

BLUETOOTH 5 & BLUETOOTH LOW ENERGY

A DEVELOPER'S GUIDE

*Exercises using Nordic nRF52840,
Thingy:52, nRF Sniffer, nRF Cloud
& Ellisys Bluetooth Tracker*

MOHAMMAD AFANEH

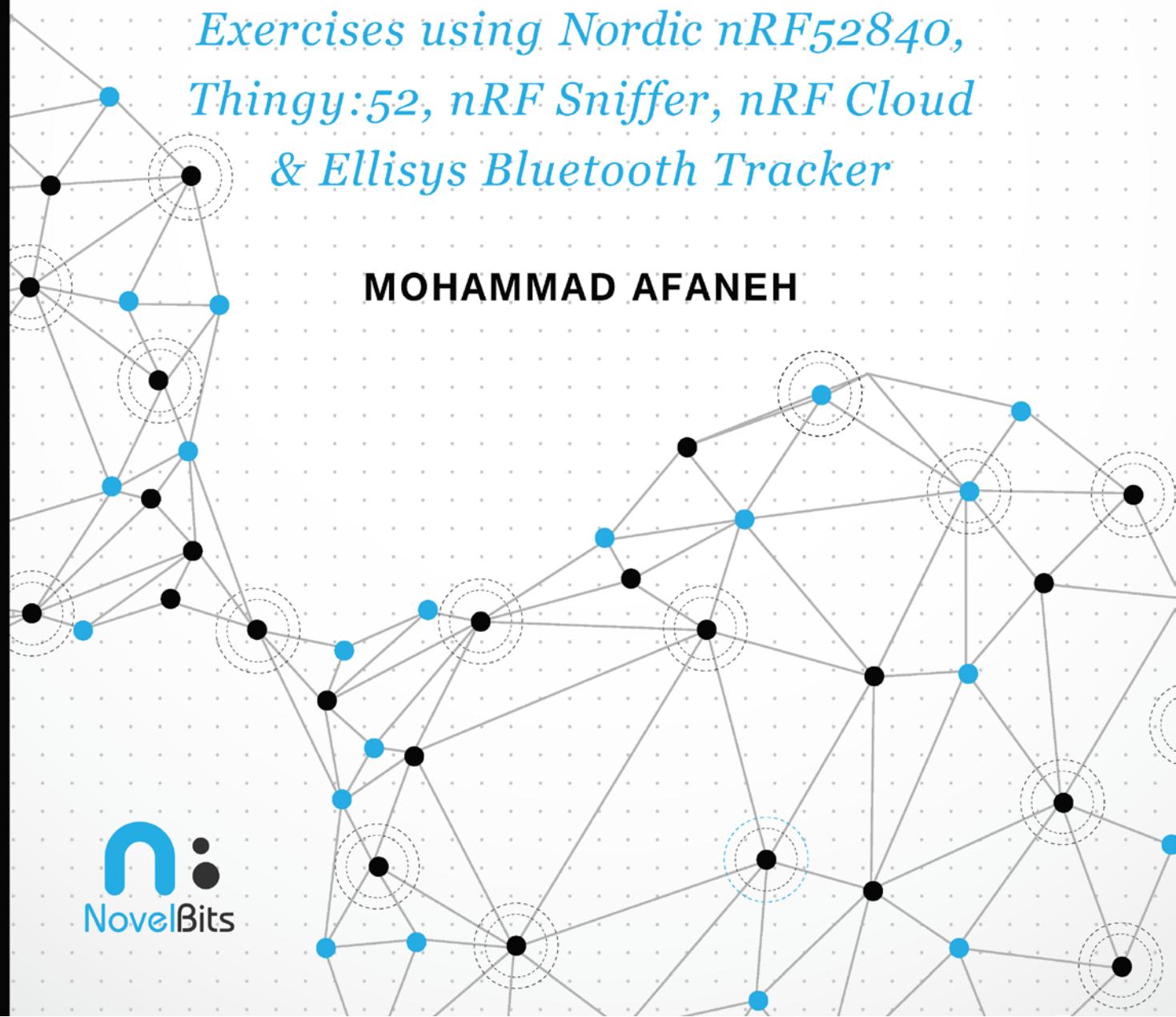


Table of Contents

Acknowledgments

Preface

Legal Disclaimer

Main Project

Basics of Bluetooth Low Energy

BLE Peripherals and Centrals

Advertising and Scanning

Connections

Services and Characteristics

GATT Design Guidelines

GAP Design Guidelines

GitHub Repository

Introduction to the nRF5 SDK

Development Environment Setup

Main Project Structure

nRF Development and Troubleshooting Tips

The “Hello World” Example

Bluetooth 5

Power Consumption

Security

Debugging and Testing BLE Applications

Reverse Engineering a Bluetooth Lightbulb

How to Choose a BLE Module/Chipset

Navigating the Bluetooth Specification Document

The Bluetooth Certification Process

Remote Control Source Code Walkthrough

Gateway Source Code Walkthrough

Using nRF Cloud as the Internet Gateway

Glossary

References and Resources

Acknowledgments

I dedicate this work to my mother Ameena, my wife Dana, and my two sons Bassam and Yaseen. Thank you for your endless love and support.

A special thank you to my best friend and wife Dana for being my rock, believing in me, and always inspiring me and encouraging me to move forward. Thank you to my sons for putting up with the many long workdays and worked weekends while remaining patient and always putting a smile on my face.

I'd also like to thank David Elvig for the many hours he spent in reviewing most of the content of the book. Last, but not least, thank you to all my family members, in-laws, friends, colleagues, and followers for your continuous support.

Preface

Why I Wrote This Book

My interest in Bluetooth Low Energy sparked in 2014. When I first started learning about BLE and how to develop applications on the embedded side, I spent hours, days, and weeks just researching and reading everything I can find on the topic. The learning curve was steep and going through the Bluetooth specification document was utterly frustrating!

Over the years, though, things started to make a lot more sense and I became much more comfortable with the technology. Looking back, I wished there were better books and resources that targeted developers with more practical exercises and examples.

