Expertise
and insight
for the future

Tuan Nguyen Anh

Over-the-Air Firmware Update for Bluetooth Low Energy Devices

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

30 September 2019

Author Title Number of Pages Date	Tuan Nguyen Anh Over-the-Air Firmware Update for Bluetooth Low Energy Devices 33 pages 30 September 2019
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Smart Systems
Instructors	Kimmo Sauren, Senior Lecturer

Internet of Things (IoT) is taking place all over the world. More and more devices start to join the IoT revolution, opening endless opportunities and connections to transform our lives. However, security is a big issue that should be taken care of properly. Since the IoT devices require connectivity to operate, they are becoming more vulnerable to external attacks. One way to ensure that our embedded devices operate securely is to deliver firmware updates on a regular basis to improve their function and security.

The thesis aims to study the mechanism of delivering Over-the-Air (OTA) firmware update to embedded devices with Bluetooth Low Energy (BLE). It starts with the introduction of embedded systems, BLE, device firmware update and OTA programming. A proof-of-concept application is also developed to demonstrate the feasibility and practicality of the method.

The project utilises the NRF52840 Development Kit as the prototyping hardware and its Software Development Kit to implement the desired functionalities. In the end, the prototype is successfully developed and tested, passing the requirements set at the design phase. It allows the device to operate as a sensor node in an IoT system and offers firmware OTA update feature.

Keywords	Bluetooth Low Energy, OTA, Internet of Things

Contents

List of Abbreviations

ATT	Attribute Protocol
BLE	Bluetooth Low Energy
CO2	Carbon Dioxide
CPU	Central Processing Unit
DFU	Device Firmware Update
GAP	Generic Access Profile
GATT	Generic Attribute Profile
HCI	Host Controller Interface
IoT	Internet of Things
L2CAP	Logical Link Control and Adaptation Protocol
LED	Light Emitting Diode
LL	Link Layer
LoRaWAN	Long Range Wide Area Network
MBR	Master Boot Record
nRF SDK	Software Development Kits provided by Nordic Semiconductor
OTA	Over-the-air
PHY	LE Physical Layer
SIG	Special Interest Group

SMP	Security Manager Protocol
SoC	System on a chip
SPI	Serial Peripheral Interface
SVC	Supervisor Call
UART	Universal asynchronous receiver-transmitter
USB	Universal Serial Bus

1 Introduction

This chapter introduces the problem addressed by this thesis work. The first two sections present the current interests in embedded devices and the need for a device firmware update process. After that the last section states the purposes and goals of the project.

1.1 Increasing interest in embedded devices

Embedded devices are highly specialised devices meant for one or a few special purposes. They are usually embedded into a larger object or a part of bigger systems that serve greater purposes. The hardware of these devices is relatively small and straightforward featuring, for example, a 32-bit microcontroller, a dedicated processor, called an application-specific integrated circuit or digital signal processor. The software also differs from application software. Unlike application software, embedded software is developed toward specific hardware in which the capabilities and the addition of third-party components are strictly controlled. It may use either no operating system or a real-time operating system. Embedded devices have extensive applications in commercial, consumer, industrial, automotive, healthcare and many other industries because of their diminutive and inconspicuous nature. In the late twentieth century and at the beginning of the twenty-first century, the development of the Internet and the advancement in material science brings the potential to computerise and connect industrial and consumer components such as cars, manufacturing plant, wristwatch, or even home kitchen.

The Internet of Things (IoT) is an emerging topic of technical, social, and economic significance. The term IoT refers to the scenarios where objects, sensors, and everyday items with networking connectivity gather data about their surrounding environment so that they can be used to analyse, interpret, decide, and act on that data and other associated information [1]. IoT allows devices to generate, exchange, and consume data to form a fully interconnected “smart” world operating with minimal human intervention toward predefined goals. As a result, physical objects and machines are being gradually computerised by, for example, adding networking connectivity, to take advantage of the transformational potential of the IoT.

1.2 The needs for firmware update

As the number of devices connected to the internet increase, new opportunities to exploit the vulnerabilities also grow. IoT devices, in particular, are the ideal candidates for attackers, for several reasons: Firstly, an IoT device is usually connected to the internet in some ways, so attackers do not have to have physical access to exploit. Secondly, many IoT devices are operated unattended, which means that the attacks might not be recognised until something significant happens. Thirdly, embedded systems are inherently difficult to update, which leaves them more vulnerable to newly-discovered security issues. Hence, new IoT devices should be developed with upgradability in mind. It will allow the delivery of regular software updates from manufacturers, which not only patch bugs and enhance security, but also add new features and functionalities.

1.3 Purposes and goals

The first part of the thesis work aims to study the technologies that relate to Firmware OTA update such as Bluetooth Low Energy (BLE), the device firmware update (DFU) process. Besides, it presents and discusses the firmware update process in detail. Finally, a proof-of-concept application will be developed that features Firmware OTA update over BLE for an IoT device to demonstrate the feasibility and practicality in real-world scenarios.

2 Background

This chapter describes the main concepts related to the project work. Firstly, the chapter provides a definition and an example of embedded systems. Then, it introduces the Bluetooth Low Energy along with some common concepts. Finally, the last two sections provide an overview of the project and the chosen hardware.

2.1 Embedded systems

The definition of embedded systems varies among people. Some people, who usually work with desktop or web application, may refer to mobile devices as an embedded system. Some, however, who have developed application for 8-bit microcontroller system seem not to agree. Though it is hard to define, embedded systems share some common characteristics:

- Application-specific: As opposed to general-purpose computing devices such as a personal computer, embedded devices are developed with predefined use cases in mind and only offer a limited set of functions.
- Real-world interaction: Embedded systems are developed to solve real-world problems. They can be used to control some hardware, sense the environment with sensors, or manipulate the physical world with actuators.
- Constraints of hardware: The resources, such as energy, CPU, memory, are limited. There are several ways to classify hardware used in embedded systems in terms of resources. For instance, the Internet Engineering Task Force has its taxonomy of constrained-node networks [2]:
  (a) Class 0 devices have smallest resources, typically <10 KiB of RAM and <100 KiB of flash size.
  (b) Class 1 devices have medium-level resources, typically >10 KiB of RAM and >100 KiB of flash size.
  (c) Class 2 devices have more resources, typically >50 KiB of RAM and >250 KiB of Flash size.
- Constraints of software: Some systems require the software must act deterministically, soft real-time, or hard real-time. Other systems require that the software be fault-tolerant in the face of errors (e.g., flight control systems on an aeroplane) or to cease the operation at the first sign of problems (e.g., medical devices). [3, p.1]

An example of an embedded device is shown in Figure 1, an IoT sensor kit named Nordic Thingy:52. It is an easy-to-use prototyping platform helping in rapidly building prototypes and demos [4].

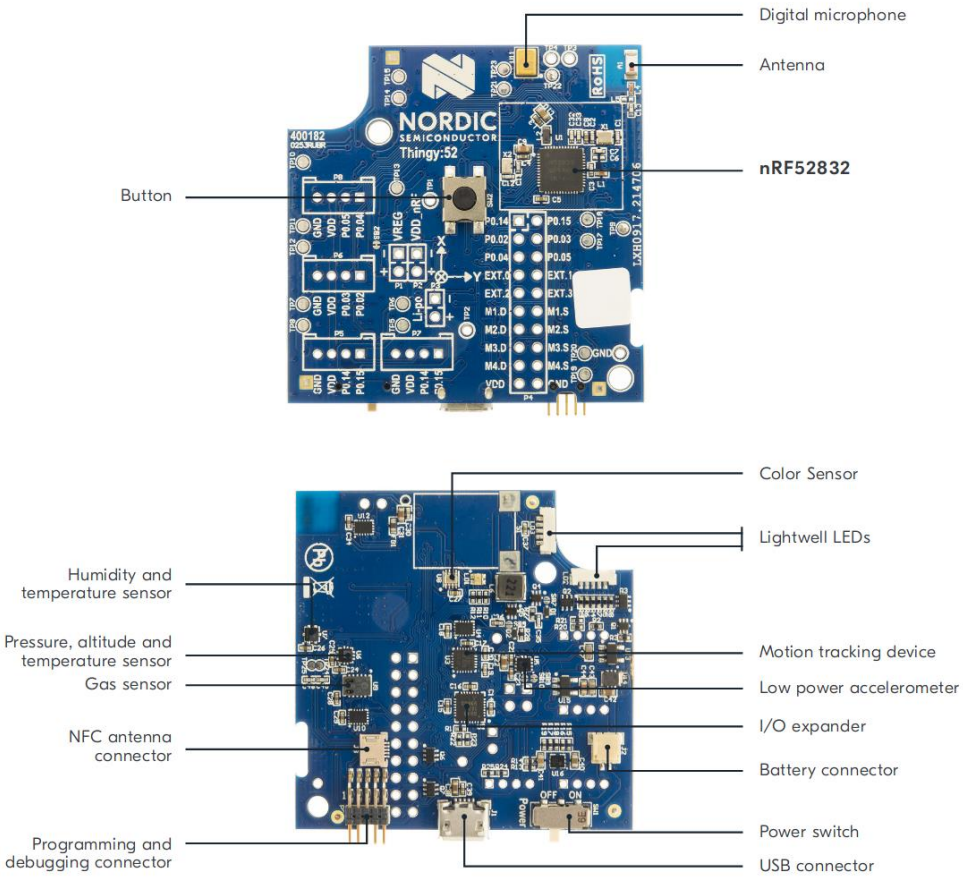

Figure 1. Nordic Thingy:52. Adapted from [4].

As illustrated in Figure 1, Nordic Thingy:52 consists of several components of a typical embedded system:

- Microcontroller/Microprocessor/System-on-Chip (SoC): This is the heart of the system as it glues all components together. Its key job is to communicate with sensors for input variables and actuators for output variables as well as processing the data. Others hardware are built into the chip to assist these communications such as analog-to-digital converters, digital-to-analog converters.
- Sensors: Each application interests to a different set of data for its purposes. The data collected by these sensors serve as the input to the system. The example device includes various types of sensors, such as colour sensor, humidity and temperature sensor, and others.
- Actuators: In order to manipulate the environment in which the device operates, some actuators have to be built into the system. After processing the input data collected by sensors, the central controller outputs signal to control some actuators, for instance, LEDs, screen, to perform the desired actions.

