  - `diss.h`, includes the Device Information service.
  - `custs1.h`, 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.

8.45 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,
#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,
#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.

8.46 BLE Application Abstract Code Flow

Figure 41 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.
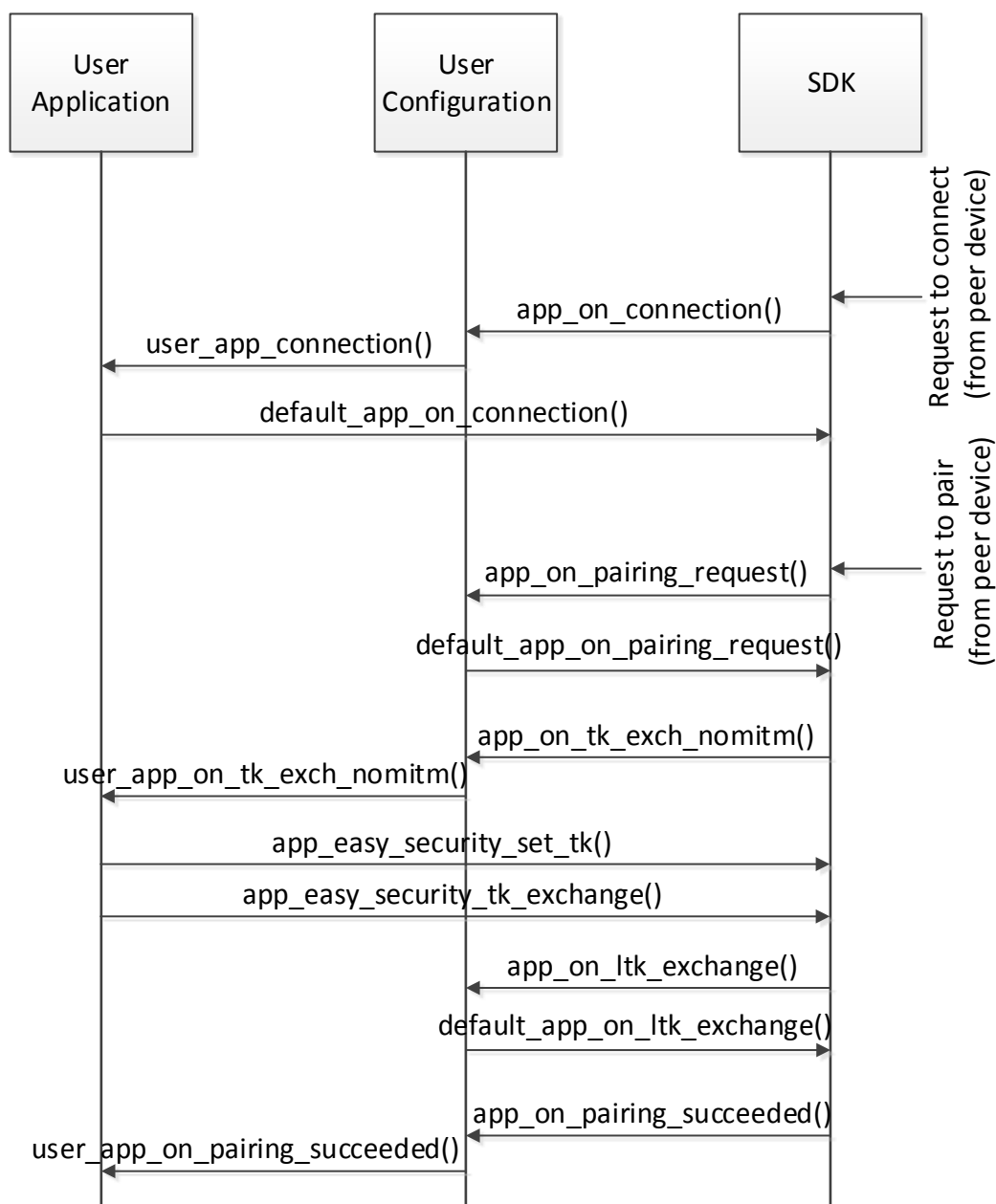
Figure 41: Pillar 4 Application - User Application Code Flow for Pairing using Passkey Entry

`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 42.
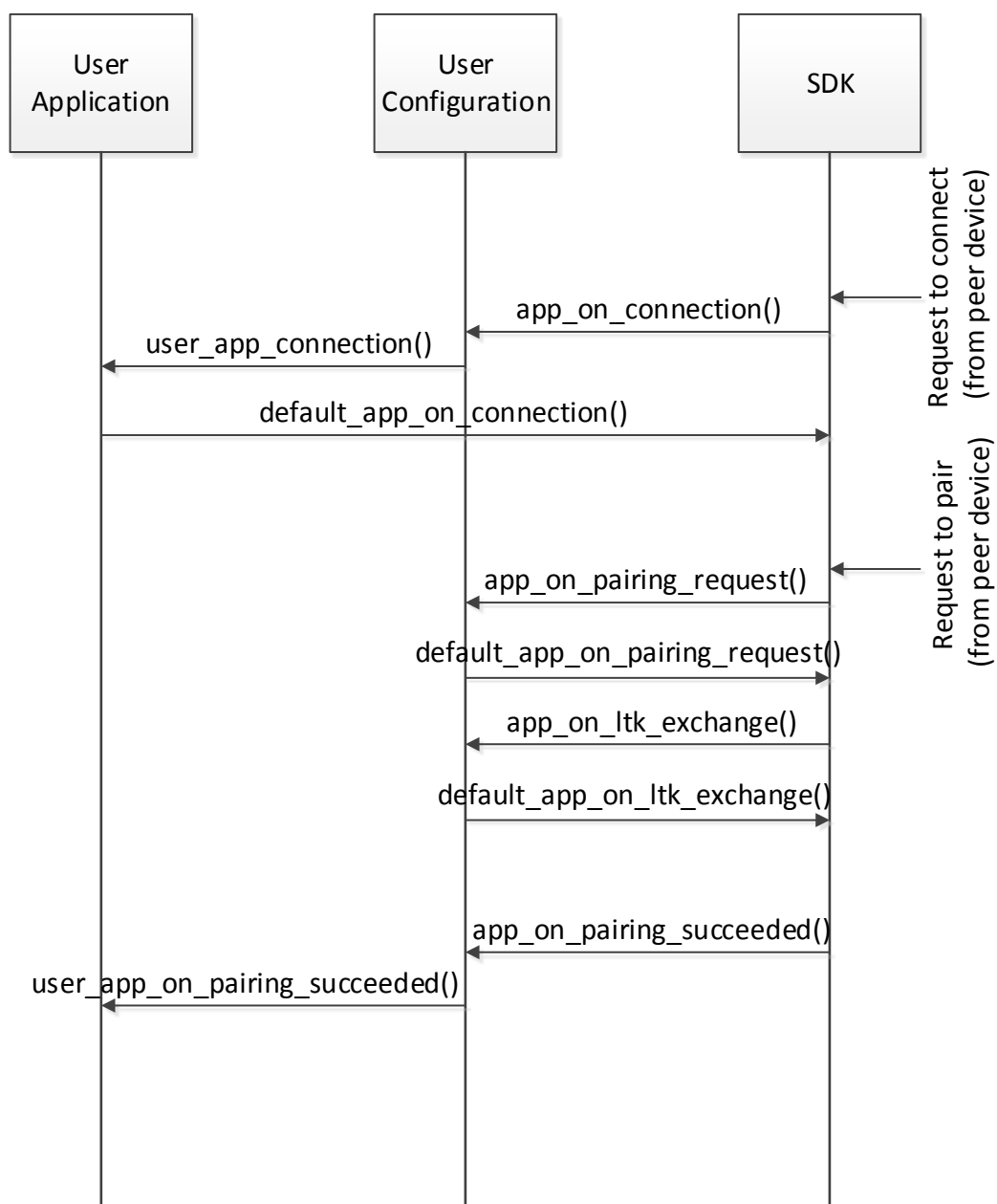
Figure 42: Pillar 4 Application - User Application Code Flow for Pairing using Just Works

8.47 Building the Project for Different Targets and Development Kits

The Pillar 4 application can be built for three different target processors: DA14580, DA14581 and DA14583. The selection is done via the Keil tool as depicted in Figure 43.
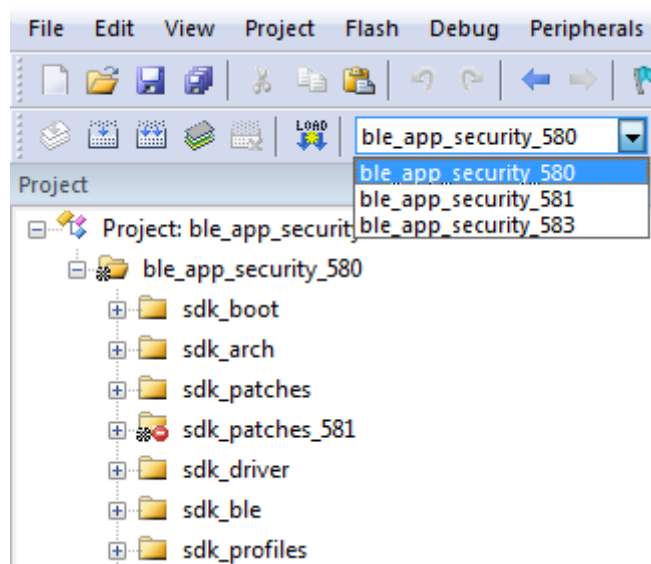
Figure 43: Building the Project for Different Targets

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 44.
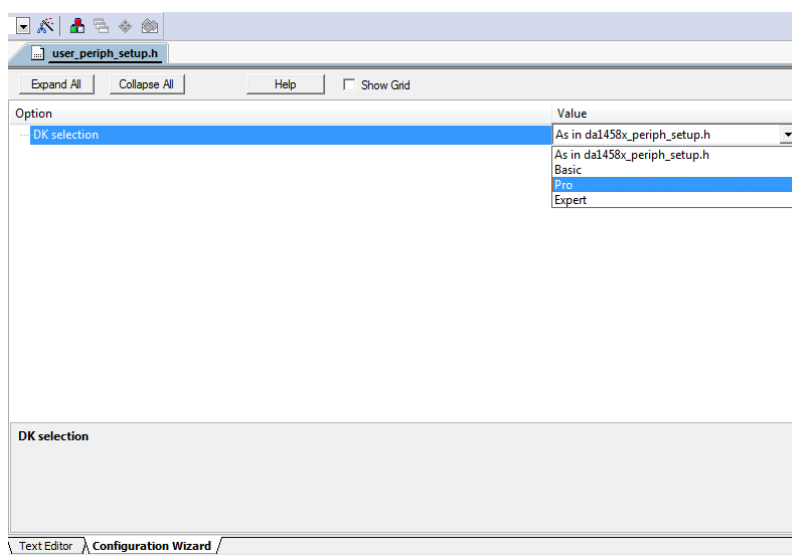
Figure 44: Development Kit Selection for Pillar 4 Application