It's probably true for many technologies, but I've found that many of the resources and books on BLE leave a huge gap when going from theory to practice. I would learn something interesting about BLE from the specification, but then find out that using it and implementing it for a given platform was very disconnected from what I read.

Finally, last year in July 2017, I decided to write a book on Bluetooth 5 and Bluetooth Low Energy to help other developers starting on the same journey I had traveled before. I hope you will find it to be a great resource that can make learning BLE more fun and engaging!

Who Is This Book For?

My goal with this book is to serve three main purposes:

- To help embedded developers learn Bluetooth Low Energy technology by guiding them through many exercises of implementing BLE applications on the Nordic nRF52 platform.
- To help mobile developers learn BLE and how it's implemented on the embedded side through the many exercises implemented for the Nordic nRF52 series platform.
- To help non-developers learn the basics of BLE. The book is very practical in nature and assumes some software development knowledge. However, the main chapters of the book focus on the technology and the protocol — the development exercises always come at the end of a chapter or in completely separate chapters themselves.

How To Read This Book

The book is best read in sequence starting from the beginning to the end.

Main Project Chapter

The book starts off with a chapter describing the Main Project referenced and used in exercises throughout the book. It represents a simple Home Automation project that we will be referenced and implemented throughout the chapters. This project requires a number of different hardware components that can be purchased by the reader. (*A list of these components can be found towards the end of this chapter.*)

Basics of Bluetooth Low Energy Chapters

The following **three** chapters go over the basics of Bluetooth Low Energy. They start with a high-level overview of the architecture of the technology, and then go deeper into the layers that are most relevant to a BLE application developer. These chapters are titled:

- Basics of Bluetooth Low Energy
- The Generic Access Profile (GAP)
- Advertising and Scanning
- Connections
- Services and Characteristics
- GATT Design Guidelines
- GAP Design Guidelines

Development Environment Chapters

Following the Basics of BLE chapters, we get into how to set up your development environment for the **Nordic Semiconductor nRF52 series** platform. We will also give an overview of the nRF52 platform, the nRF52 SDK as well as some “walkthroughs” of simple examples. These chapters are titled:

- The Companion GitHub Repository
- Introduction to the nRF52 Software Development Kit
- Development Environment Setup
- Main Project File Structure
- nRF Development and Troubleshooting Tips
- The “Hello World” Example

Advanced Topics Chapters

In these chapters, we will get into some more advanced aspects of Bluetooth Low Energy such as Bluetooth 5, Debugging, Security, Power Consumption, etc.

These chapters are titled:

- Bluetooth 5
- Power Consumption
- Security
- Debugging and Testing BLE Applications
- Reverse-Engineering a Bluetooth Lightbulb
- How to Choose a BLE Module/Chipset
- Navigating the Bluetooth Specification Document
- The Bluetooth Certification Process

Main Project Details Chapters

In the final chapters we go through and explain the source code used to implement the Main Project described in the first chapter of the book. It will go over the parts that are developed using the nRF52 series platform. These chapters are titled:

- The "Remote Control" Source Code Walkthrough
- The "Gateway" Source Code Walkthrough
- Using nRF Cloud as an Internet Gateway

Glossary and References

Finally, we will list a glossary of the most important terms used in the book. We will also list some references and useful resources that the reader can refer to for continued learning of Bluetooth Low Energy.

Hardware and Software Components

Throughout the book, we utilize a few software and hardware components that can make developing and testing BLE applications a much smoother process.

The following components are required/recommended to get the most benefit out of the practical portions of the book including exercise and the Main Project implementation:

- A computer running macOS, Windows, or Linux (required)
- An internet connection to download various software packages (required)
- Two [nRF52840 development kits](#) (required)
- A [Playbulb Candle](#) (highly recommended)

- A [Thingy:52 IoT sensor kit](#) (highly recommended)
- Another nRF52 or nRF51 development kit for the sniffer exercises (recommended)
- A USB hub to connect the various development kits (recommended)
- An [Ellisys Bluetooth Tracker](#) (optional)

About the Author

Mohammad Afaneh has been developing embedded software and firmware for over 12 years. He has worked at and consulted for multiple large companies including: Allegion, Motorola, Technicolor, Audiovox, and Denon & Marantz Group. Throughout his career, he has worked on multiple IoT (Internet of Things) products including: connected electronic door locks, satellite receivers, a wireless doorbell, and many other side projects.

In August 2015, he decided to leave his full-time job to start his own company [Novel Bits, LLC](#). At Novel Bits, he focuses on providing consulting services, on-site training, and online education, all primarily focused on Bluetooth Low Energy technology.

You can reach Mohammad directly at his email mohammad@novelbits.io, or by connecting with him on [LinkedIn](#).

Feedback

No work piece is perfect, so I would love to get your feedback on the contents of the book and how it can be improved to provide more value to you!

This book will continue to be updated, and I am committed to delivering more content to supplement it in the future. Best of all, these updates, revisions, and enhancements will all be provided to **you**, the reader, for free.

Feel free to connect with me with any questions, comments or feedback:

- Email: mohammad@novelbits.io
- LinkedIn: [Mohammad's Profile](#)
- Twitter: [Mohammad's Twitter Handle](#)

Legal Disclaimer

All rights reserved. This publication is protected by copyright, and permission must be obtained from the author prior to any reproductions, transmissions, copying, and resale or likewise. To obtain permission to use major portions of this book, please contact the author via e-mail at mohammad@novelbits.io. The content and source code within this e-book is meant for educational purposes only. Any other purpose or use is at the discretion of the user.