- Power supply: This is the essential component of the system. Two primary sources of power supply for an embedded system are wall power and batteries. Hence, the system could be powered by any one of these models:
  (a) Wall powered
  (b) Wall powered with batteries as a backup
  (c) Battery-powered
- Interfaces with other systems: An embedded system usually has some interfaces to communicate with other systems. They can be, for example, programming and debugging interface, or USB port. They are essential not only to the developers, who have to use debugging interfaces to aid the development process but also the user to extend the existing functionalities by incorporating other devices.
- Other application-specific components: Some systems require specific parts for their application. For example, the nRF52832 SoC needs an external antenna for wireless communication.

2.2 Bluetooth Low Energy (BLE)

Bluetooth Low Energy (BLE) is a subset of Bluetooth classic and was first introduced as a part of the Bluetooth 4.0 core specification. It was started as a project initially developed by Nokia called Wibree and was later adopted by the Bluetooth Special Interest Group (SIG). [5, p.1]

The key design goals of BLE are robustness, low power consumption, and low cost. It was intended to replace cables with short-range wireless communication connecting portable and fixed electronic devices. In addition, the BLE was also designed for use cases and applications with low data rates and low duty cycles [6, p.180]. For example, a wireless sensor node with BLE for data transmission, which only activates for a few minutes a day, can last for months with a single coin cell battery. The most appealing feature of BLE is that most major platform, as listed below, support Bluetooth 4.0 and Bluetooth Low Energy:

- iOS 5+
- Android 4.3+
- Apple OS X 10.6+
- Windows 8,10
- GNU/Linux Vanilla BlueZ 4.93+

Hence, engineers and product designers can easily develop new products that are able to talk to any modern computing platform, especially mobile devices. BLE can also lower the barrier to adopt new products since users are already accustomed to using smartphones or tablets, which means the learning curve of using new external devices is gentler [5, p.2]. As a result, BLE is widely adopted in consumer electronic devices. For example, it exists in most smartwatches nowadays for data exchange with users' mobile phones.

2.2.1 Configurations and network topology

A Bluetooth core system consists of a Host and one or more Controllers. It means that a BLE system can exist with or without Bluetooth classic. Figure 2 illustrates two types of BLE system.

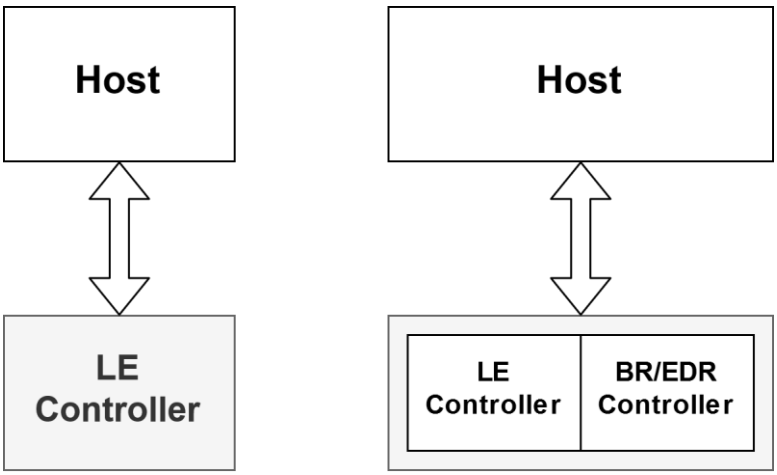

Figure 2. Configurations for devices with BLE.

A Host is the upper layers in the BLE protocol stack which are below the non-core profiles and above the Host Controller Interface (HCI). A Controller is the lower layers in the BLE protocol stack including the physical layer, Link Layer and optionally HCI. These layers can be implemented in a single integrated circuit, or they can be split and communicate through other protocols (UART, USB, SPI, or other).

A Bluetooth Low Energy device can communicate with the other peers in two ways: broadcasting or connections. Broadcasting allows the device, which acts as a broadcaster, to send one-way data to other scanning device or receiver in the listening range,

which acts as an observer. The device broadcasts data by sending an advertising packet containing 31-byte payload, which includes data describing itself, its capabilities, and other user-defined information. BLE also supports a secondary advertising packet, which is also 31 bytes. In total, the device can transmit up to 61 bytes to other interested peers without the need of making connections. However, advertising packets are not encrypted. Therefore, they are not suitable for sensitive data. To transmit data bi-directionally, or a large amount of data, two devices have to make a connection. One device acts as central (master) and initiates the connection while the other device acts as a peripheral (slave). Once a connection is established, the peripheral device stops advertising, and both devices can start exchanging data. The advantages of connections compared to broadcasting are security and data organisation with services and characteristics.

2.2.2 The BLE protocol stack

The BLE protocol stack is a combination of several layers that provide the functionality required by the BLE protocol specification.

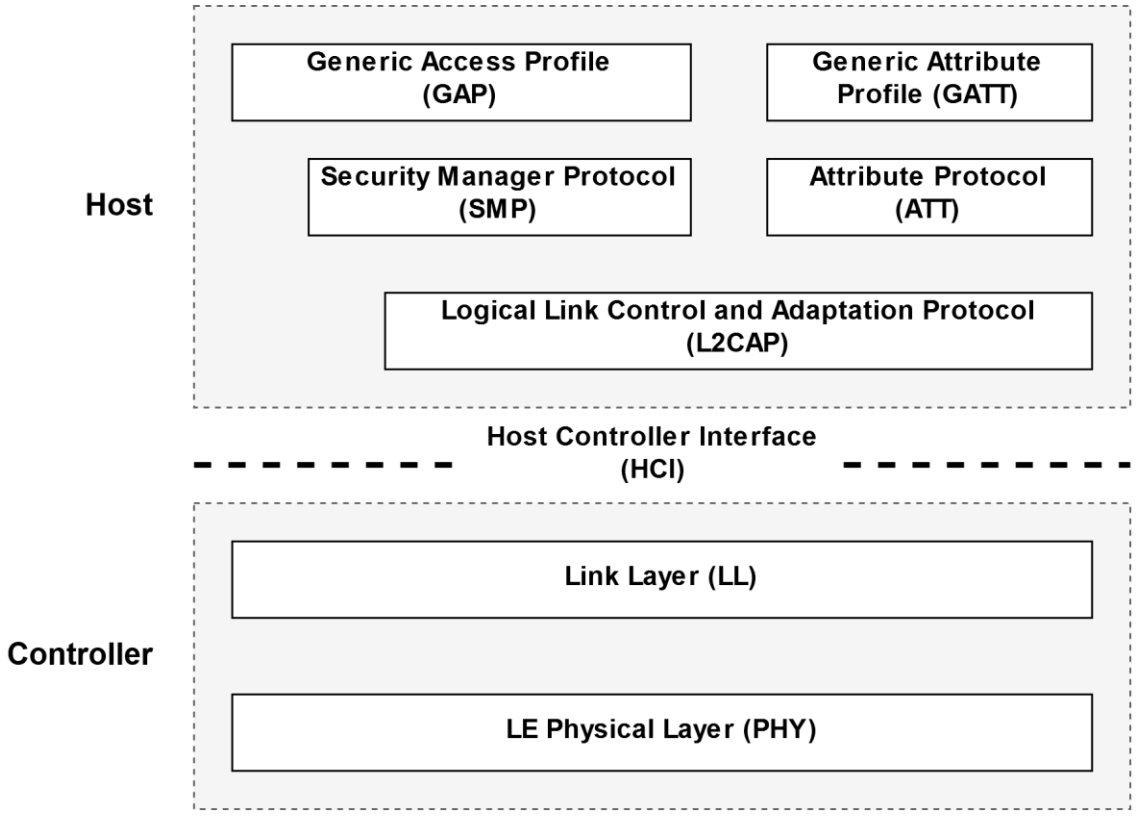

Figure 3. The BLE stack.

As shown in Figure 3, each layer belongs to either Host or Controller. The HCI is an optional component and appears in both Host and Controller.

- Host:
  (a) Generic Access Profile (GAP): This layer represents the base functionality common to all BLE devices such as modes and access procedures used by the transports, protocols and application profiles. Its services include device discovery, connection modes, security, authentication, association models and service discovery. [6, p.198]

  (b) Generic Attribute Profile (GATT): This layer represents the functionality of the attribute server and, optionally, the attribute client. GATT provides a hierarchy and abstraction model in terms of services, characteristics, and attributes for data exchange between devices. It provides interfaces for discovering, reading, writing, and indicating service characteristics and attributes. [6, p.198]

  (c) Attribute Protocol (ATT): This layer implements the peer-to-peer protocol between an attribute server and an attribute client. To read and write values of attributes on a peer device with an ATT server, the client sends requests to the server, and the server sends data to the client. The client can also send commands, and confirmations to the server while the server can also send notifications, and indications to the client as well. [6, p.198]

  (d) Security Manager Protocol (SMP): SMP is the peer-to-peer protocol used to generate encryption keys and identity keys. It also manages the storage of the encryption keys and identity keys and is responsible for generating random addresses and resolving random addresses to known device identities. During the encryption or pairing procedures, SMP interfaces directly with the Controller to provide stored keys used for encryption and authentication. [6, p.197-198]

  (e) Logical Link Control and Adaptation Protocol (L2CAP): L2CAP has two main functionalities. Firstly, it encapsulates data from multiple upper protocols into the standard BLE packet format and vice versa. Secondly, it performs packet fragmentation and recombination for transmission. The maximum payload size of the BLE packet is 27-byte including 4-byte L2CAP header. [5, p.25]

- Host Controller Interface (HCI): HCI is a set of commands and events for the Host and the Controller to interact with each other. It also transports data packet with pre-defined flow control in addition to other procedures. The physical transport protocols can be, for example, UART, USB, SPI. HCI is an optional component of the BLE stack, and only appears in some configurations. [5, p.24-25]

- Controller:
  (a) Link Layer (LL): This layer is a combination of hardware and software and is responsible for complying with timing requirement as defined in the specification. It manages the link state of the radio, which is how the device connects to other devices. It also takes care of encoding

and decoding of Bluetooth packets from the data payload and parameters. [5, p.17-18]

(b) LE Physic Layer (PHY): The PHY layer contains the analogue communications circuitry, responsible for transforming a stream of data into required formats for transmitting and receiving packets of information on the physical channel. The radio uses the 2.4 GHz ISM (Industrial, Scientific, and Medical) band to communicate and divides this band into 40 channels from 2.4000 GHz to 2.4835 GHz. [6, p.200]

2.3 Project overview

The thesis work focuses on the implementation of the device firmware update strategy for BLE-enabled devices. In the end, users can update new firmware for their devices through a mobile application on both Android and iOS phones. To achieve this goal, firmware for a BLE-enabled device is developed that features OTA firmware update and a practical application. The device acts as a sensor node in an IoT system. Users (such as mobile applications, desktop applications) can connect to the device to retrieve the sensor data or even control other devices, for example, LEDs. Chapter 3 discusses further the chosen system architecture, the applications, and the OTA mechanism.