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

8.48   Interacting with BLE Application
8.49   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 45 shows the result when the iPad/iPod/iPhone device manages to connect to the DA14580/581/583 (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 passkey value.
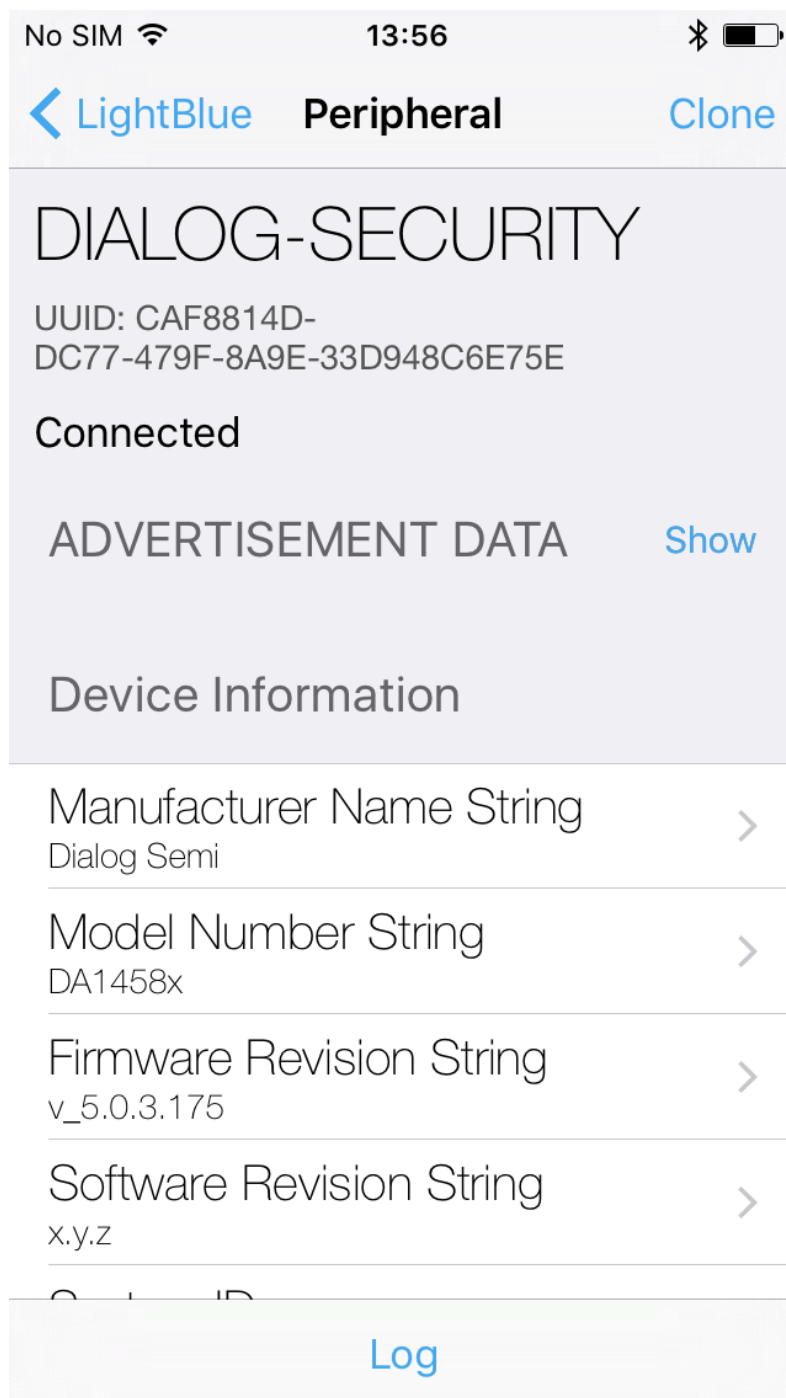
Figure 45: LightBlue Application Connected to Pillar 4 Application
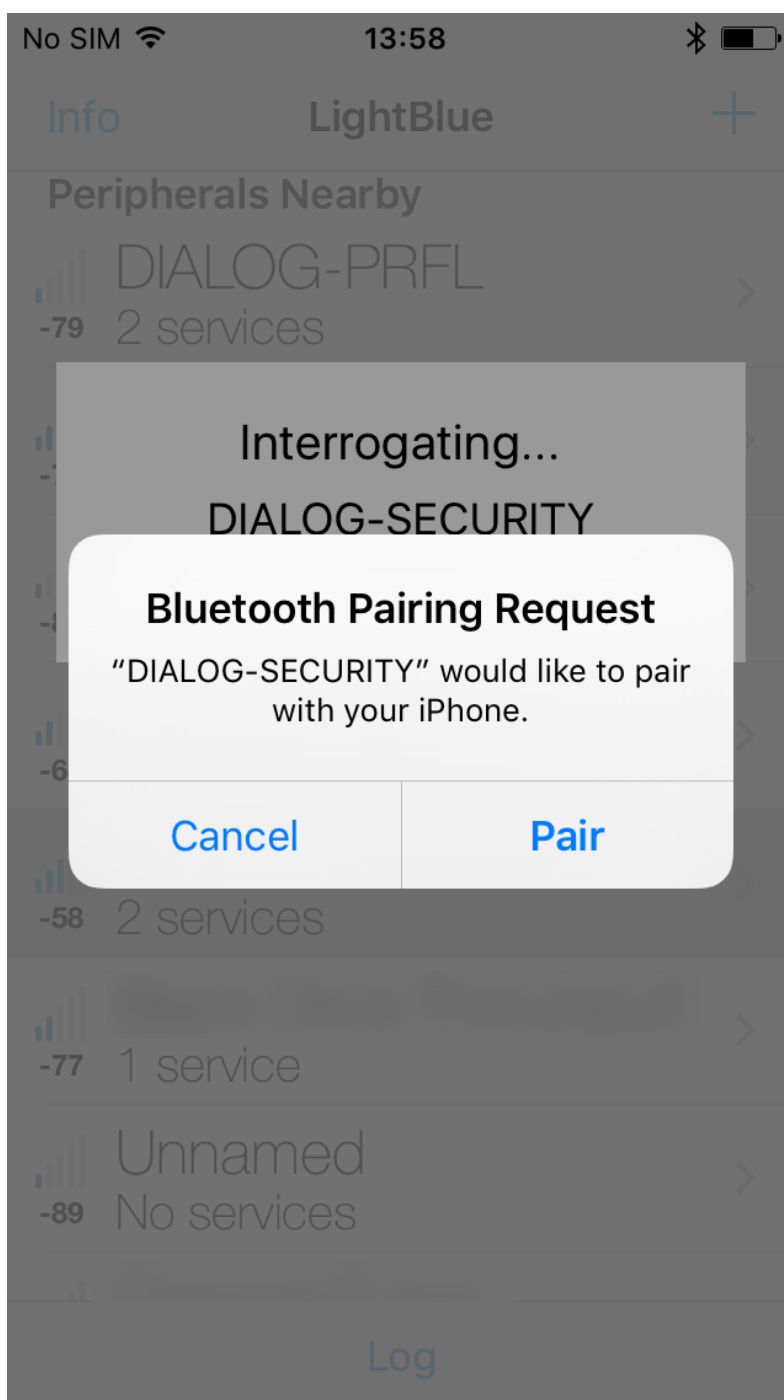
Figure 46: LightBlue Application Pairing with Pillar 4 Application using Just Works
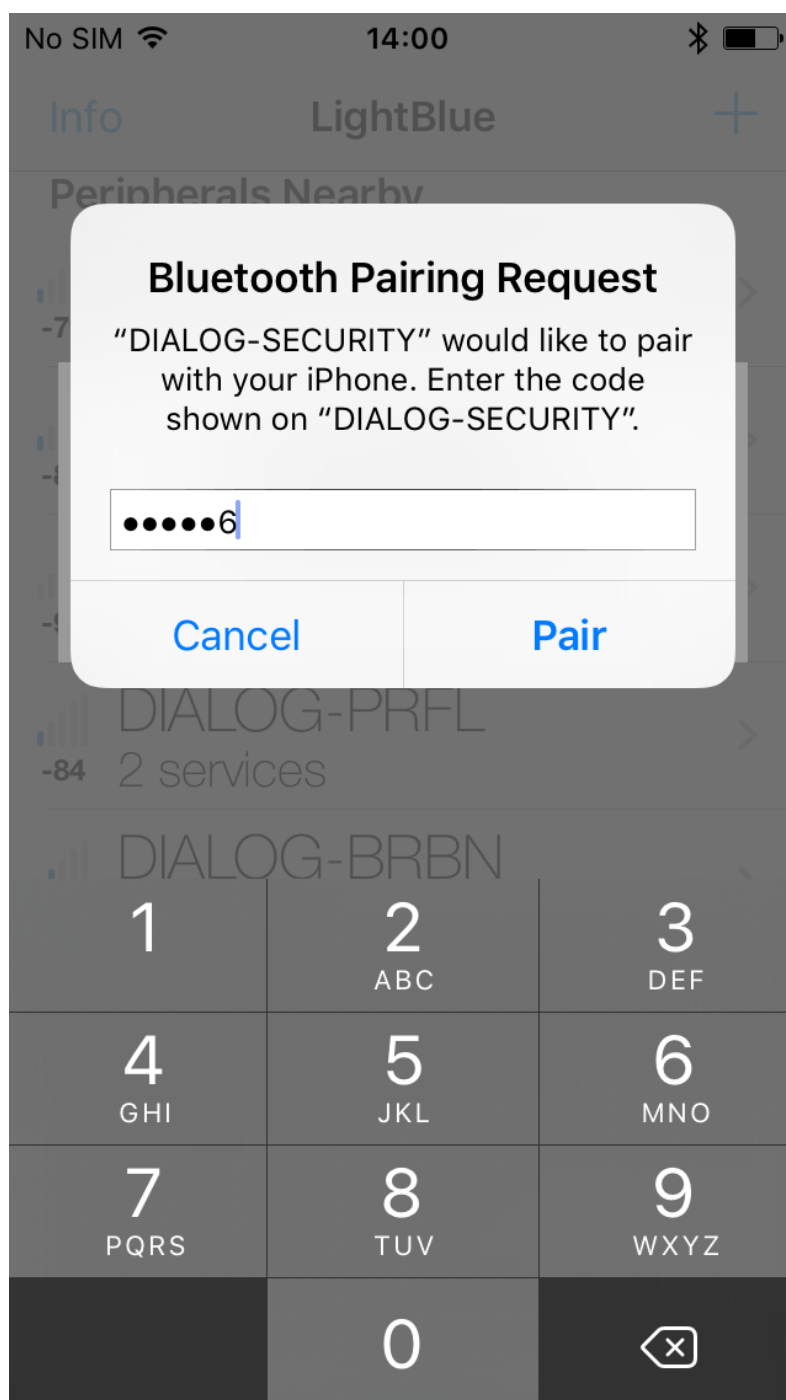
Figure 47: LightBlue Application Pairing with Pillar 4 Application Using Passkey with MITM

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

8.50   Pillar 5 (Sleep Mode)
8.51   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 available sleep modes are:

● Extended sleep mode

● Deep sleep mode

The application uses the "Integrated processor" configuration.