The material is provided "AS IS," without a warranty of any kind, express or implied, including but not limited to the warranties of fitness for a particular purpose and merchantability. In no event shall the author be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the publication, source code, or the use or other dealings with the materials within this book.

Source Code Licenses

All source code provided with the book (in the GitHub repository as well as in the text) is either licensed with the [MIT license](#) or a modified version of source code provided by [Nordic Semiconductor](#).

Nordic Semiconductor's license information states:

```
/**  
 * Copyright (c) 2014 – 2017, Nordic Semiconductor ASA  
 *  
 * All rights reserved.  
 *  
 * Redistribution and use in source and binary forms, with or without modification,  
 * are permitted provided that the following conditions are met:  
 *  
 * 1. Redistributions of source code must retain the above copyright notice, this  
 *    list of conditions and the following disclaimer.  
 *  
 * 2. Redistributions in binary form, except as embedded into a Nordic  
 *    Semiconductor ASA integrated circuit in a product or a software update for  
 *    such product, must reproduce the above copyright notice, this list of  
 *    conditions and the following disclaimer in the documentation and/or other  
 *    materials provided with the distribution.  
 *  
 * 3. Neither the name of Nordic Semiconductor ASA nor the names of its  
 *    contributors may be used to endorse or promote products derived from this  
 *    software without specific prior written permission.  
 *  
 * 4. This software, with or without modification, must only be used with a  
 *    Nordic Semiconductor ASA integrated circuit.  
 *  
 * 5. Any software provided in binary form under this license must not be reverse  
 *    engineered, decompiled, modified and/or disassembled.
```

```
*  
* THIS SOFTWARE IS PROVIDED BY NORDIC SEMICONDUCTOR ASA "AS IS" AND ANY EXPRESS  
* OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES  
* OF MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE ARE  
* DISCLAIMED. IN NO EVENT SHALL NORDIC SEMICONDUCTOR ASA OR CONTRIBUTORS BE  
* LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR  
* CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE  
* GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)  
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT  
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT  
* OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.  
*  
*/
```

Main Project

Introduction

Rather than going through the different aspects of BLE and not understanding how they relate to real-world applications, I thought it'd be great to start with an overview of the project that will be used referenced throughout the book.

In my search for a project that would be suitable as the book's main exercise, I wanted to choose one that any reader would relate to. Because of this, I thought: there's no better project than a **home automation** project. We all live in a home in one way or another — whether it's an apartment, house, condo (or even a trailer)!

Home automation is one of the major consumer applications for the Internet of Things (IoT) and Bluetooth Low Energy (especially with the recent release of Bluetooth mesh in July 2017). We've all seen and read about those magical scenarios where a user walks into the home and magical things happen with minimal or no user interaction — almost like our homes can read our minds! It may sound a bit far fetched, but realistically, in the future with all the advancements in technology these scenarios may actually become feasible.

For the sake of simplicity and being to the point, the project we'll build in this book will be a simple, yet powerful, one that showcases the possibilities of Bluetooth Low Energy and how it relates to you, the reader, in your own personal environment as well.

Project and System Components

Note: To get the most benefit from the book — especially in terms of practical knowledge and going from theory to implementation — make sure you have access to all the hardware components listed for the main project.