2.4 Hardware overview

There are many factors that affected the hardware of choice for the project. The most important is the support of BLE and the ease of development. The nRF52840 SoC from Nordic Semiconductor was chosen after rigorous evaluations of different options. The initial prototype firmware uses the nRF52840 DK, the development kit for the nRF52840 SoC, for the development process.

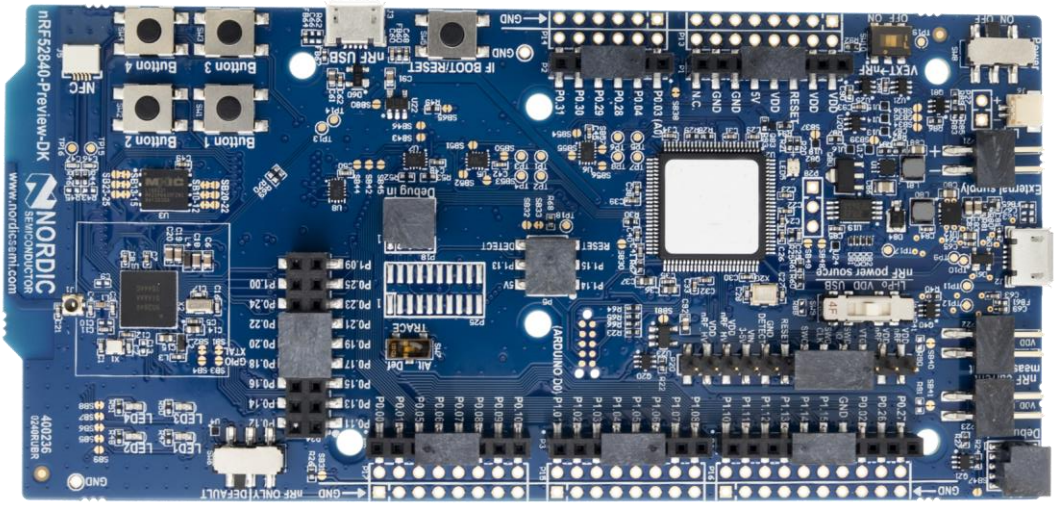

Figure 4. The nRF52840 DK. Adapted from [7]

The nRF52840 SoC is the most advanced member of the nRF52 series SoC family. It has protocol support for Bluetooth 5, Bluetooth mesh, Thread, Zigbee, 802.15.4, ANT and 2.4 GHz proprietary stacks. The nRF52840 is built around the 32-bit Arm Cortex-M4 CPU with the floating-point unit running at 64 MHz. The flash and RAM size if 1 MB and 256 KB respectively, which is generous even for demanding applications. [7]

The Bluetooth protocol stack on the hardware is implemented as SoftDevice and distributed as proprietary software. It is a Bluetooth Low Energy Central and Peripheral protocol stack solution which integrates a BLE Controller and Host and provides API for building BLE application. There also exists open-source BLE protocol stack implementation for the nRF52840, such as in the Zephyr project [8]. However, for the ease of development, the SoftDevice is used along with the provided nRF Software Development Kit (SDK).

3 Firmware update for embedded devices

This chapter explains the device firmware update (DFU) mechanisms in detail for BLE-enabled embedded devices. Firstly, the chapter introduces the concepts of DFU. After that, it provides some examples of existing solutions for DFU based on OTA programming. Secondly, it presents the system architecture of the prototype that was developed as a part of this thesis work.

3.1 Device firmware update

The device firmware update is the process of updating the software of an operating device to a newer version. It aims to deliver fixes and new features to improve the system's stability and security. As discussed in Chapter 1, keeping up to date is becoming more important nowadays, especially for connected devices.

In order to perform firmware updates once the device is deployed, the initial firmware that has been loaded by the product manufactured has to have the ability to load new code without any external programming hardware or tools. The bootloader is a mandatory component in every system that supports DFU. Once the microcontroller resets, the bootloader is the first software that is executed. It is responsible for deciding whether to boot into an existing application or to obtain and install new firmware images. There are two common approaches when implementing the bootloader: a single bootloader, and a multi-stage bootloader. Figure 5 shows the memory map and the boot flow of a single bootloader.

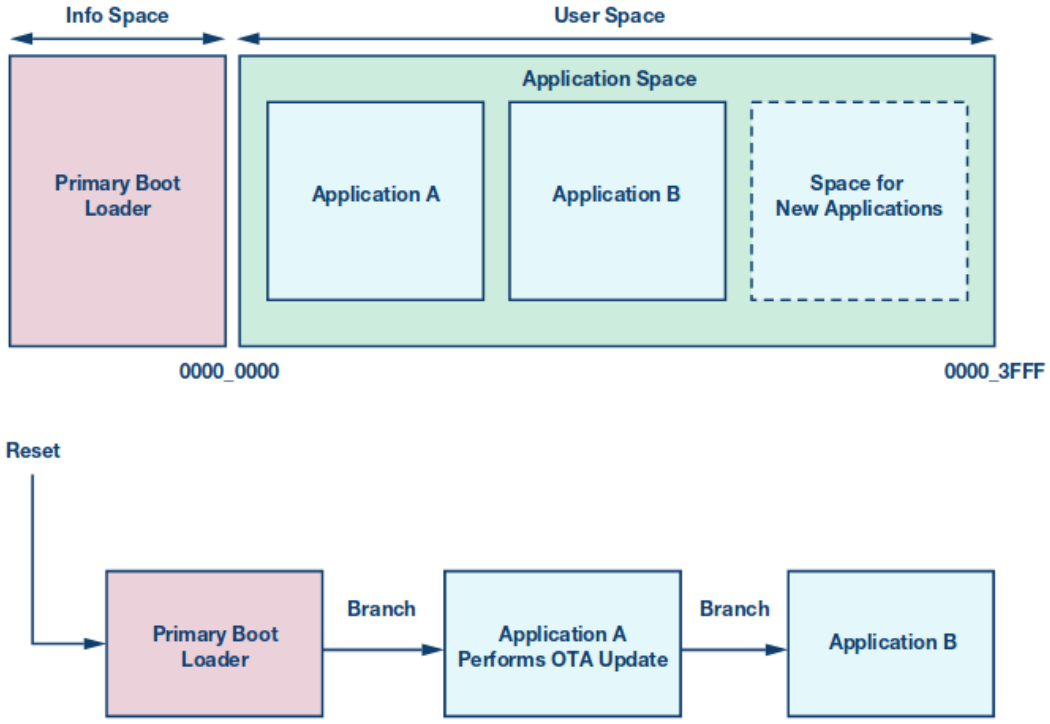

Figure 5. Example memory map and boot flow of a single bootloader. Adapted from [9].

In addition to the bootloader, the original application has built-in functionality for the firmware update process. After initiating the update request, either by the device itself or by an external party, the original application (for example, Application A in Figure 5) downloads, verifies and installs the new firmware. Next, it transfers the control to the new application by performing a branch instruction to the reset handler of the new application. However, the drawback of this approach is that in some systems, the only safe way to terminate a program is through a reset. Another approach is to split the bootloader into separate stages. Figure 6 demonstrates this implementation in detail.

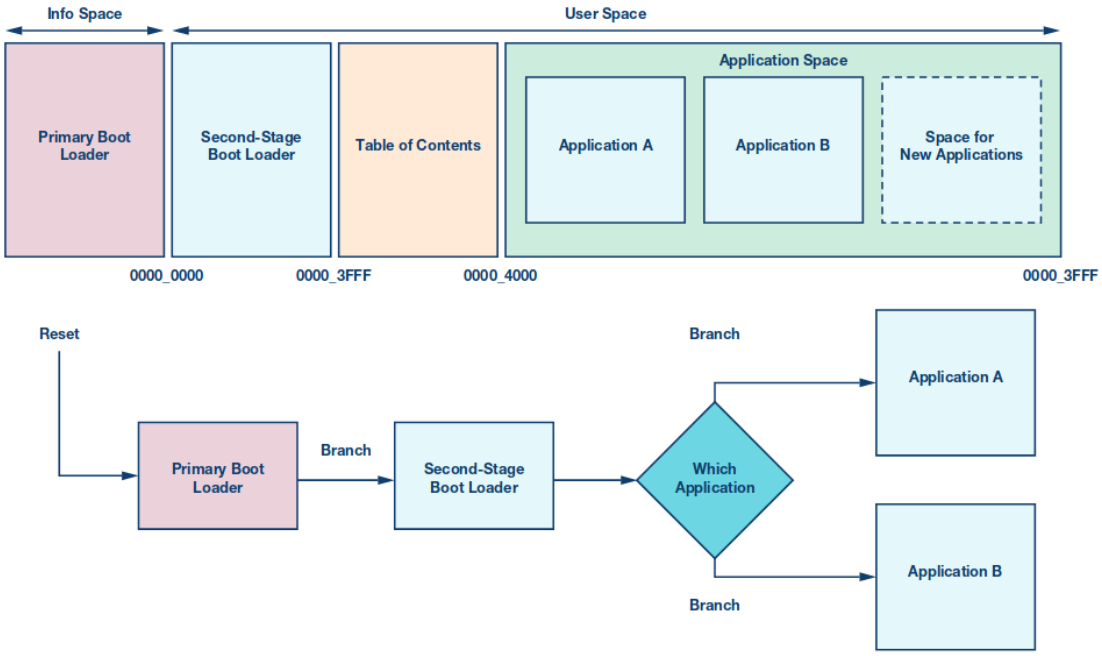

Figure 6. Example memory map and boot flow of a multi-stage bootloader. Adapted from [9].

The first stage bootloader may offer minimal functionality, such as the configuration of the device, initialisation of primary peripherals, and preparation for next stage bootloaders. Some hardware vendors hardcode this type of bootloader in the device. Thus, it cannot be changed by users. The second stage bootloader may implement more complex functionality. It can either initiate the firmware update process or boot into the desired application. With this approach, the application can get rid of a large amount of update-related code. To perform an update while the device is operating, a pre-defined value indicating update request is set to one special register, which does not change after rebooting the device, followed by a soft reset triggered by the application. After the second stage bootloader has been loaded, it acknowledges the request and branches

into the firmware update process. As a result, the multi-stage bootloader reduces the code duplication and simplifies the application-specific software.