8.52   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 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 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`.

8.53 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 deep sleep.

● 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.

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

● 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.

8.54 Loading the Project

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

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

Figure 48 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.
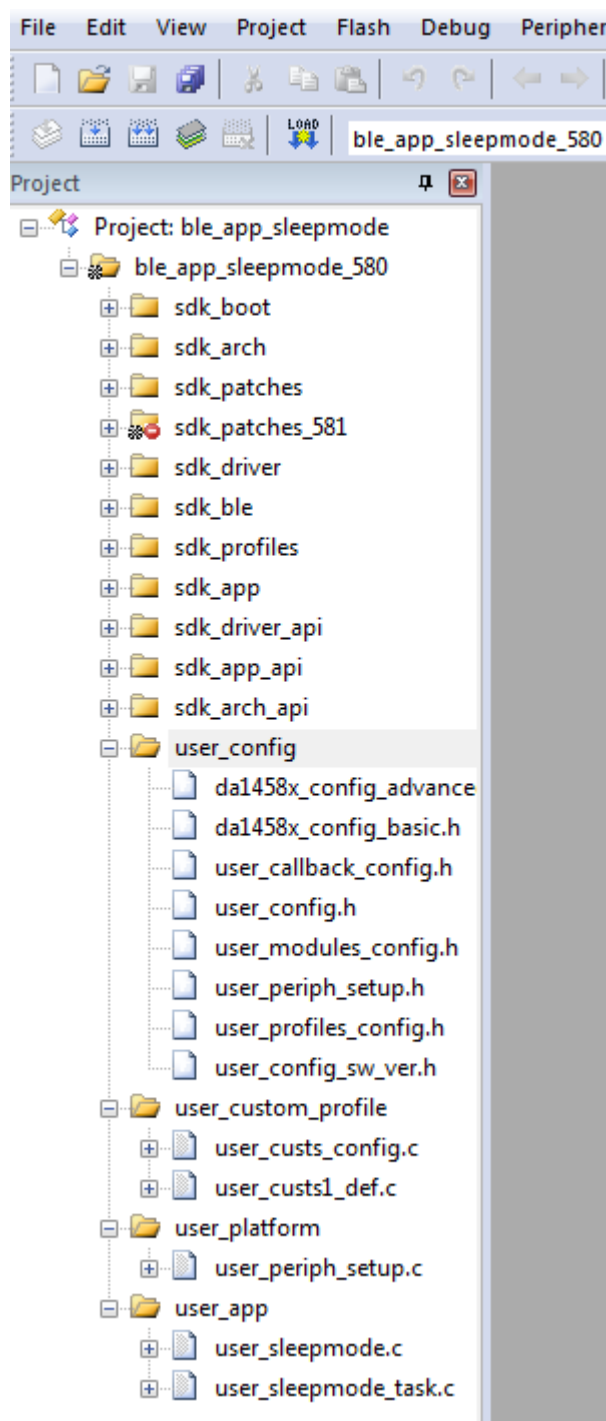
Figure 48: Pillar 5 Keil Project Layout

8.55 Going Through the Code
8.56 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 DA14580/581/583 advanced configuration settings.
- `da1458x_config_basic.h`, holds DA14580/581/583 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:
  - `diss.h`, includes the Device Information service.
  - `custs1.h`, 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.