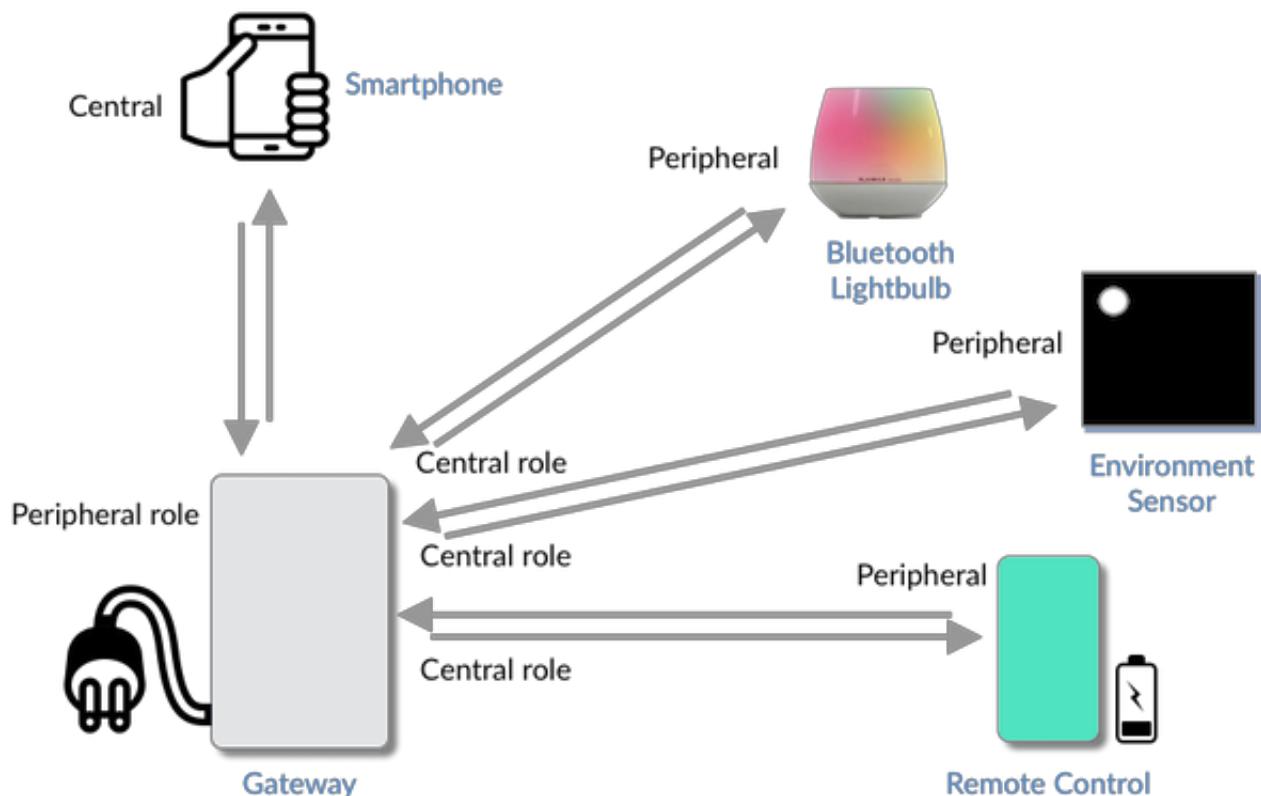


Figure 1: Main project system diagram

The components, hardware, and environment required for the project include:

- A computer for development (all three operating systems are supported: macOS, Windows, and Linux).
- A mobile phone for connecting to the [Nordic nRF Cloud](#) (to run the [iOS app](#) or [Android app](#)).

An alternative is to use a [Raspberry Pi 3](#) for connecting to nRF Cloud — for instructions refer to [this article](#) and [this one](#).

- **Two nRF52 Development Boards**

We'll be using two [nRF52840 development kits](#). Alternatively, the [nRF52832 development kit](#) could be used but will not be able to utilize the Bluetooth 5 long range feature (LE Coded PHY) in any examples that utilize this feature. Additionally, the project files included alongside the book only support the nRF52840 — they would need to be modified to build for the nRF52832 instead.