Another important consideration when designing a firmware update process is the choice of the data transport protocol. Standard communication protocols, such as UART, USB, SPI, BLE, LoRaWAN can be employed to transmit the new firmware from a master device to other slave devices. The decision which protocol to use depends on how the update process is carried out. For in-field firmware update, it is usually performed by a human and requires a physical connection between the microcontroller and the updater tool. The advantage of this method is that in cases of update error, the operator can manually retry the process. For Over-the-Air firmware update, wireless connectivity is required to manage the update process. The running application manages its firmware update through the mechanism described above. This method can scale to a large number of devices and require minimal human effort. However, in case of update error, a retry may be challenging to support.

There are many challenges that the designers have to address when developing the firmware update mechanism. First, the developers must choose a suitable binary format for the application and the method of delivering and organising data in the target system. Since most of the application binary is too large to fit in a single transfer, the data are often split and delivered in chunks. Depending on the system constraints, the bootloader should implement an efficient strategy to verify the incoming data and commit them to the flash. Next, the update process can fail for various reasons such as transmission error, transmission failure, transmission loss, or the new firmware does not function properly. Hence the system should be able to detect these issues and take appropriate actions to preserve the availability and responsiveness. Finally, the design of the system must address common security threats. The bootloader must be able to verify that the new firmware is from a trusted party. The data sending from the master device must also be encrypted and not corrupted when arriving at the slave device.

3.2 Over-the-Air (OTA) programming

Over-the-Air programming is a mechanism that delivers the firmware update wirelessly to devices such as IoT nodes, cell phones. There are a number of standards that specify

OTA functions. In general, they share the same high-level architecture, as illustrated in Figure 7.

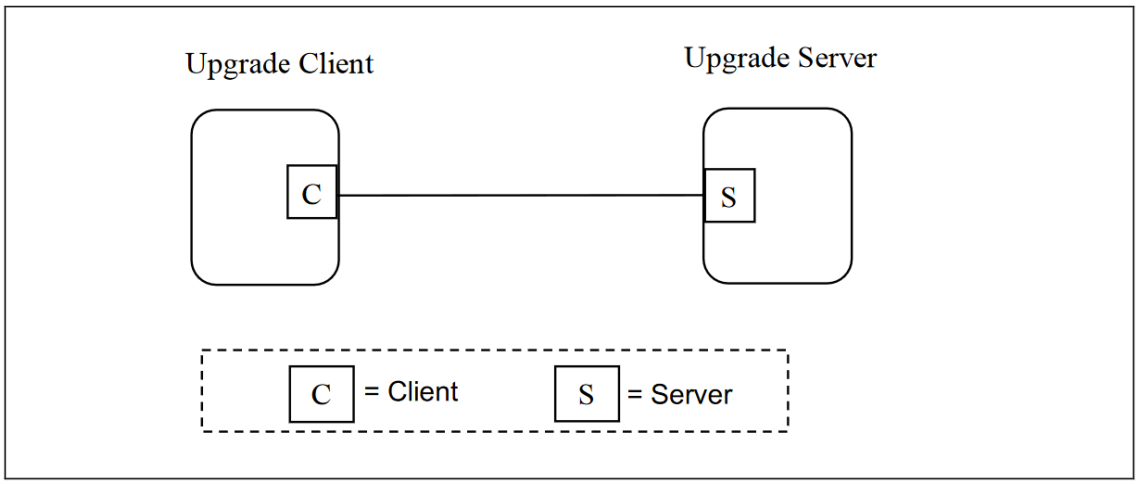

Figure 7. A typical OTA Upgrade Cluster. Adapted from [10].

Depending on the requirements of the application, one must adapt the existing standard to develop a reliable way of updating device firmware. For example, if the device is a smartwatch, the users' phone can transfer the data through short-range radios such as BLE. In contrast, a wireless sensor network scattered in a city may require long-range radios such as LoRaWAN or cellular networks. In addition, the OTA update procedures between the OTA client and OTA server should also be designed with care to make sure the OTA process can work seamlessly, thus reducing the interruption of devices operations. Figure 8 describes the OTA messages utilised in ZigBee OTA Upgrade Cluster standard.

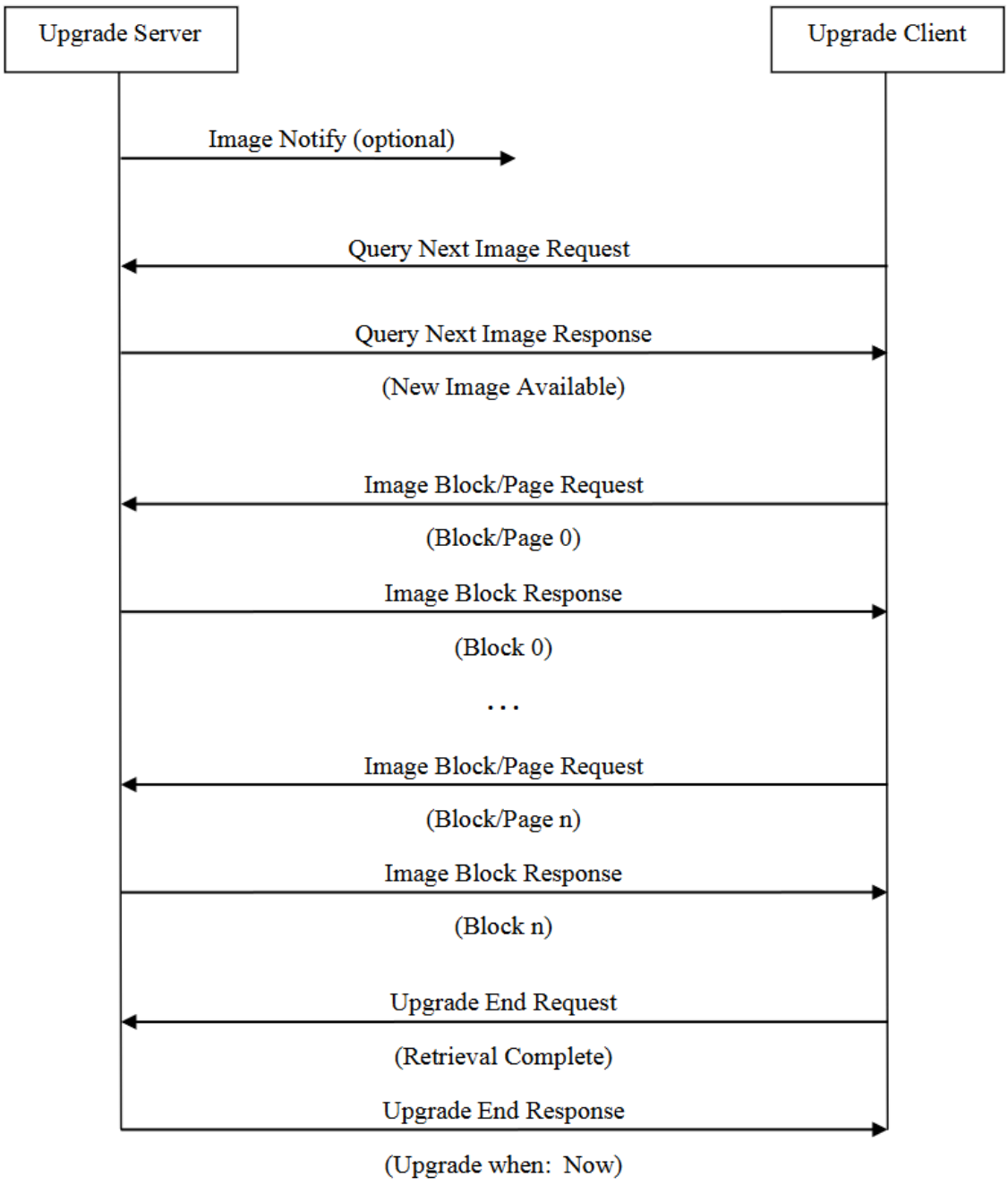

Figure 8. OTA Upgrade message diagram. Adapted from [10].

As can be seen from Figure 8 above, the update process consists of four phases. Firstly, the server notifies the client that new firmware exists. Secondly, the server and the client start a negotiation on the latest firmware image properties. After that, the server transmits the image block by block while the client stores them to a temporary storage. Finally, the server and the client agree on when to start the upgrading. Usually, the bootloader on

the client handles the process of receiving new images and upgrading the device firmware. The client then reports to the server that the upgrading process is successful or not and any actions should be taken if any abnormality arises.

4 Prototype

This chapter describes in detail the design and implementation of a prototype firmware for a BLE-enabled device. It explains the overall architecture, the firmware update procedure as well as the required building blocks. Then, the following sections explain the implementation of each component in the firmware in detail.

4.1 System architecture

4.1.1 Firmware building blocks

The architecture of the firmware is based on the requirements discussed in the previous chapters. The firmware consists of several components. Each of them, as shown in Figure 9 below, serves different purposes to form a functional system.

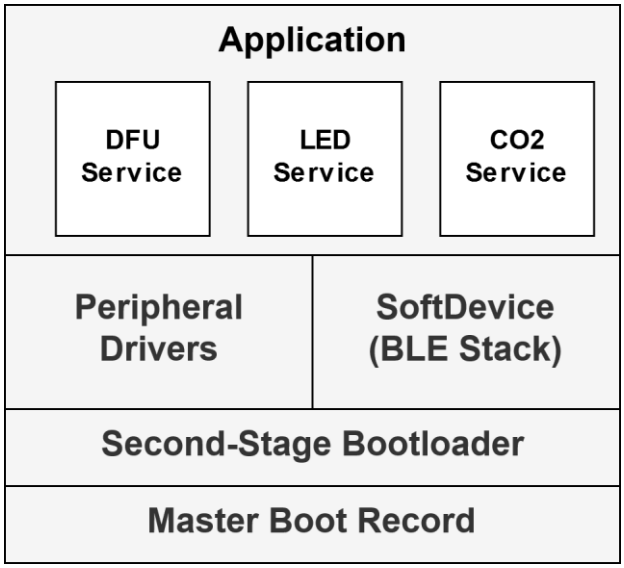

Figure 9. The firmware building blocks.

The Master Boot Record (MBR) is a required component in the system using NRF52840 SoC. MBR provides an interface in-system update of the application, the SoftDevice, and bootloader firmware. The MBR ensures that if any unexpected resets arise during an ongoing update process, the bootloader can recover to a deterministic state. In regular operation, the MBR branches the bootloader after resetting the device.