8.57 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,
#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_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,
};
```

8.58 BLE Application Abstract Code Flow

Figure 49 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.
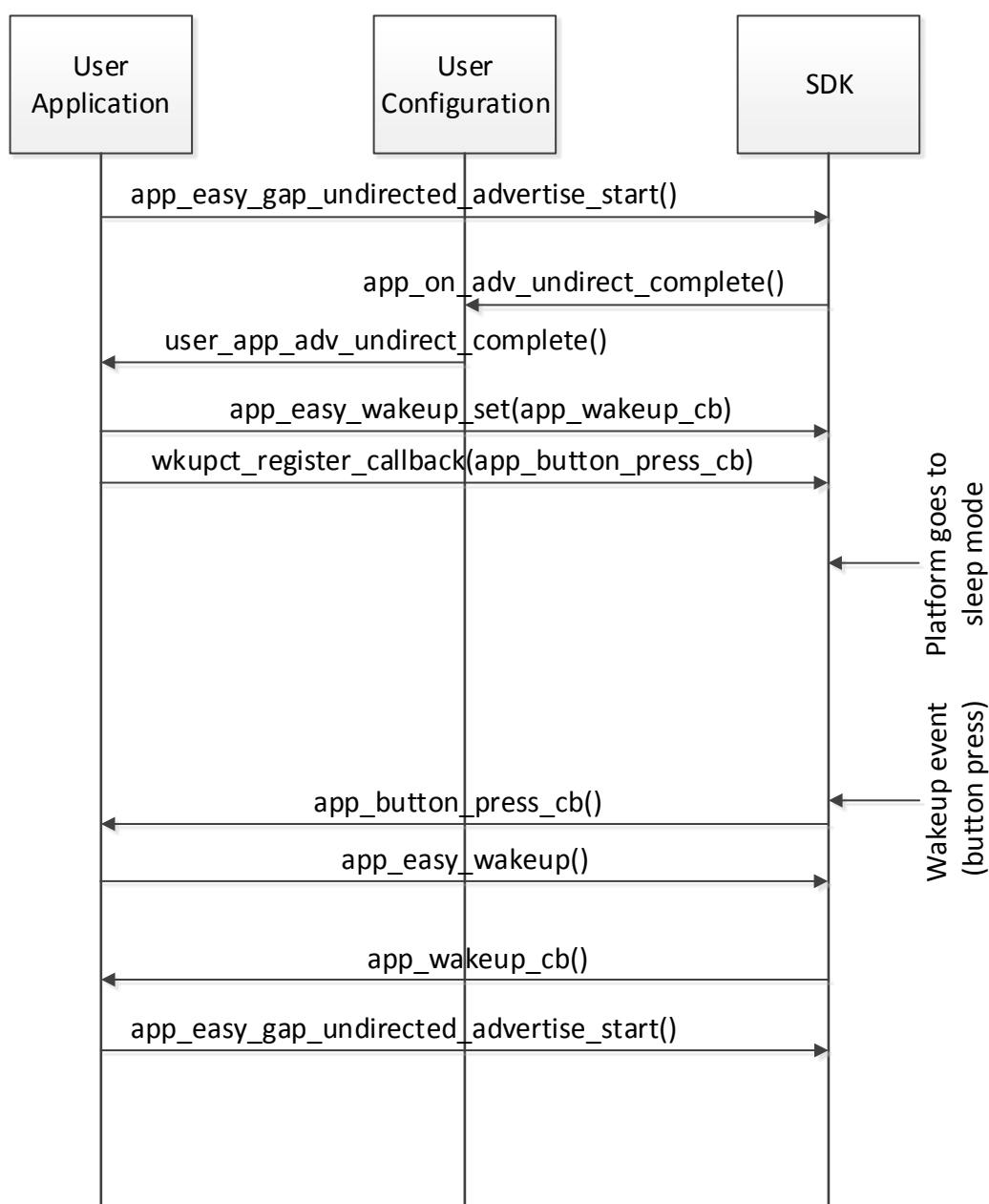
Figure 49: Pillar 5 Application - User Application Code Flow

8.59   Building the Project for Different Targets and Development Kits

The Pillar 5 application can be built for three different target processors: DA14580, DA14581 and DA14583. The selection is done via the Keil tool as depicted in Figure 50.
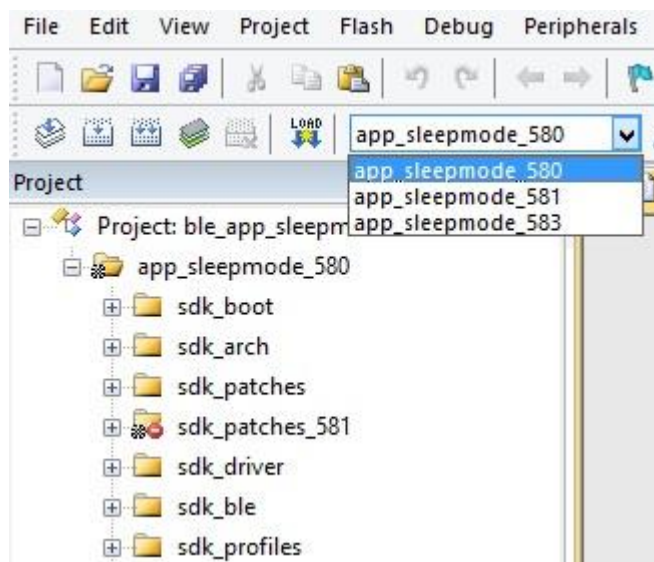
Figure 50: Building the Project for Different Targets

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 51.
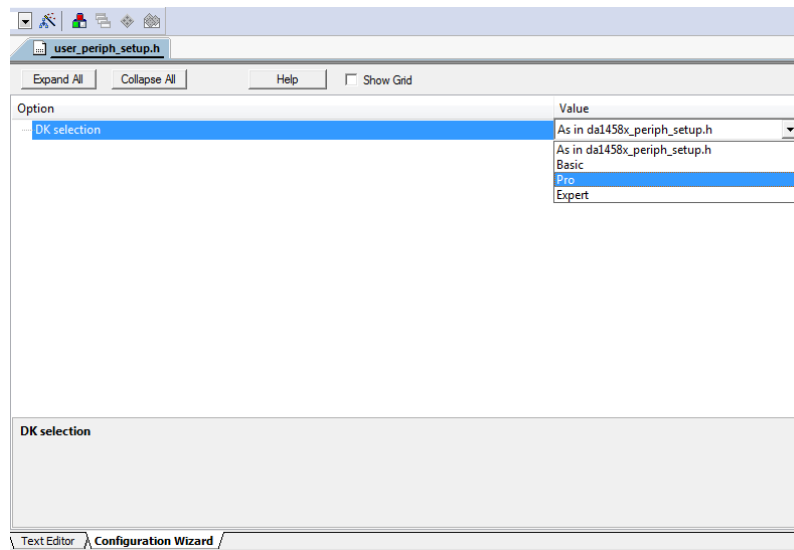
Figure 51: Development Kit Selection for Pillar 5 Application

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

8.60 Interacting with BLE Application
8.61 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 52 shows the result when the iPad/iPod/iPhone device manages to connect to the DA14580/581/583 (the application's advertising device name is **DIALOG-SLEEP**).
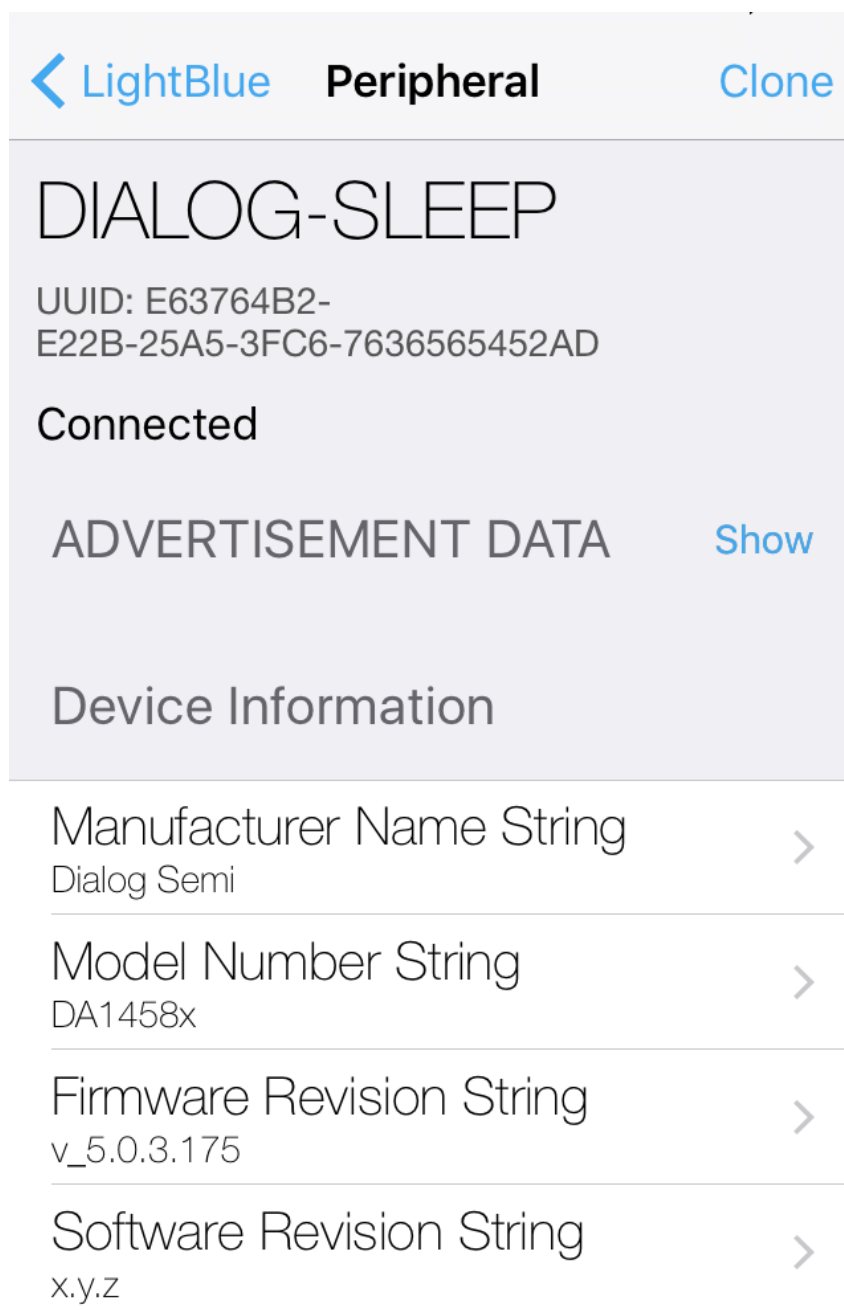
Figure 52: LightBlue Application Connected to Pillar 5 Application

8.62 Pillar 6 (OTA)
8.63 Application Description

The Pillar 6 (OTA) BLE example application demonstrates the same as the Pillar 2 application, plus the SPOTAR service (16-bit UUID). The SPOTAR 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.

8.64 Basic Operation

Supported services:

- Inherits the services from Pillar 2 application.
- SPOTAR 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 SPOTAR Service Characteristic Values and Properties

Name	Properties	Length (B)	Description/Purpose
SPOTA 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	Information about the already applied patches and the entire patch area for SPOTA. In case of SUOTA: Number of bytes transferred.
SPOTA PATCH LENGTH	READ, WRITE	2	Length of a new SPOTA patch or block length of SUOTA image to be sent at a time.
SPOTA PATCH DATA	READ, WRITE, WRITE NO RESPONSE	20	20 bytes of SPOTA data, word aligned. MS byte first.
SPOTA SERVICE STATUS	READ, NOTIFY	1	SPOTA Service Status.

More detailed description of the SPOTAR service internals can be found in Ref. [7].

8.65 User Interface

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

● Download patches or the whole firmware image over a Bluetooth Low Energy link. The detailed procedure description as well as image preparation process can be found in Ref. [7].

8.66 Loading the Project

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

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

Figure 53 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.
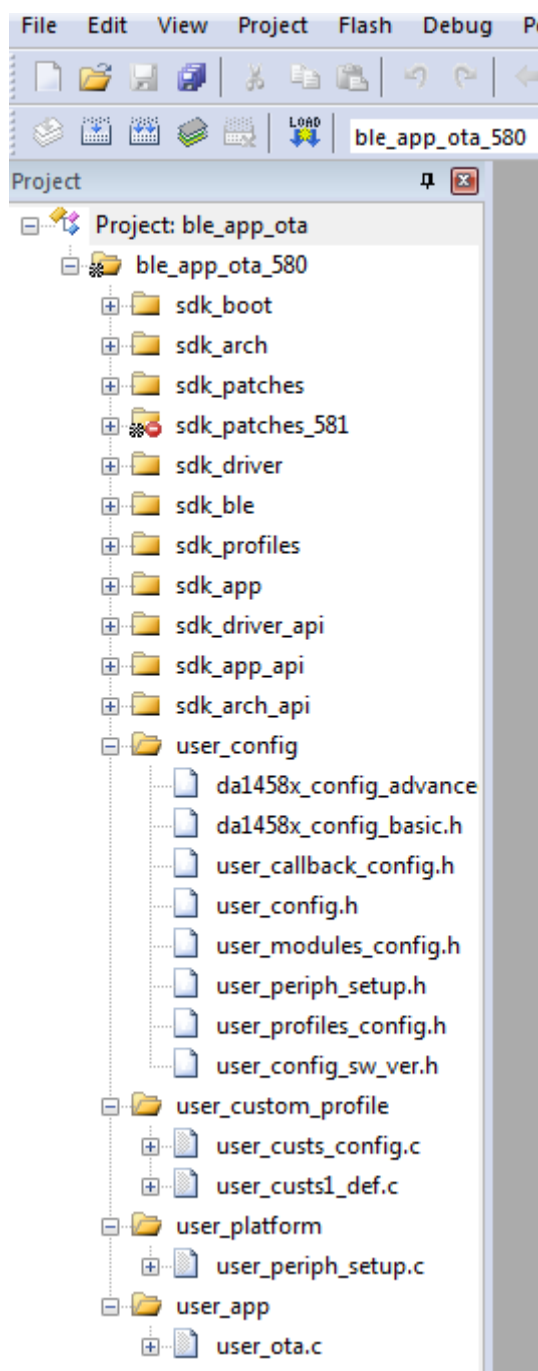
Figure 53: Pillar 6 Keil Project Layout

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

8.67 Going Through the Code
8.68 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 DA14580/581/583 advanced configuration settings.
- `da1458x_config_basic.h`, holds DA14580/581/583 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:
  - `diss.h`, includes the Device Information service.
  - `spotar.h`, includes the SPOTAR service.
  - `custs1.h`, includes the Custom 1 service.
- `It also exposes some configuration flags for the SPOTAR service:`