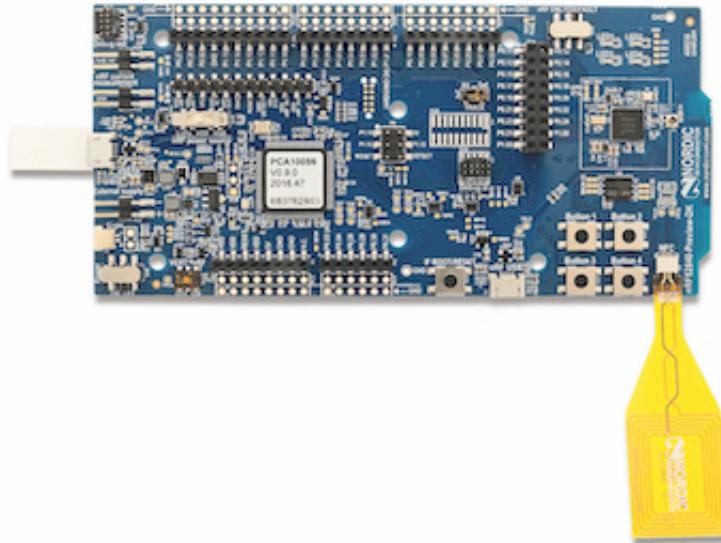


Figure 2: nRF52840 development kit

- **Thingy:52 IoT Sensor Kit**

From Nordic's product page for the Thingy:52:

The Nordic Thingy:52® is a compact, power-optimized, multi-sensor development kit. It is an easy-to-use development platform, designed to help you build IoT prototypes and demos, without the need to build hardware or write firmware.

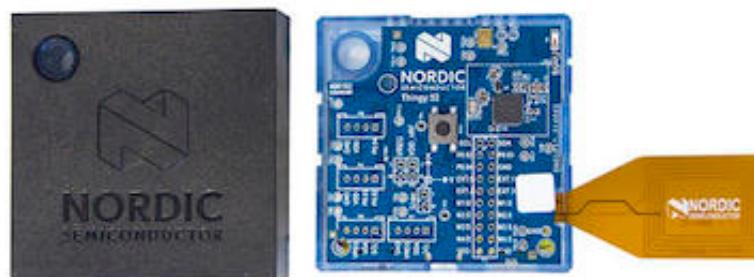


Figure 3: Nordic Thingy:52 sensor kit

- **Playbulb Candle**

Any Bluetooth lightbulb may be used. However, the steps for reverse-engineering the protocol used by the device will probably differ from the exercises provided in the book.



Figure 4: Playbulb candle Bluetooth light bulb

- A USB power plug (used to power one of the nRF52 development kits — the Gateway device), or simply connected to a PC
- A coin cell battery (CR2032)
- An Internet connection for connecting to the nRF Cloud service
- An [nRF Cloud](#) account

User Scenarios

When you start work on a project — whether it is a commercial or a simple hobby project — it is good practice to describe it from the user's perspective. This allows you to put yourself in their shoes, think of the system use cases at a high-level, and achieve the best user experience.

Let's go ahead and describe the main user scenarios for our home automation system:

1. The homeowner can use a remote control device to turn on/off the Playbulb candlelight.

Technical details: The remote device is one of the nRF52 development kits — the one powered by a coin cell

battery.

2. The homeowner can monitor changes in the temperature in the area where the environment sensor is placed — via connecting to the Gateway (directly from a smartphone or via the nRF Cloud service).

In my testing and development, I used the garage as the environment for placing the sensor, so you may come across the word "garage" in the terminology. This is simply an example and can obviously be changed to anything else. In the Main Project source code in the companion GitHub repository (refer to the chapter titled "GitHub Repository"),

Technical details: The Gateway device will have notifications turned on for the temperature characteristic on the Thingy:52 device that is placed in an area of interest.

3. The homeowner can monitor changes in the humidity in the area where the environment sensor is placed — via connecting to the Gateway directly (from a BLE Central, or via the Cloud service).

Technical detail: The Gateway device will have notifications turned on for the humidity characteristic on the Thingy:52 device that is placed in an area of interest.

4. The homeowner is notified of the battery levels of the Remote Control, Playbulb candlelight, and Thingy:52.

Technical detail: The Gateway device will have notifications enabled on the battery level characteristics of the Remote Control, Playbulb candlelight, and the Thingy:52.