The second-stage bootloader or bootloader, in short, serves two main functionalities: to handle in-system procedures or to start the application in regular operation. In addition, it is responsible for the integrity of the firmware, including the SoftDevice, the application, or the bootloader itself. If neither the system is an update process nor a valid application presents, the bootloader can be implemented to wait for a valid application transmitted or put the system in sleep mode. In this state, external parties can wake up the device, for example, by pressing a specific button.

The SoftDevice is a BLE Central and Peripheral protocol stack solution. It integrates a Bluetooth Low Energy Controller and Host and provides a full and flexible API for building BLE application with NRF52840 SoC. It is distributed as a precompiled and linked binary image. The SoftDevice consists of three main components:

- SoC Library: common APIs for shared hardware resource management
- SoftDevice Manager: specific APIs for SoftDevice management
- Bluetooth 5 Low Energy protocol stack: implementation of protocol stack and APIs

The peripheral drivers are the hardware access layer and the driver layer, which are distributed in the nRF SDK. They abstract common hardware of the chip and provide easy-to-use APIs to drive them. Consequently, they speed up the development process and improve the quality of the software. However, there is a trade-off in the ease of development and the ability to fully control the hardware.

The last and most essential component in the system is the application. It is designed to behave like a sensor node in an IoT system. The main functionality of the application is to transmit the sensor data to interested parties and to provide an interface to control its connected devices such as LEDs. The firmware also has to support OTA firmware updating process. As a result, these requirements can be achieved by implementing three separate services on top of the BLE stacks:

- DFU service: It is a BLE service that enables entering DFU mode from a BLE application during a firmware updating process. The service is intended to be used on devices that do not offer physical access or to provide better user experience when performing the firmware update.
- CO2 service: It is a BLE service that provides data collected from the CO2 sensor attached to the device. In addition, users can set the threshold to trigger physical alerts when the CO2 measurement exceeds a particular value.
- LED service: It is a BLE service that offers the ability to retrieve the state of the LEDs on the device. The service also implements a mechanism to notify users of their new events.

4.1.2 Firmware OTA updating procedure

The firmware OTA updating process requires two devices: the target and the controller. The controller is the device that requests the update and transfers the new firmware image, which can contain a new application, SoftDevice, bootloader, or a combination of SoftDevice and bootloader. The target is the device that is updated with the new firmware. [11] The update process requires two steps:

- The transportation of the update package over wireless communication. In this case, the project uses BLE.
  (a) The controller connects to the target and sends a command to request the DFU.
  (b) The controller sends the initial package that contains the metadata of the update package. Then, the target decides if it is valid and proceed to request the new firmware data.
  (c) The controller repeatedly sends the firmware over BLE, while the target stores them in temporary locations.
  (d) Once the transmission is completed, the target performs the final validation and starts persisting the new firmware.
- The copy of the new firmware to the program memory. The target may want this operation power fail-safe since invalid corrupted firmware will put the device into an unstable state.

To safely perform the firmware update, the new firmware is copied to a separate location in the memory until it has been validated. Once the latest firmware is verified and activated, the old image will be replaced. This technique is called A/B updates or dual-bank and single-bank updates. It ensures that there is always one valid firmware on the device. However, it requires that the device has enough free memory to store the new firmware during the updating process.

4.2 SoftDevice

Since SoftDevice is distributed as a prebuilt binary image, it is not possible to modify its source code. The SoftDevice API is available to applications as a C programming language interface based on Supervisor Call (SVC) and defined in a set of header files. When the application code calls SoftDevice API, SVC interrupts are being triggered. The SoftDevice is then responsible for handling the interrupt. To transfer the control to the application after processing the call, the SoftDevice triggers interrupts to signal the events. It is now the responsibility of the application for handling the interrupts and processing the events. The advantage of this communication model is that the application does not have to link the SoftDevice in the binary. Instead, it is required to include relevant headers containing information for invoking the API functions. In addition, the SoftDevice binary must have existed on the device already.

4.3 Bootloader

The project uses the secure bootloader included in the nRF SDK. The bootloader consists of two components in the program memory: the bootloader code and the bootloader settings page.

The bootloader is built as a separate image and loaded to the device before the application. It implements the logic to boot the application and DFU functionality. It also implements the DFU BLE service as the transport layer to facilitate the firmware update over BLE. When the device enters the DFU, the device advertises the service and waits for other devices to connect. Once there is a connection, the master device sends the update package and triggers the update process. The bootloader also embeds a public key, which is required for the DFU functionality.

The bootloader settings page resides in non-volatile memory. It is used to keep the bootloader and DFU information: current firmware, pending firmware, the progress of the firmware update, progress of the firmware activation, current firmware versions (application and bootloader), transport-specific data. Listing 1 shows the pseudo implementation of the bootloader's main functionalities.

```
int main(void)
{
    // ...
    // Initialize the bootloader
    nrf_bootloader_init();
    // Boot the main application in case of DFU module is not enabled
    nrf_bootloader_app_start();
    // ...
}

ret_code_t nrf_bootloader_init()
{
    // ...
    bool dfu_enter;
    // Check if an update needs to be activated and activate it.
    activation_result = nrf_bootloader_fw_activate();
    switch (activation_result) {
        case ACTIVATION_NONE: dfu_enter = dfu_enter_check();
        case ACTIVATION_SUCCESS_EXPECT_ADDITIONAL_UPDATE: dfu_enter = true;
        case ACTIVATION_SUCCESS: bootloader_reset(true);
        case ACTIVATION_ERROR: on_error()
    }
    if (dfu_enter) {
        // ...
        nrf_dfu_init();
    } else {
        // ...
        nrf_dfu_settings_backup();
        nrf_bootloader_app_start();
    }
    // ...
}

void nrf_bootloader_app_start(void)
{
    // Get the start address of the application
    uint32_t start_addr = get_start_addr();
    // Disable and clear interrupts
    disbale_irq();
    // Set up vector table
    set_up_vector_table();
    // ...
    // Start the application
    app_start();
}
```

Listing 1. Snippet of the bootloader implementation.

As can be seen from listing 1, the bootloader first initialises and checks whether an update needs to be activated or not. Four cases can occur at this stage:

- ACTIVATION_NONE: No update was found. Then the bootloader performs an additional check whether should it enter the DFU or not.
- ACTIVATION_SUCCESS_EXPECT_ADDITIONAL_UPDATE: The firmware update is successfully activated, but there are likely more updates to be transferred. This scenario can be seen when users perform the update on more than two components of the system. These components are the application, the bootloader, and the SoftDevice.

- ACTIVATION_SUCCESS: The firmware update is successfully activated. The device will trigger a reset.
- ACTIVATION_ERROR: The firmware update could not be activated. The error handler will be called in this situation.

The final step is to start the application if it exists in the program memory. In addition, the bootloader also takes care of other duties such as preparing the interrupt vector table, exchanging information (such as advertisement name to use when entering DFU mode, bonding information, system attribute table) with the application.

4.4 Application

As described above, the application features three functionalities: the DFU service, the LED service, and the CO2 service. The DFU service is modified from the nRF SDK. The other two services are developed as part of the project.

It is straightforward to integrate the DFU service into the application. The application defines an event handler and initialises a variable that captures the state of the service. After enabling the BLE stack, the service observes the BLE events and reacts based on the defined handler.

The LED service offers the ability to retrieve the state of LEDs on the sensor node. It includes a LED state characteristic. After connecting to the node, users can get the LEDs state or set up the notification to be informed when the LEDs state change. Listing 2 shows the interface of the LED service.

```
#define BLE_LED_SERVICE_DEF(_name)                                  \
static ble_led_t _name;                                             \
NRF_SDH_BLE_OBSERVER(_name ## _obs,                                 \
                     BLE_LED_BLE_OBSERVER_PRIO,                     \
                     ble_led_on_ble_evt, &_name)

// Base UUID: 709E0000-C6D8-45CE-BA5A-406667428FCE
#define BLE_LED_SERVICE_BASE_UUID   {0xce, 0x8f, 0x42, 0x67, 0x66, 0x40, 0x5a,
0xba, 0xce, 0x45, 0xd8, 0xc6, 0x00, 0x00, 0x9e, 0x70}

// Service & characteristics UUIDs
// LED Service:                709E0001-C6D8-45CE-BA5A-406667428FCE
//  LED state Characteristic:  709E0002-C6D8-45CE-BA5A-406667428FCE
#define BLE_LED_SERVICE_UUID        0x0001
#define BLE_LED_STATE_CHAR_UUID     0x0002

// LED Service event type
```

```
typedef enum
{
    BLE_LED_STATE_EVT_NOTIFICATION_ENABLED,
    BLE_LED_STATE_EVT_NOTIFICATION_DISABLED,
} ble_led_evt_type_t;

// Function for initializing the LED Service
uint32_t ble_led_init(ble_led_t *p_led, ble_led_init_t *p_led_init);

// Function for handling the Application's BLE Stack events
void ble_led_on_ble_evt(ble_evt_t const *p_ble_evt, void *p_context);

// Function for updating the LED state Characteristic
uint32_t ble_led_state_characteristic_update(ble_led_t *p_led, uint8_t
*led_state);
```

Listing 2. The LED service interface

To utilise the LED service, the application must follow the following steps:

- First, it defines an instance of the service. The macro `BLE_LED_SERVICE_DEF` also registers the service-specific BLE stack events handler in the SoftDevice.
- Second, the application creates a service handler function and registers it through the `ble_led_init` function. The service handler will be called in case of new events generated by the service, which are defined as `ble_led_evt_type_t`.
- Finally, the application can update the characteristic value if the state of the LEDs changes through the `ble_led_state_characteristic_update` function.