  - #define SPOTAR_PATCH_AREA (1), Place where the SPOTAR service is placed. 0 for RetRAM and 1 for SYSRAM.
  - #define CFG_SPOTAR_I2C_DISABLE , Disable I2C external memory module
  - #define CFG_SPOTAR_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.

8.69 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,
#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,
#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.

8.70 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 31 for the Pillar2 application. The whole OTA functionality requires no custom user application code to be written.

8.71 Building the Project for Different Targets and Development Kits

The Pillar 6 application can be built for three different target processors: DA14580, DA14581 and DA14583. The selection is done via the Keil tool as depicted in Figure 54.
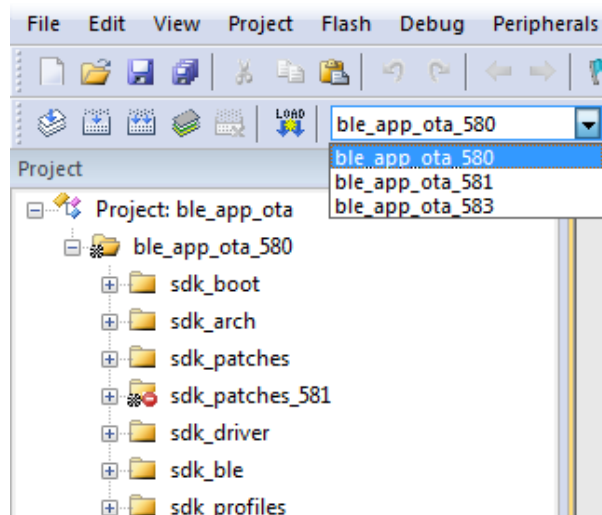
Figure 54: Building the Project for Different Targets

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 55.
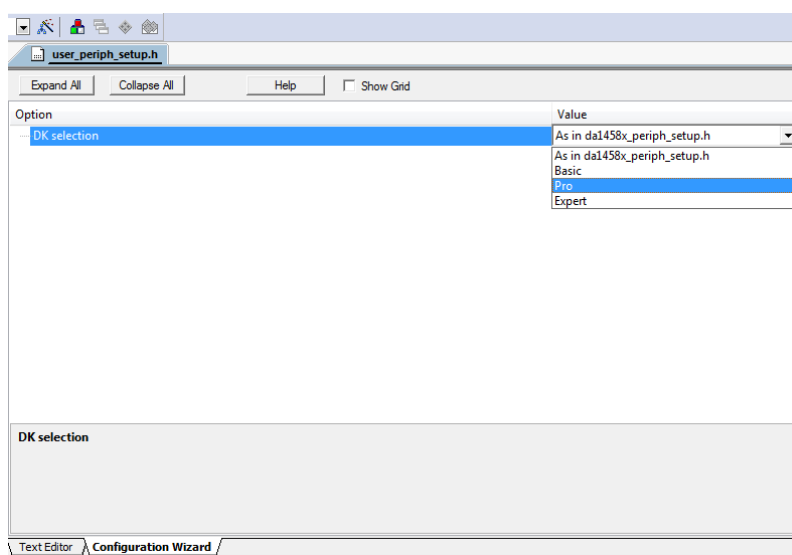
Figure 55: Development Kit Selection for Pillar 6 Application

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

8.72   Interacting with BLE Application
8.73   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 DA14580/581/583 (the application's advertising device name is **DIALOG-OTA**).
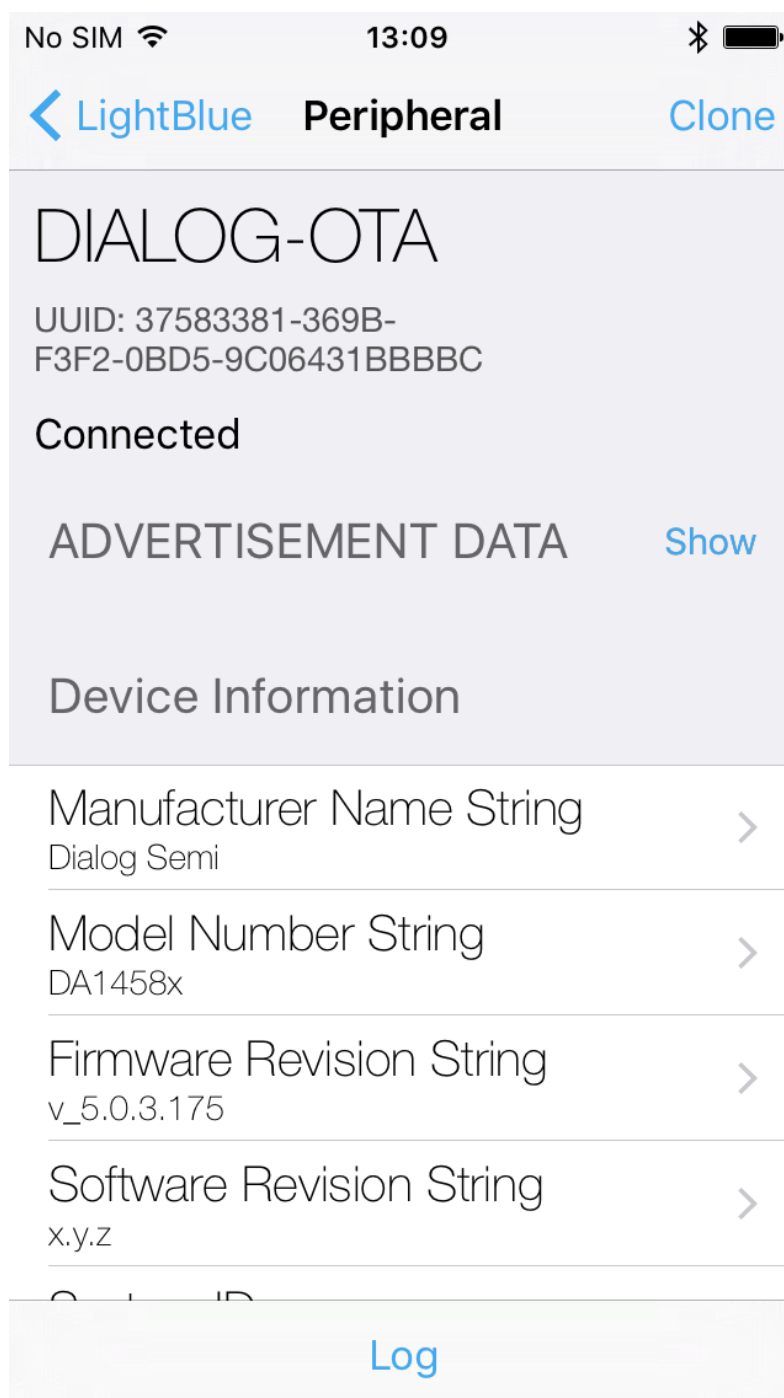
Figure 56: LightBlue Application Connected to Pillar 6 Application

8.74   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 SPOTAR service are shown to the user. The detailed SUOTA update procedure is described in Ref. [7].
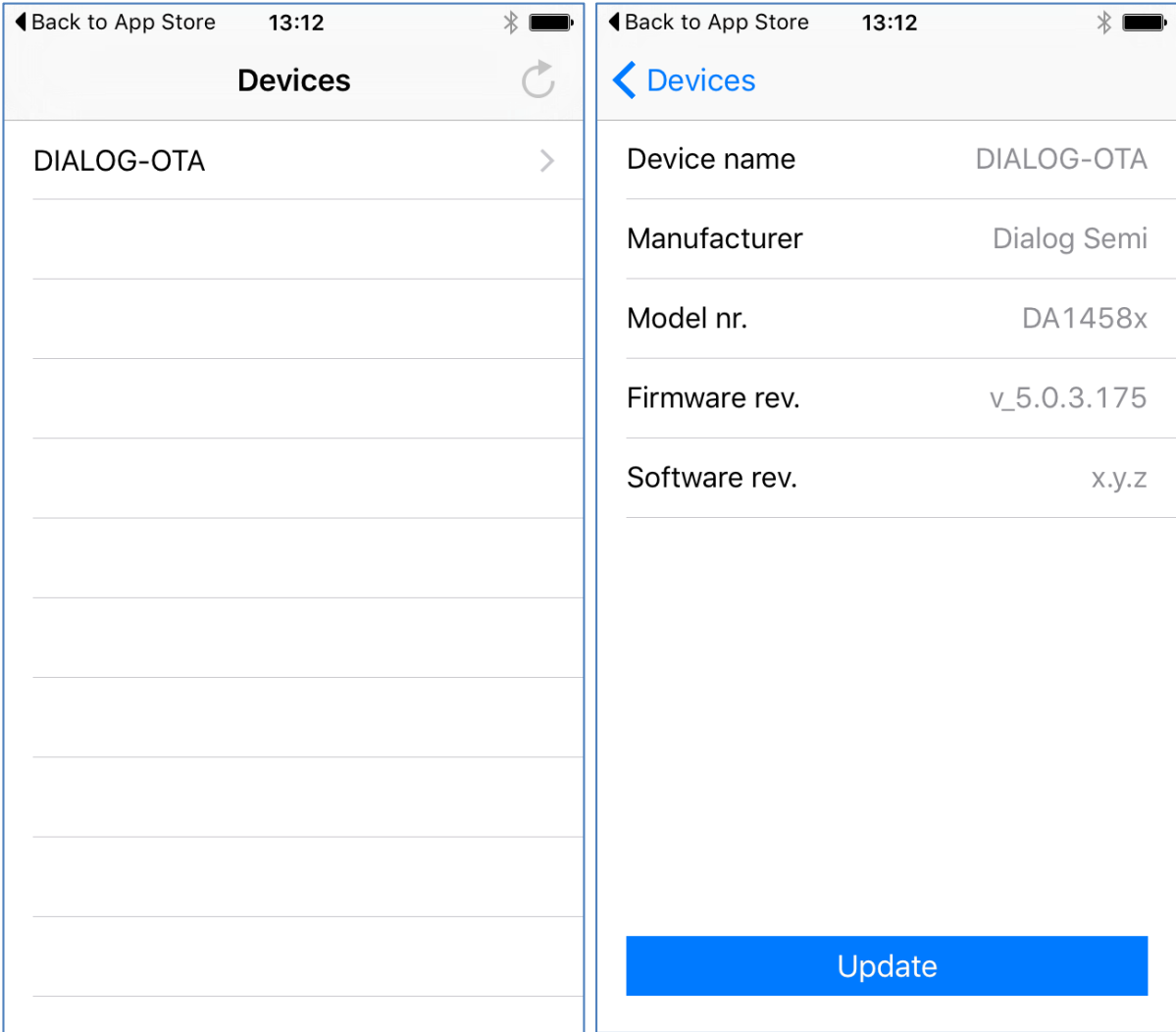
Figure 57: Dialog SUOTA Application Discovering Pillar 6 Application

8.75 Pillar 7 (All in One)
8.76 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.

8.77 Basic Operation

Supported services:

- Inherits the services from Pillar 2 application.
- Inherits the SPOTAR 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 SPOTAR Service Characteristic Values and Properties

Name	Properties	Length (B)	Description/Purpose
SPOTA 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	Information about the already applied patches and the entire patch area for SPOTA. In case of SUOTA: Number of bytes transferred.
SPOTA PATCH LENGTH	READ, WRITE	2	Length of a new SPOTA patch or block length of SUOTA image to be sent at a time.
SPOTA PATCH DATA	READ, WRITE, WRITE NO RESPONSE	20	20 bytes of SPOTA data, word aligned. MS byte first.
SPOTA SERVICE STATUS	READ, NOTIFY	1	SPOTA Service Status.

Pillar 7 provides the same SUOTA functionality as Pillar 6.

8.78 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 and patching as in the Pillar 6 application.

8.79 Loading the Project

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

`projects\target_apps\ble_examples\ble_app_all_in_one\Keil_5\ble_app_ota.uvprojx`

Figure 58 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.
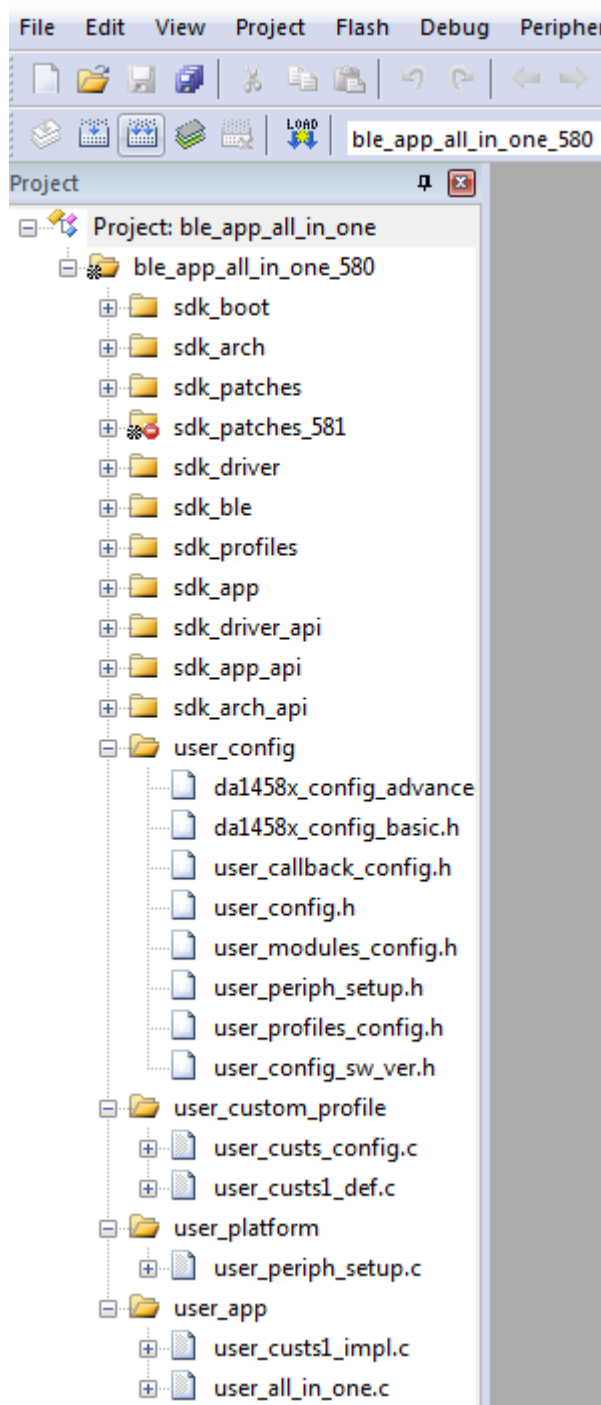
Figure 58: Pillar 7 Keil Project Layout

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

8.80 Going Through the Code
8.81 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 DA14580/581/583 advanced configuration settings.

- `da1458x_config_basic.h`, holds DA14580/581/583 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:
  - `diss.h`, includes the Device Information service.
  - `spotar.h`, includes the SPOTAR service.
  - `custs1.h`, includes the Custom 1 service.

- It also exposes some configuration flags for the SPOTAR service:
  - #define SPOTAR_PATCH_AREA   (1), Place where the SPOTAR service is placed. 0 for RetRAM and 1 for SYSRAM.
  - #define CFG_SPOTAR_I2C_DISABLE   , Disable I2C external memory module
  - #define CFG_SPOTAR_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.

8.82   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,
#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,
#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.

8.83 BLE Application Abstract Code Flow

Figure 59 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.
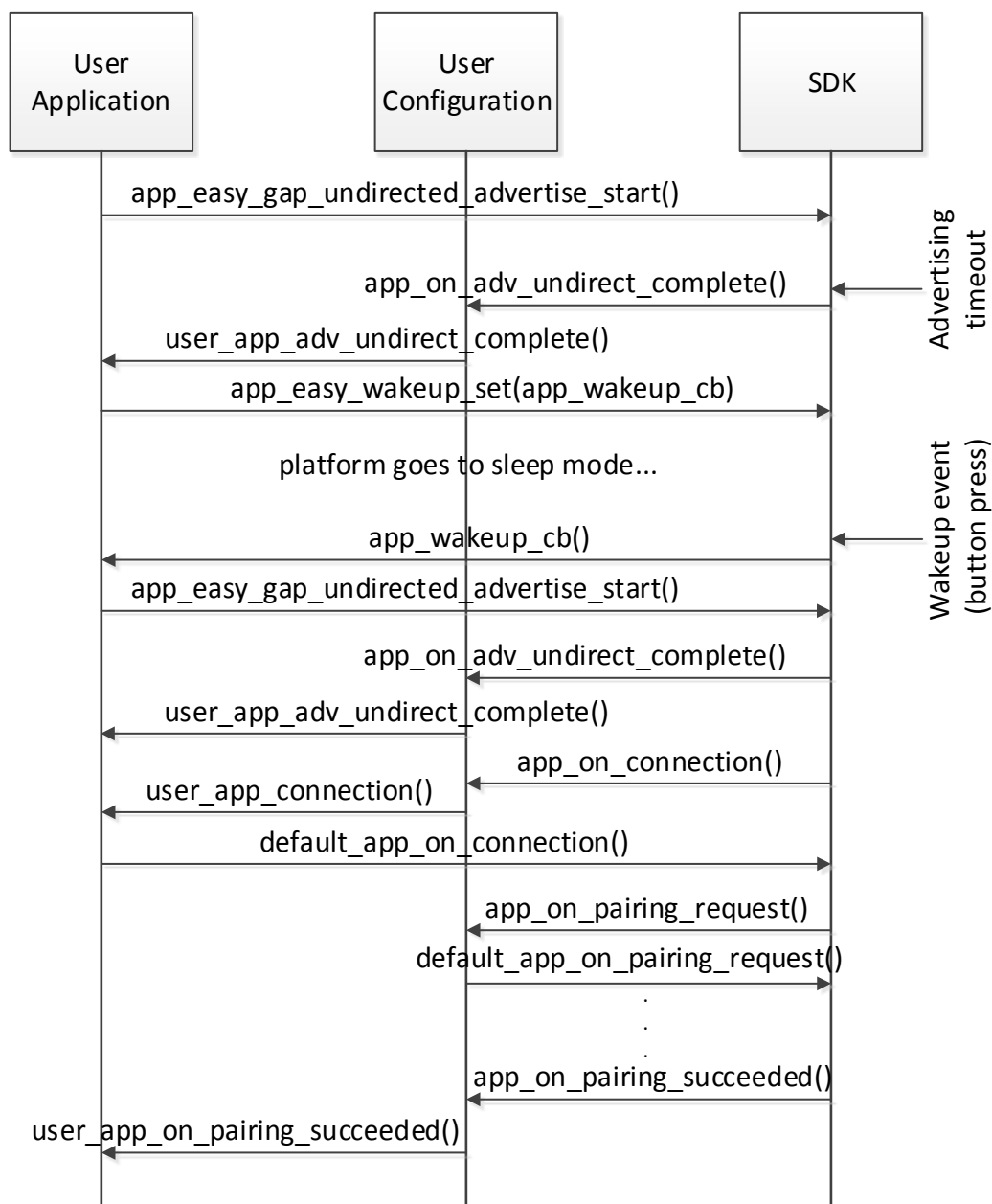
Figure 59: Pillar 7 Application - User Application Simplified Code Flow

8.84 Building the Project for Different Targets and Development Kits

The Pillar 7 application can be built for three different target processors: DA14580, DA14581 and DA14583. The selection is done via the Keil tool as depicted in Figure 60.
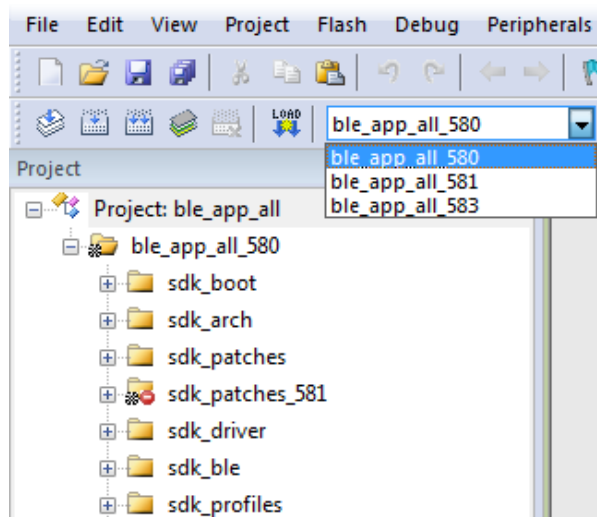
Figure 60: Building the project for different targets

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 61.
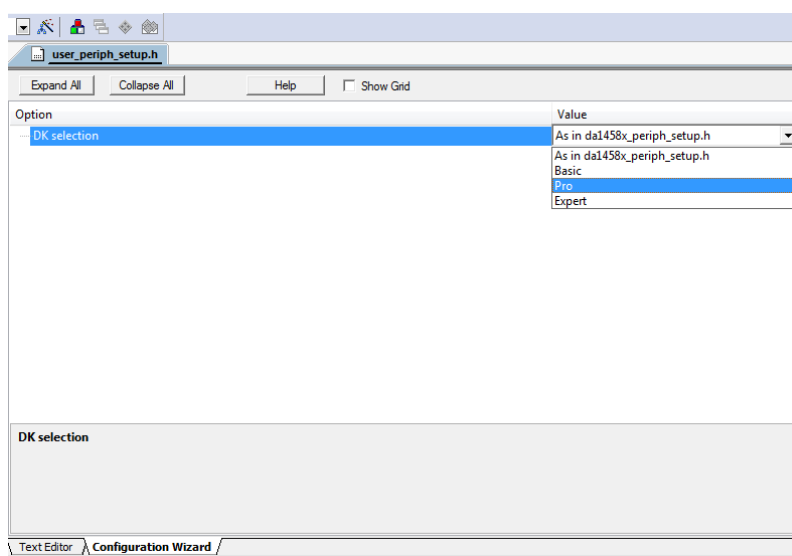
Figure 61: Development Kit Selection for the Pillar 7 Application

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

8.85 Interacting with BLE Application
8.86 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 62 shows the result when the iPad/iPod/iPhone device manages to connect to the DA14580/581/583 (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.
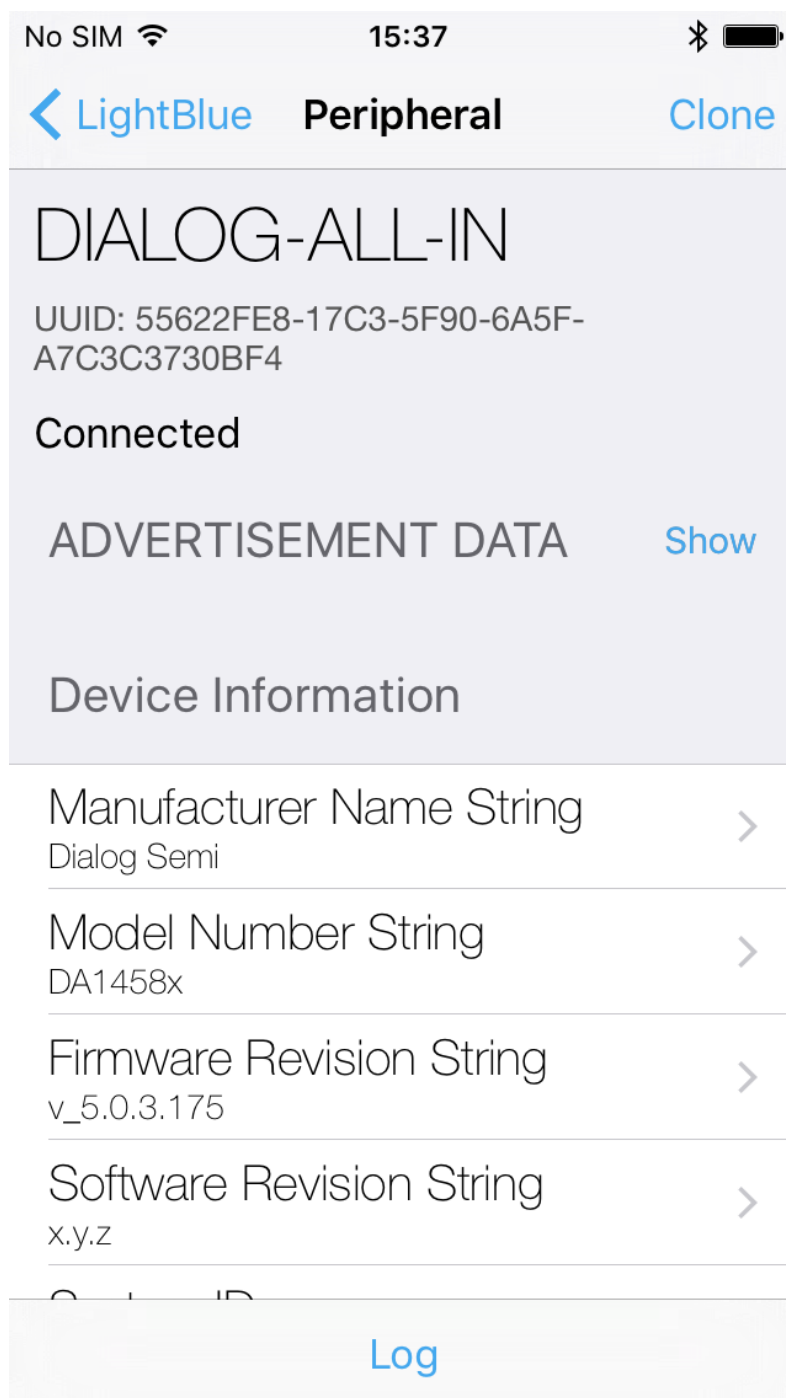
Figure 62: LightBlue Application Connected to Pillar 7 Application

9 Creating Your BLE Application

This section describes the steps needed for creating a new application project starting from the application template provided in the DA1458x SDK.

9.1 Using the Empty Project Template

1. Open Windows Explorer, locate the DA1458x SDK distribution and open the folder:
   `projects\target_apps\template`

2. If the new application is for a peripheral device in Integrated Processor configuration then make a clone of the `empty_peripheral_template`. For a device in External Processor configuration, make a clone of the `empty_template_ext`. In this example the `empty_peripheral_template` will be used in order to walk thought the process. The steps for the `empty_template_ext` case are quite similar.

3. Rename the newly created folder to e.g. `my_application`.
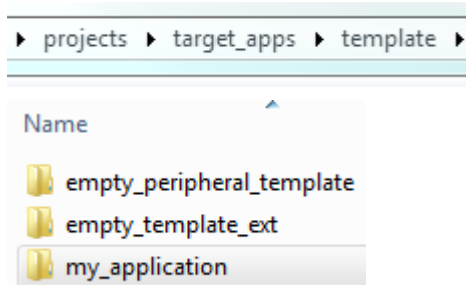
   Note: Make sure the newly created directory has the same depth as the original one. This is required since the Keil projects have inclusion path dependencies.

4. Open folder `my_application` and depending on the Keil version installed (in this example Keil_5 is used) open the respective Keil folder and rename the project files to:
   `my_application.uvprojx`
   `my_application.uvoptx`

5. Open folder `projects\target_apps\template\my_application\src` and rename the files:
   `user_empty_peripheral_template.c` to e.g. `user_my_application.c`
   `user_empty_peripheral_template.h` to e.g. `user_my_application.h`

6. Open file `user_my_application.c`, change the file name in the file header comments and change the include header `user_empty_peripheral_template.h` to `user_my_application.h` as shown below:

   ```
   /*
    * INCLUDE FILES
    ******************************
    */