The CO2 service provides the CO2 sensor data attached to the device. It includes two characteristics: the CO2 Level characteristic for getting the sensor data, and the CO2 Alert characteristic for setting the threshold to trigger the alert when the CO2 value exceeds that value. Listing 3 presents the interface of the CO2 service. To integrate it into the application, the aforementioned steps used with LED service should be followed.

```
#define BLE_CO2_SERVICE_DEF(_name)                                  \
static ble_co2_t _name;                                             \
NRF_SDH_BLE_OBSERVER(_name ## _obs,                                 \
                     BLE_CO2_BLE_OBSERVER_PRIO,                     \
                     ble_co2_on_ble_evt, &_name)

// Base UUID: B4330000-8F44-42AB-94B9-FE4C80B0F7DF
#define BLE_CO2_SERVICE_BASE_UUID {0xdf, 0xf7, 0xb0, 0x80, 0x4c, 0xfe, 0xb9,
0x94, 0xab, 0x42, 0x44, 0x8f, 0x00, 0x00, 0x33, 0xb4}

// Service and characteristics UUIDs
// CO2 Service:                        B4330001-8F44-42AB-94B9-FE4C80B0F7DF
//  CO2 Level Characteristic:          B4330002-8F44-42AB-94B9-FE4C80B0F7DF
//  C02 Alert threshold Characteristic: B4330003-8F44-42AB-94B9-FE4C80B0F7DF
#define BLE_CO2_SERVICE_UUID        0x0001
```

```
#define BLE_CO2_LEVEL_CHAR_UUID     0x0002
#define BLE_CO2_ALERT_CHAR_UUID     0x0003

// CO2 service event type
typedef enum
{
    BLE_CO2_EVT_NOTI_ENABLED,
    BLE_CO2_EVT_NOTI_DISABLED,
    BLE_CO2_EVT_ALERT_NEW_VALUE,
} ble_co2_evt_type_t;

// Function for initializing the CO2 service
uint32_t ble_co2_init(ble_co2_t *p_co2, ble_co2_init_t *p_co2_init);

// Function for handling the Application's BLE stack events
void ble_co2_on_ble_evt(ble_evt_t const *p_ble_evt, void *p_context);

// Function for updating the CO2 level characteristic
uint32_t ble_co2_level_update(ble_co2_t *p_co2, uint16_t *co2_value);
```

Listing 3. The CO2 service interface.

The application starts by initialising the necessary hardware, interrupt, BLE stacks, and other modules. Then, the device begins advertising and waiting for other devices to connect. Figure 10 illustrates the flow of the application.

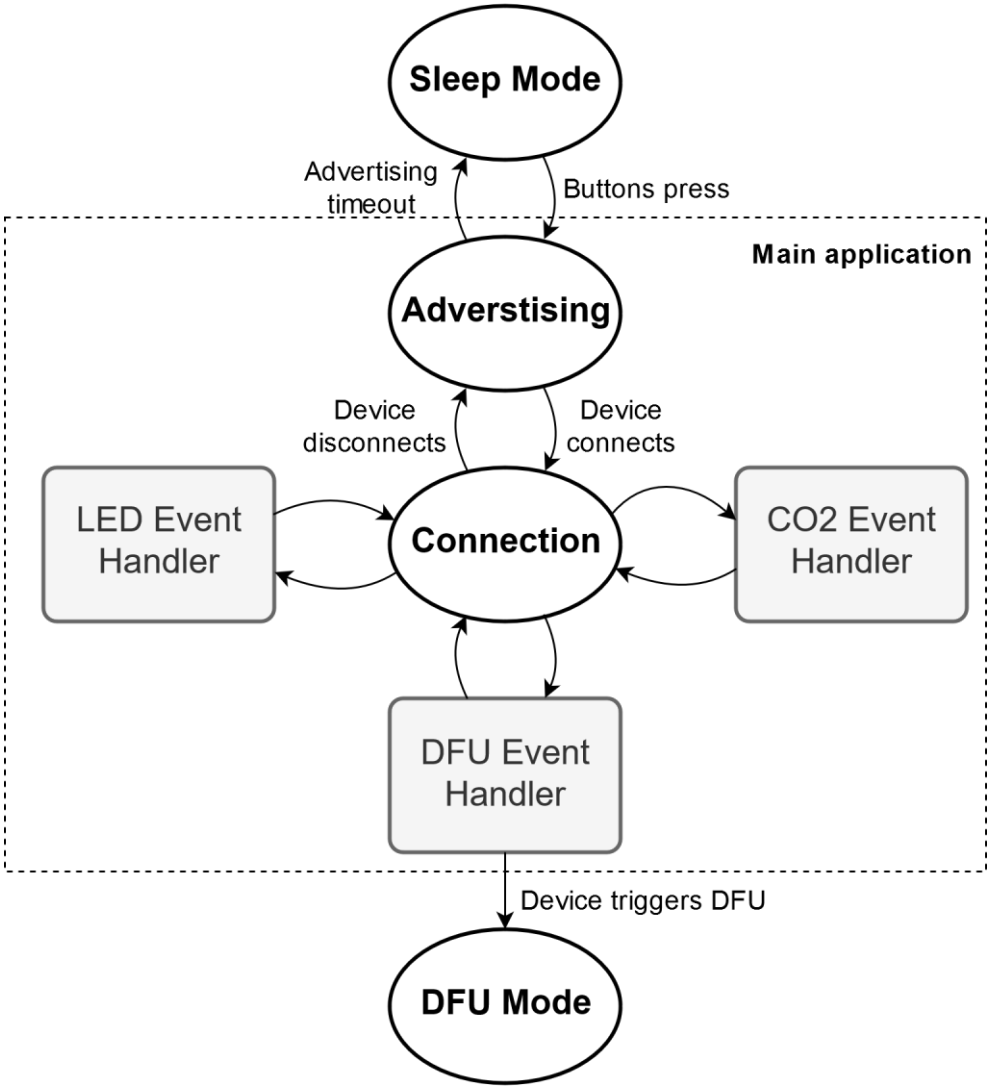

Figure 10. The application flow diagram.

The device advertises for 180 seconds. If there is no connection after that, the device puts itself to the sleep mode to converse the battery. The only way to wake it up is to press a button. If there is a connection, the device listens to the BLE event and reacts based on the predefined event handler. If the master device decides to initiate the DFU process, the application transfers its control to the bootloader and puts the sensor in the DFU mode.

5 Firmware testing process

This chapter describes the firmware testing process. It starts with presenting the test setups, then follows by showing the test results. In addition, the last section compares the Firmware OTA update to other relevant mechanisms.

5.1 Test setups

The preparation steps include building the necessary binary images, preparing the hardware. It also needs a device that supports BLE to act as the update controller. For demonstration purposes, a mobile phone with the nRF Connect app installed is used. Figure 11 illustrates the structure of the project.

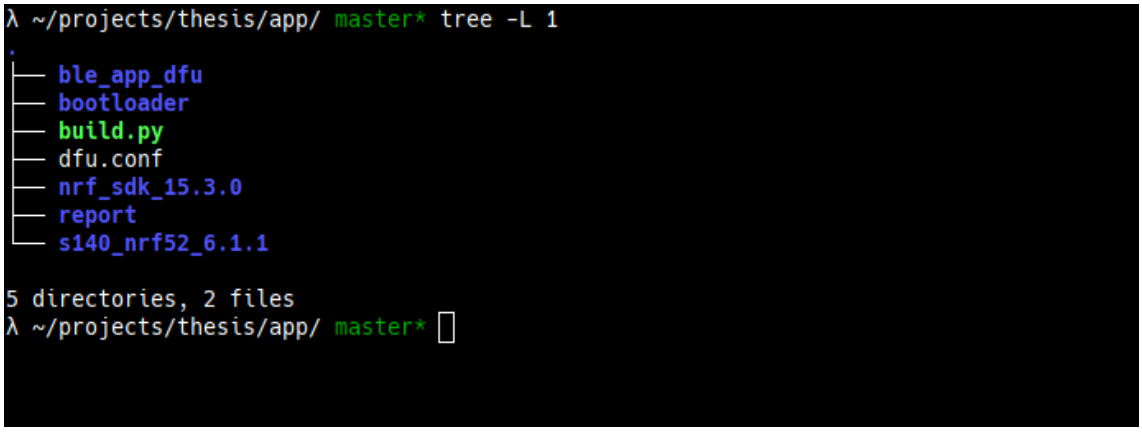

Figure 11. The structure of the project.

The application is organised in the `ble_app_dfu` folder while the bootloader is put in the `bootloader` folder. The SoftDevice and the nRF SDK are distributed the chip manufacturer and placed in `s140_nrf52_6.1.1` and `nrf_sdk_15.3.0` folder respectively. The project has a manifest file called `dfu.conf`. It contains metadata (such as version, hardware version, private key file) of the application, bootloader, and the SoftDevice. In order to simplify the building step, the project also has a script named `build.py` to walk into these directories and execute the necessary command recursively. It can be used to build the required binary images, generate bootloader settings, create firmware update package, merge binary files, and flash the firmware to target devices.

The hardware that is used is the development kit described above. The development board has four buttons that serve as the input to the device. It also connects to four external LEDs to indicate the current state of the device. Figure 12 shows a picture of the set up in detail.

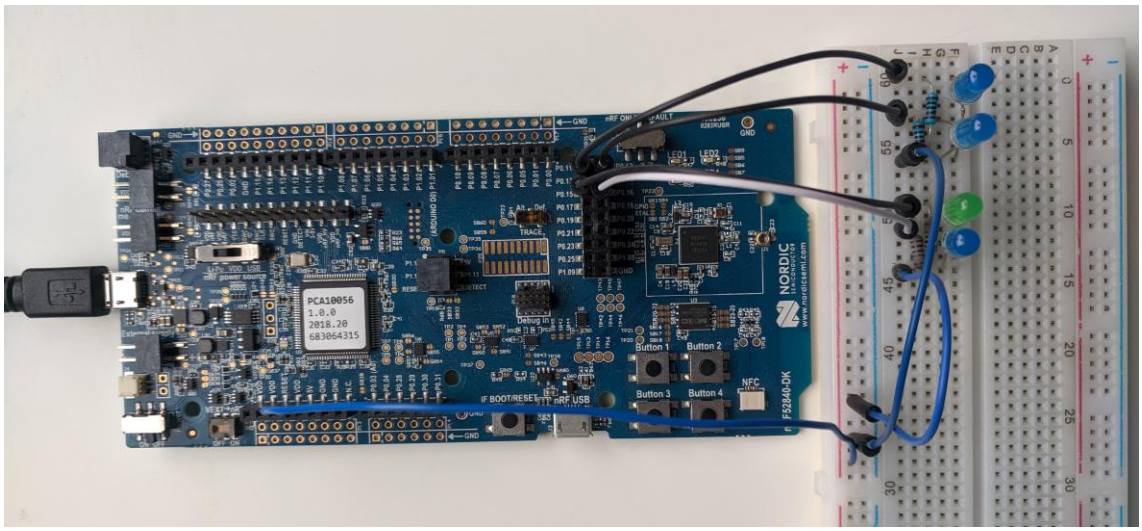

Figure 12. The hardware being used for testing.

The application is built split into two versions. The first version contains only the DFU service and the LED service. It then merges with the bootloader, the SoftDevice, and the bootloader settings into one binary image. This image is physically flashed to the device. This step simulates the initial firmware loading process in production before shipping the product to end-users. The second application version adds the CO2 service to the device. Instead of flashing the new firmware directly, the device was received by the OTA update. This step mimics the process of updating the device performed by end-users.

5.2 Test requirements

To evaluate the prototype implementation, there are many requirements that it has to meet. Since the SoftDevice is used as-is, the testing procedures only concern the application and the bootloader. Table 1 lists the mandatory tests that the application should pass.

	Description	**Test case**
BLE advertisement	The sensor node can advertise with correct parameters.	1.1

	Other devices can connect to the sensor node.	1.2
DFU service	Other devices can discover the service.	2.1
	Other devices can communicate with the Buttonless DFU characteristic.	2.2
	The Buttonless DFU characteristic can trigger the DFU process with correct commands from the controller device.	2.3
LED service	Other devices can discover the service.	3.1
	Other devices can communicate with the LED state characteristic.	3.2
	The characteristic reports the correct status of LED on the target device.	3.3
CO2 service	Other devices can discover the service.	4.1
	Other devices can communicate with the CO2 Level characteristic to retrieve CO2 data.	4.2
	The characteristic reports the correct status of CO2 measurement.	4.3
	Other devices can communicate with the CO2 Alert characteristic to set the new alert trigger value.	4.4

Table 1. Testing requirements of the main application.

The testing checklist for the bootloader is different from the application since it is modified from the nRF SDK. Hence, we are concerned only about the behaviour of the bootloader in three scenarios:

- No DFU in progress and a valid application exists
- No DFU in progress and no valid application exists
- DFU in progress

5.3 Test results

Each component in the system was tested independently while being developed. To obtain the final test results, the following steps were taken.

- (1) Building and merging three binary images (application, bootloader, SoftDevice)
- (2) Flashing it to the target device
- (3) Using the nRF Connect app to scan and connect to the sensor node
- (4) Testing the LED service
- (5) Performing firmware OTA update
- (6) Using the nRF Connect app to scan and connect to the sensor node, if the OTA update successfully finished

- (7) Testing the CO2 service

Figure 13,14,15 show the result in the controller device after carrying out steps from (1) to (4).

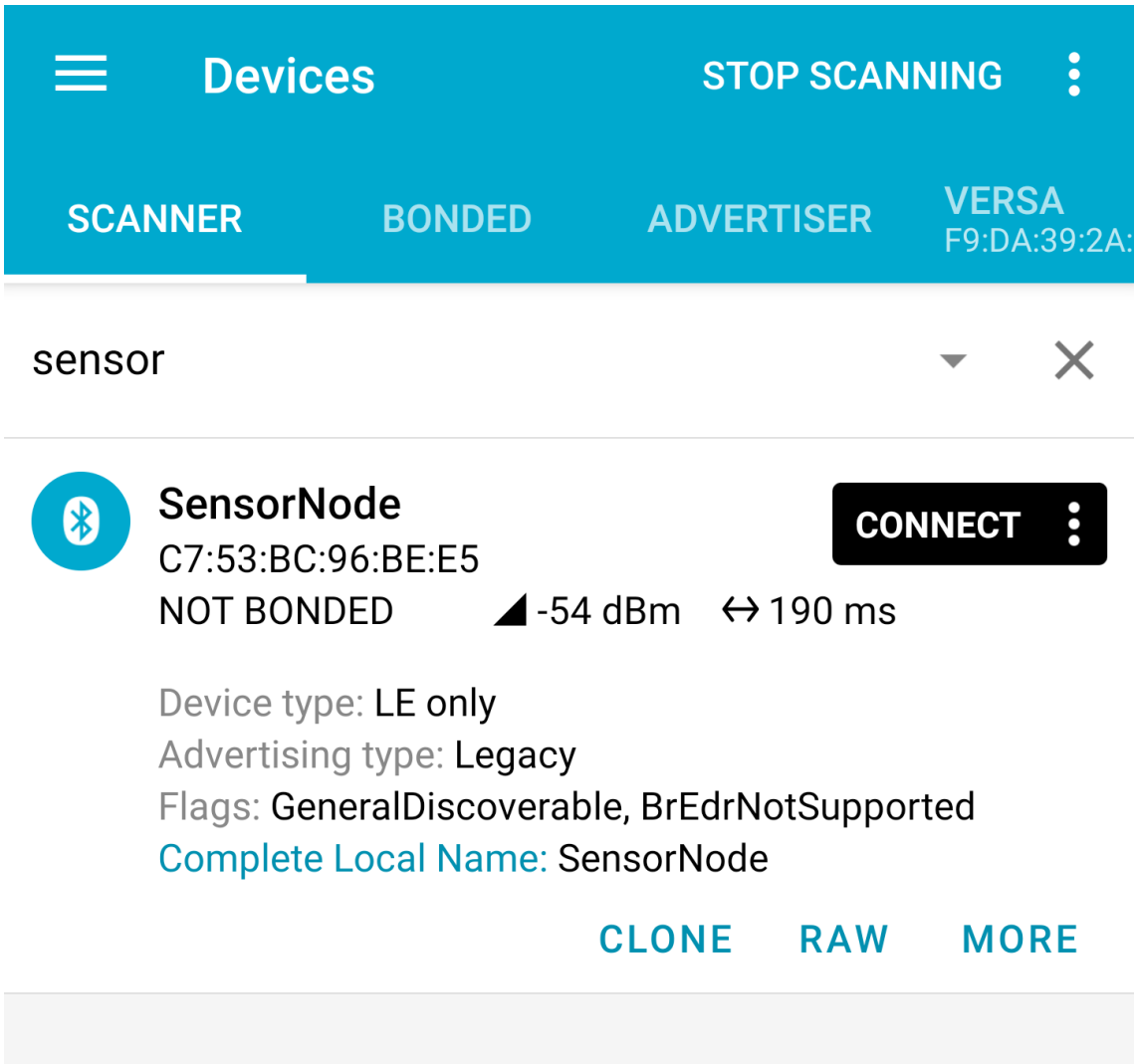

Figure 13. The target device is running firmware version 1.

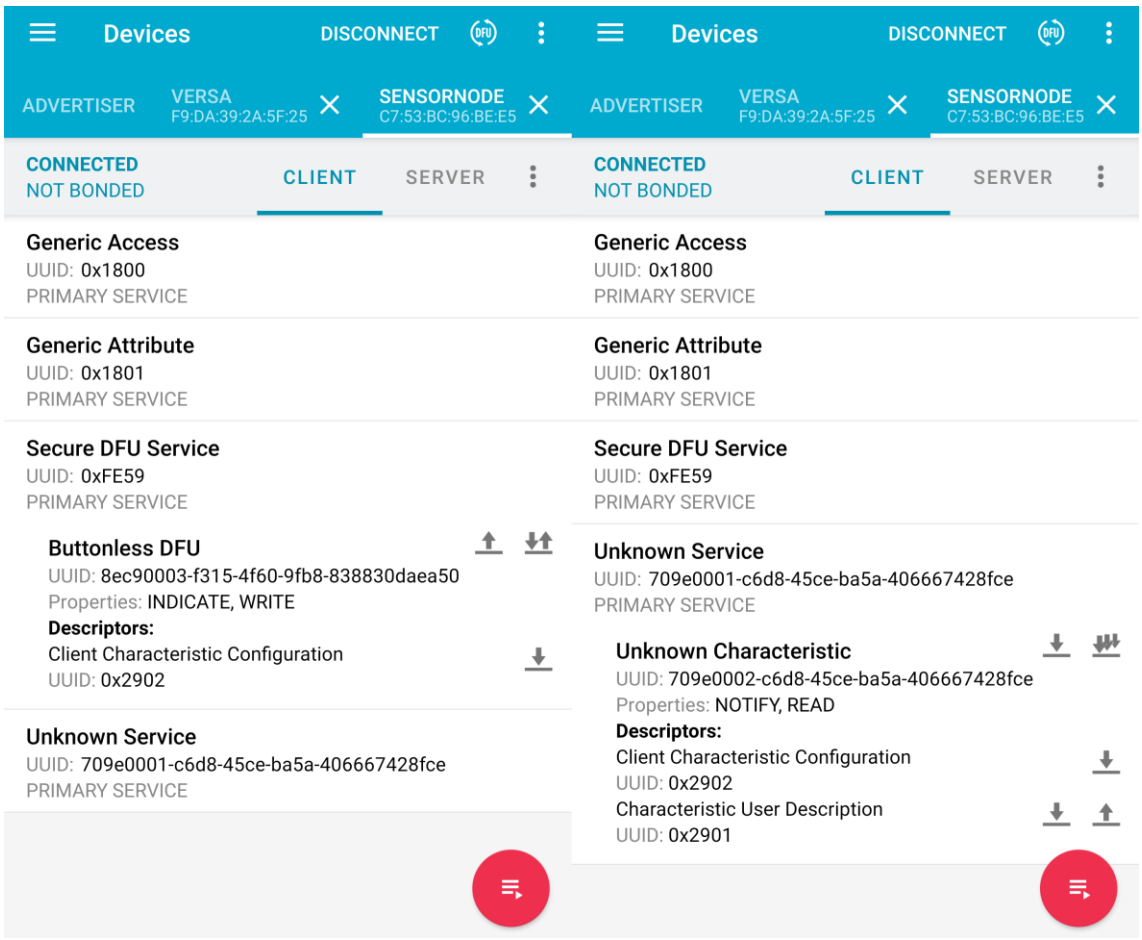

Figure 14. Available services in firmware version 1.

The sensor node is advertised as it should be. After connecting to it, the controller device can explore its services and characteristics, which include: Generic Access, Generic Attribute, Secure DFU Service with Buttonless DFU characteristic and LED Service (the Unknown Service with UUID 709E0001-C6D8-45CE-BA5A-406667428FCE) with LED State characteristic. Users are also able to query the LED and set up the notification. At this point, test case 1.1, 1.2, 2.1, 3.1, 3.2, 3.3 are passed.

Figure 15,16 are the result after performing steps (5), which requests to update the target device.