   #include "app_api.h"
   #include "user_my_application.h"
   ```

7. Similarly, open file
   `projects\target_apps\template\my_application\src\config\user_callback_config.h` and change the include header file from `user_empty_peripheral_template.h` to `user_my_application.h`.

8. Open file `user_my_application.h`, change the file name in the file header comments and edit the preprocessor directives as follows:

```
#ifndef USER_MY_APPLICATION_H_
#define USER_MY_APPLICATION_H_
```

Edit the comment at the end of the file to match the changes:

```
#endif //USER_MY_APPLICATION_H_
```

9. At this point the Keil project can be opened for the final changes before we build the new application. Open folder `projects\target_apps\template\my_application\Keil_5` and double click on the project file `my_application.uvprojx`. This will open the new project on the Keil development environment.

10. In the left window the project file groups are listed. Expand the `user_app` group. Note that it contains the old `user_empty_peripheral_template.c` file. Click on this file to highlight and press the delete key on your keyboard to remove the file from the group. Click on the `user_app` group to highlight, right-click on your mouse and select the menu option "Add Files to Group 'user_app'…". Browse to the new file `projects\target_apps\template\my_application\src\user_my_application.c`, click on the file and press the "Add" button and close the window. Check that the `user_app` group now contains the correct file `user_my_application.c`.

11. Click on the "File extensions" button to change the project target names. Double click on the `empty_peripheral_template_580` and change to `my_application_580`. Repeat for 581 and 583. as shown below and close the window.
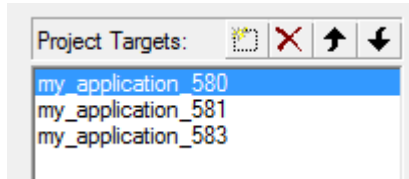
12. Click on the "Target Options" button to change the executable name from `empty_peripheral_template_580 to my_application_proj_580` as shown below (this step needs to be repeated for the other targets: my_application_581 and my_application_583).
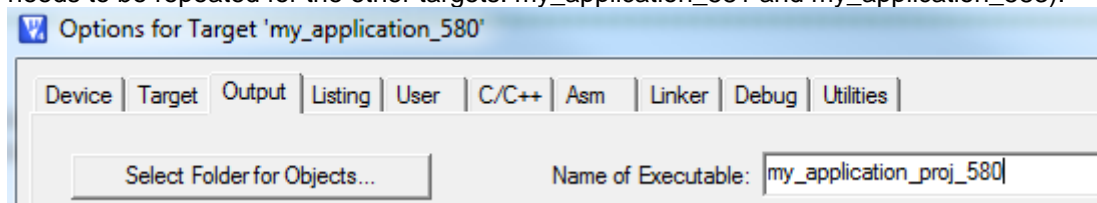
13. At this point the new project is ready and it can be built. Open the `user_periph_setup.h` file. Click on the "Configuration Wizard" tab at the bottom of the window. Chose the right board configuration from the drop down menu, as shown below, and press "F7" to build.
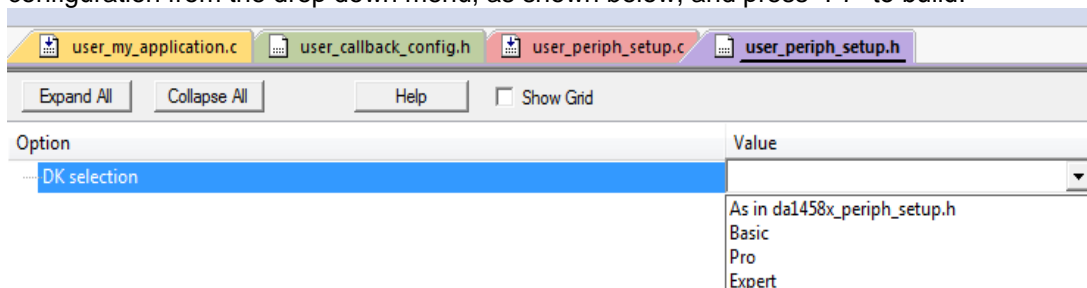
14. To save the changes in the new project close the Keil program or close the project.

9.2 Configuring Your Application

The user configuration files and a short description for each file are shown in Table 21.

The user configuration files can be found under the Keil project file group `user_config`.

Table 21: User Configuration Files

File Name	Description
da1458x_config_basic.h	The basic configuration options are included in this file, such as: ● Integrated or external processor configuration ● BLE security functionality ● Enable/Disable Watchdog ● Sleep mode memory map configuration ● Maximum concurrent connections supported by application ● Enable/Disable development and debug mode ● UART Console Print
da1458x_config_advanced.h	In this file the advanced configuration options are included, such as: ● Low Power clock selection ● Wakeup from external processor ● Scatter file - Memory maps configuration ● NVDS configuration ● Enables True Random number Generator The list of the preprocessor directives in this file is quite extensive, thus the user should read the comments in the file for more details.
user_callback_config.h	Callback functions configuration file. In this file the user can replace the default callbacks with user defined callbacks. The callbacks are grouped in the following structures: user_app_callbacks user_profile_callbacks user_default_app_operations user_app_main_loop_callbacks For example, in the user_app_callbacks structure the user can add new functions for app_on_connection and/or app_on_disconnect.
user_config.h	In this file the user can configure the default behavior of the application for the following: ● Sleep mode ● Security ● Advertise ● Connection parameters update ● GAPM configuration
user_modules_config.h	In this file the user can exclude or not a module in user's application code. If a module is excluded then the user must handle in his own the module messages.
user_periph_setup.h	In this file the user can configure the hardware related settings relative to the Development Kit used. For example: ● I2C EEPROM configuration ● SPI FLASH configuration ● LED and button configuration ● UART GPIO configuration

File Name	Description
user_profiles_config.h	In this file the user can specify which BLE profiles (Bluetooth SIG adopted or custom ones) will be included in user's application. This is done by including the C header files of the respective BLE profile. For example, to add the Device Information Service (DIS) server role, the following line shall be added: #include "diss.h"
user_config_sw_ver.h	Use the preprocessor directives in this file to tag the user code software version. These defines are used by the DIS service
user_custs1_def.h/c	These files define the structure of the Custom profile database and the cust_prf_funcs[] array, which contains the Custom profile API function calls

9.3   Using the API
9.4   GAP API

The Pillar 1 example (section 8.2) is a good introduction to the GAP API. Refer to this example to understand how to configure GAP, how to define user specific callback functions for various GAP events and how to configure advertise and create the advertise message for the new application.

9.5   Profile API

The Pillar 2 example (section 8.14) is a good introduction to the API available for the application to interact with the relevant profile. In the example it is explained how to include a SIG profile to the application and how to create a custom profile. All the supported profiles are included in the template project and the relative files can be found under the project file group "sdk_profiles".

Note: The user can remove the profiles that are not relevant to his application from the group. This will improve compilation time of the project. The final application image size is not affected since the code which is not used will not be linked in.

9.6   Peripheral Interface

The Pillar 3 example (section 8.26) is a good introduction to the API available for the application to handle the events triggered from the profile task and interface with peripherals (e.g. LEDs). Also, section 7 introduces all the supported peripherals using detailed examples for each peripheral. All the supported drivers are included in the template project and the relative files can be found under the project file groups sdk_driver and sdk_driver_api.

Note: The user can remove the drivers that are not relevant to his application from the group. This will improve compilation time of the project. The final application image size is not affected since the code which is not used will not be linked in.

9.7   Sleep Mode API

The DA1458x SDK supports three operating modes listed below in order of lowest power consumption (see Ref. [3] for more information):

- **Deep Sleep.** This mode could be used when the image does not exceed 32KB and can be programmed in to OTP so that the system boots from OTP.

- **Extended Sleep.** This mode can be used when the system does not boot from OTP memory. Instead the image will be loaded to SRAM from a non-volatile memory.

- **Active Mode.** This mode could be used during development so that the IDE debugger can be used. Also, this mode could be used for an application running on a device with no power constrains (device connected to mains).

- Firstly, the user has to decide what will be the lowest power consumption mode of the system and configure the memory map so that it can accommodate this mode. This is done in `da1458x_config_basic.h` as described below:

```
/*****************************************************************************************/
/* Sleep mode memory map configuration. Only one or none of the following directives must be defined.   */
/* The selection determines the memory map configuration. Sleep mode is determined by host application sleep  */
/* mode by using arch_sleep API functions.                                               */
/*      -   CFG_MEM_MAP_EXT_SLEEP     Extended sleep mode                                */
/*      -   CFG_MEM_MAP_DEEP_SLEEP    Deep sleep mode                                    */
/*      -   none                      Always active                                      */
/*****************************************************************************************/
#define CFG_MEM_MAP_EXT_SLEEP
#undef CFG_MEM_MAP_DEEP_SLEEP
```

- Secondly, the user has to set, on compile time, the default mode to be used after startup. This is done in `user_config.h` as described below:

```
//default sleep mode. Possible values   ARCH_SLEEP_OFF, ARCH_EXT_SLEEP_ON, ARCH_DEEP_SLEEP_ON
const static sleep_state_t app_default_sleep_mode=ARCH_SLEEP_OFF;
```

The application can change the sleep mode dynamically during program execution by calling the API function arch_set_sleep_mode(sleep_state_t sleep_state). An example can be found in the `app_default_handlers.c` file at the end of the default_app_on_init() function.

Note: The sleep mode **shall not** change dynamically to a lower power consumption mode than the one set in the memory map configuration. For example, if `CFG_MEM_MAP_EXT_SLEEP` has been defined, the sleep mode shall change dynamically between Active and Extended Sleep and **NOT** Deep Sleep. If `CFG_MEM_MAP_DEEP_SLEEP` has been defined, the application can switch dynamically to all modes.

9.8   Application Description

The Peripheral Template example application (`empty_peripheral_template`) implements by default the basic BLE procedures such as advertising and connection. Note that no profiles are included in this application. The application uses the "Integrated processor" configuration.

9.9   Basic Operation

Supported services:

● No services supported apart from the default Generic Access primary service (UUID 0x1800) and Generic Attribute primary service (UUID 0x1801).

Features:

● Supports Extended Sleep mode by default.
● Basic Configuration Settings:
   ○ Advertising interval
   ○ Connection interval
   ○ Slave latency
   ○ Supervision timeout
● Advertising data:
   ○ Device name

The Peripheral Template application behavior is included in C source file `user_empty_peripheral_template.c`.

9.10   User Interface

A peer connected to the Peripheral Template application is able to:

● Check the advertising device name.

9.11   Loading the Project

The Peripheral Template application is developed under the Keil v5 tool. The Keil project file is the:

`projects\target_apps\template\empty_peripheral_template\Keil_5\empty_peripheral_template.uvprojx.`

9.12   Going Through the Code
9.13   Initialization

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

- `da1458x_config_advanced.h`, holds DA14580/581/583 advanced configuration settings.
- `da1458x_config_basic.h`, holds DA14580/581/583 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.
- `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.

9.14   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 Peripheral Template application registers the following callback functions:

- General BLE events:

```
static const struct app_callbacks user_app_callbacks = {
    .app_on_connection            = user_on_connection,
    .app_on_disconnect            = user_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_undirect_complete = default_app_on_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_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,
};
```

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()) are defined in C source file `user_empty_peripheral_template.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). In this case there are no user defined handlers but just one default handler default_app_on_init.

- 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). In this case there are no user defined handlers but just one default handler default_advertise_operation.

9.15 BLE Application Abstract Code Flow

Figure 63 shows the abstract code flow diagram of the Peripheral Template application. The diagram depicts the SDK interaction with the callback functions registered in `user_callback_config.h` and the functions implemented in `user_empty_peripheral_template.c`.
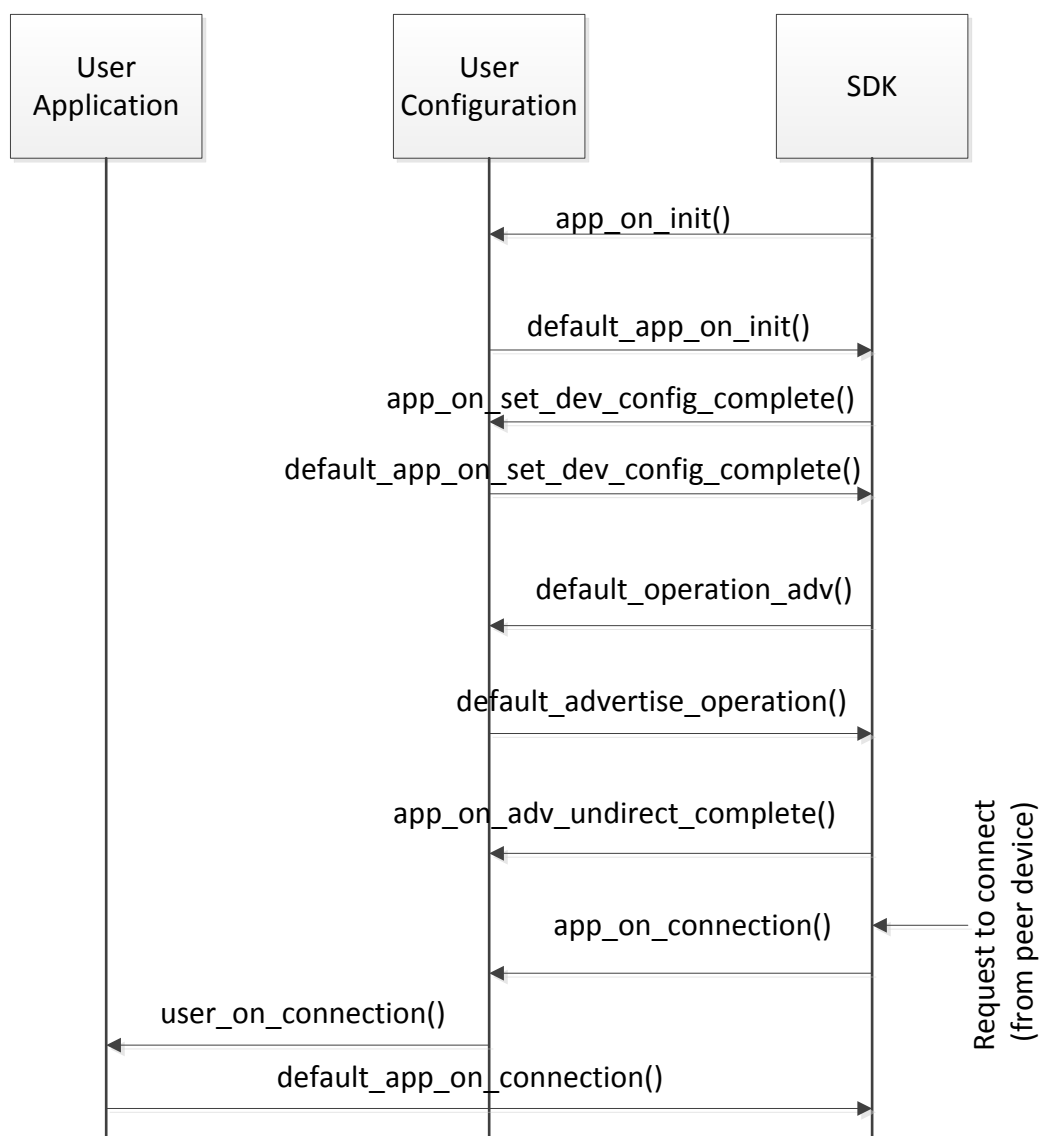

Figure 63: Peripheral Template Application - User Application Code Flow

9.16 Building the Project for Different Targets and Development Kits

The Peripheral Template application can be built for three different target processors: DA14580, DA14581 and DA14583. Also, the user has also to select the correct Development Kit in order to build and run the application. For more details on how to build, run and use the debugger per DK see Ref. [1] or Ref. [2].

DA1458x Software Developer's Guide

Revision History

Revision	Date	Description
1.0	27-Aug-2015	Initial version. Applies to SDK 5.x for DA14580/581/583.
1.1	22-Jul-2016	Added pillar examples for Sleep Mode, Security, OTA, and All in One. Removed Keil 4 references.

DA1458x Software Developer's Guide

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

Information in this document is believed to be accurate and reliable. However, Dialog Semiconductor does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information. Dialog Semiconductor furthermore takes no responsibility whatsoever for the content in this document if provided by any information source outside of Dialog Semiconductor.

Dialog Semiconductor reserves the right to change without notice the information published in this document, including without limitation the specification and the design of the related semiconductor products, software and applications.

Applications, software, and semiconductor products described in this document are for illustrative purposes only. Dialog Semiconductor makes no representation or warranty that such applications, software and semiconductor products will be suitable for the specified use without further testing or modification. Unless otherwise agreed in writing, such testing or modification is the sole responsibility of the customer and Dialog Semiconductor excludes all liability in this respect.

Customer notes that nothing in this document may be construed as a license for customer to use the Dialog Semiconductor products, software and applications referred to in this document. Such license must be separately sought by customer with Dialog Semiconductor.

All use of Dialog Semiconductor products, software and applications referred to in this document are subject to Dialog Semiconductor's Standard Terms and Conditions of Sale, available on the company website (www.dialog-semiconductor.com) unless otherwise stated.

Dialog and the Dialog logo are trademarks of Dialog Semiconductor plc or its subsidiaries. All other product or service names are the property of their respective owners.

© 2016 Dialog Semiconductor. All rights reserved.

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

User Manual   **Revision 1.1**   **22-Jul-2016**