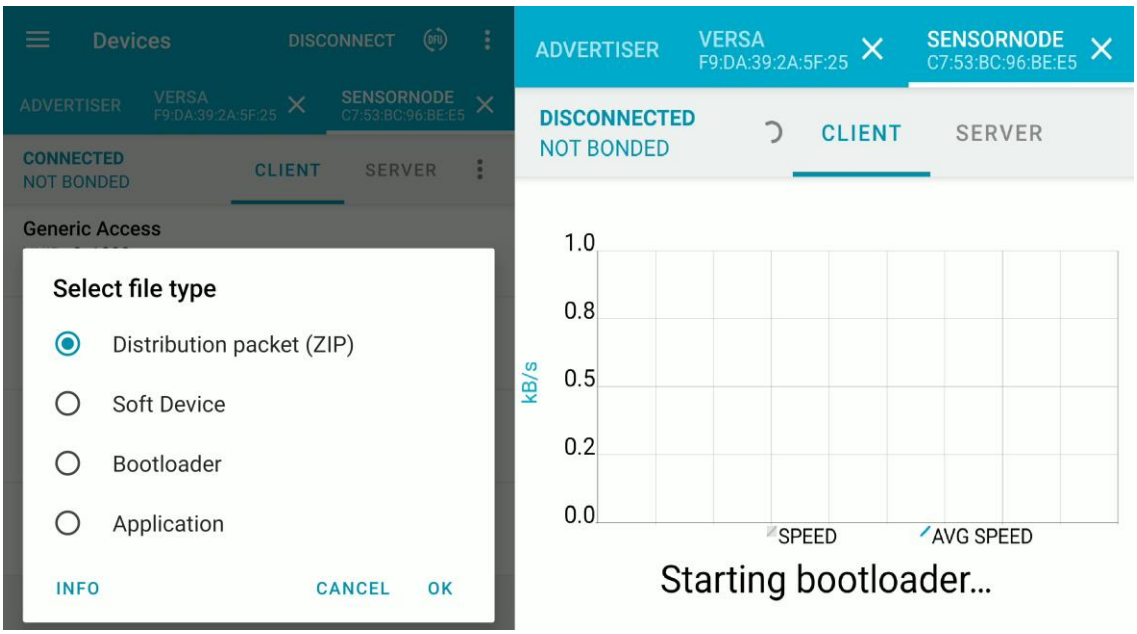

Figure 15. Starting the DFU process.

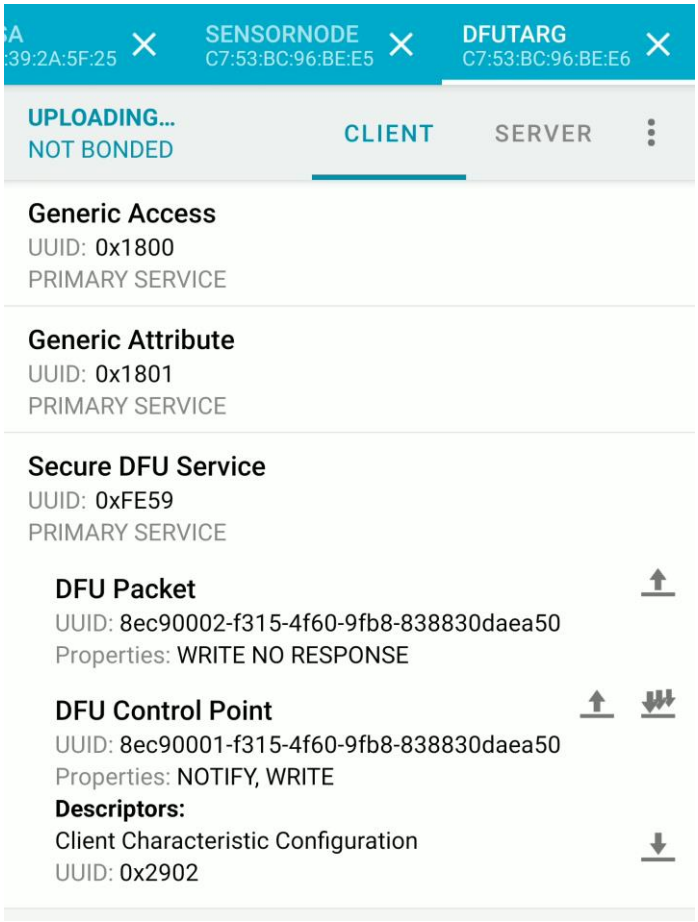

Figure 16. The BLE service for delivering the firmware image.

The target device is updated. A new GATT server is spawned (notice the MAC address of the DFUTARG is different from the MAC address of SensorNode) to handle the transmission of the update package. At this point, test case 2.2, 2.3 are passed.

Figure 15 is the result after conducting step (6) and (7).

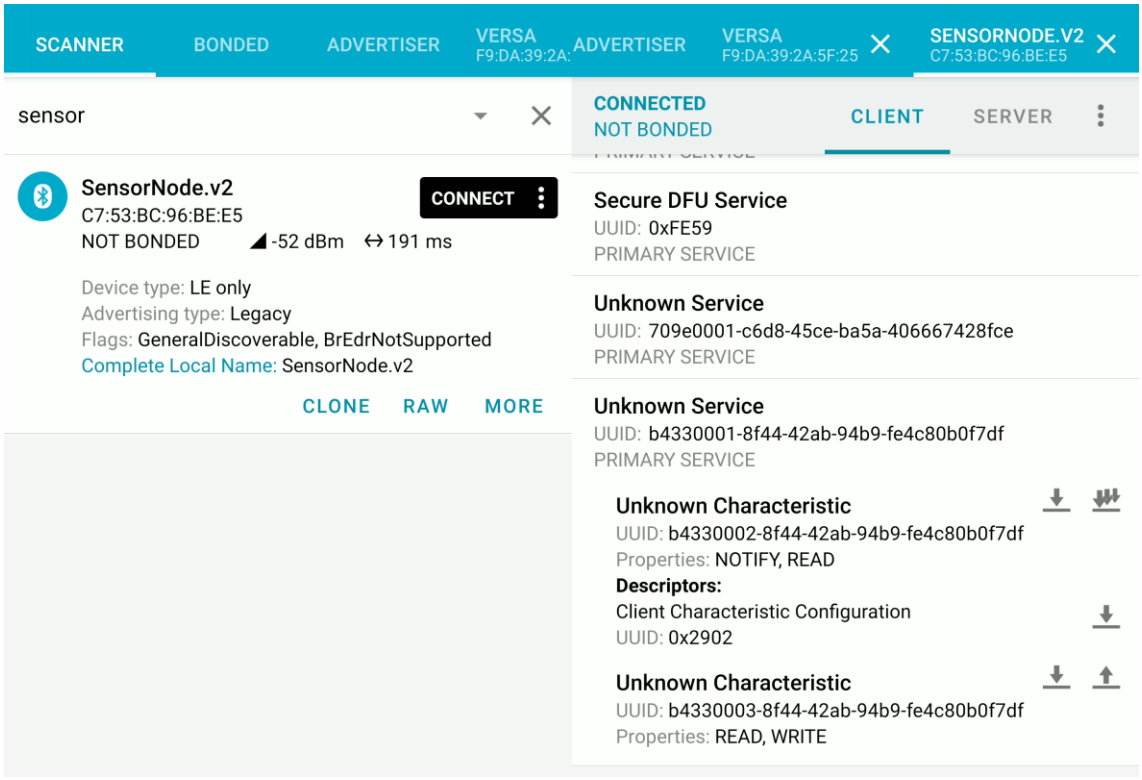

Figure 17. The target device is running firmware version 2.

The OTA update process was completed successfully. As can be seen from Figure 15, the device is running firmware version 2. Version 2 adds the CO2 service (the Unknown Service with UUID B4330001-8F44-42AB-94B9-FE4C80B0F7DF), allowing user to get the CO2 data (through the characteristic with UUID B4330002-8F44-42AB-94B9-FE4C80B0F7DF) and set the alert threshold (through the characteristic with UUID B4330003-8F44-42AB-94B9-FE4C80B0F7DF). At this point, test case 4.1, 4.2, 4.3, 4.4 are passed.

Given that the OTA process was completed successfully, and the sensor node can boot into the application, it implies that the bootloader behaves as expected.

6 Conclusions

OTA firmware update is becoming the mandatory feature that every IoT device should support. It not only provides the ability to improve the device itself continuously but also enhances the IoT system, since it is as strong as its weakest link. The thesis work presented the background on the topic and successfully implemented a prototype to demonstrate the feasibility of integrating OTA firmware update into the embedded application.

However, the implementation is heavily dependent on the example from the chip manufacturer. To generalise the solution, more work should be carried out by employing vendor-independent components such as bootloader, open-source DFU service over BLE. In addition, real sensors should be incorporated into the device.

References

1 Karen Rose, Scott Eldridge, and Lyman Chapin. The Internet of Things: An Overview. The Internet Society, 2015. URL: https://www.internetsociety.org/resources/doc/2015/iot-overview.

2 C. Bormann, M. Ersue, and A. Keranen. Terminology for constrained node networks," RFC 7228 (Informational). Internet Engineering Task Force, May 2014. URL: http://www.ietf.org/rfc/rfc7228.txt.

3 Elecia White. Making Embedded Systems: Design Patterns for Great Software. O'Reilly Media, 2011.

4 Nordic Thingy:52, IoT Sensor Kit [Online]. URL: https://www.nordicsemi.com/Software-and-Tools/Development-Kits/Nordic-Thingy-52, Accessed June 12, 2019.

5 Kevin Townsend, Carles Cufí, Akiba, and Robert Davidson. Getting Started with Bluetooth Low Energy: Tools and Techniques for Low-Power Networking. O'Reilly Media, 2014.

6 Bluetooth SIG. Bluetooth Core Specification v5.1. URL: https://www.bluetooth.com/specifications/adopted-specifications.

7 nRF52840 SoC [Online]. URL: https://www.nordicsemi.com/Products/Low-power-short-range-wireless/nRF52840, Accessed June 16, 2019.

8 Zephyr Project [Online]. URL: https://www.zephyrproject.org, Accessed June 16, 2019.

9 Benjamin Bucklin Brown. Over-the-Air (OTA) Updates in Embedded Microcontroller Applications: Design Trade-Offs and Lessons Learned. Analog Dialogue 52, November 11, 2018. URL: https://www.analog.com/media/en/analog-dialogue/volume-52/number-4/over-the-air-ota-updates-in-embedded-microcontroller-applications.pdf.

10 ZigBee Alliance. ZigBee Over-the-Air Upgrading Cluster5, Revision 236, Version 1.1. March 12, 2014. URL: https://www.zigbee.org/wp-content/uploads/2014/11/docs-09-5264-23-00zi-zigbee-ota-upgrade-cluster-specification.pdf.

11 Device Firmware Update process [Online]. URL: https://infocenter.nordicsemi.com/topic/com.nordic.infocenter.sdk5.v15.3.0/lib_bootloader_dfu_process.html, Accessed September 7, 2019.