Hardware Components

- **Gateway device (*implemented on the nRF52840 development kit*)**
 - Simultaneous central and peripheral roles.
 - Monitors the devices in the home and receives status updates from the Remote Control, Thingy:52, and Playbulb candlelight.
 - Allows data to be sent up to the nRF Cloud service.
 - Stationary.
 - Live power (via the micro USB connection on the development kit).
- **Remote Control device (*implemented on the nRF52840 development kit*)**
 - Peripheral role only.
 - Used as a remote control for the Playbulb Candle light.
 - Coin cell battery (CR2032).
 - Non-stationary.
- **Playbulb light (*off-the-shelf device*)**
 - Controlled from the Remote Control via the Gateway.
 - Light status changes reported to the Gateway.
 - Battery level reported to the Gateway.

- **Nordic Thingy:52 (*off-the-shelf device*)**

Used as an environment sensor to report:

- Temperature reading.
- Humidity reading.
- Battery level.

- **Smartphone (*connected to the nRF Cloud service*)**

◦ nRF Connect Gateway application on a mobile phone that relays data up to the nRF Cloud service.

Summary

In this chapter, we went over the project that will be used as the main exercise for the book. This is simply a starting point, though you'll have learned a great deal about the features of Bluetooth Low Energy. You can then easily expand the functionality and features to suit your need.

Basics of Bluetooth Low Energy

What is Bluetooth Low Energy?

Bluetooth started as a short-distance cable-replacement technology to replace wires in devices such as a mouse, a keyboard, or a PC. If you own a modern car or a smartphone, chances are you've used Bluetooth at least once in your life. It's everywhere: in speakers, wireless headphones, cars, wearables, medical devices, and even [flip-flops!](#)

The first official version of Bluetooth was released by Ericsson in 1994. It was named after King Harald "Bluetooth" Gormsson of Denmark who helped unify warring factions in the 10th century CE.

There are two types of Bluetooth devices: one is referred to as **Bluetooth Classic (BR/EDR)**, used in wireless speakers, car infotainment systems, and headsets, and the other is **Bluetooth Low Energy (BLE)**. BLE, introduced in Bluetooth version 4.0, is more prominent in applications where power consumption is crucial (such as battery-powered devices) and where small amounts of data are transferred infrequently (such as in sensor applications).

These two types of Bluetooth devices are incompatible with each other even though they share the same brand and even specification document. A Bluetooth Classic device cannot communicate (directly) with a BLE device. This is why some devices such as smartphones have chosen to incorporate both types of Bluetooth (also referred to as **Dual Mode Bluetooth devices**), allowing them to communicate with both types of devices.

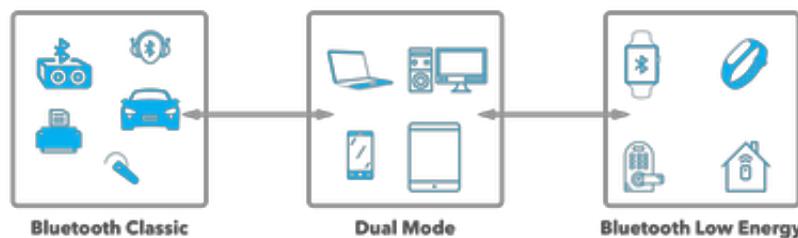


Figure 5: Types of Bluetooth devices

Here are a few important notes about BLE:

- The official Bluetooth specification document combines both types of Bluetooth (Bluetooth Classic and BLE), sometimes making it difficult to locate BLE-specific specifications.
- BLE was introduced in the 4.0 version of the Bluetooth specification, released in 2010.
- BLE is sometimes referred to as Bluetooth Smart or BTLE, and sometimes mistaken as Bluetooth 4.0 (since this version really included both types of Bluetooth).
- Both Bluetooth Classic and BLE operate in the same frequency spectrum (the 2.4 GHz Industrial, Scientific, and Medical (ISM) band).

Since many **Internet of Things (IoT)** systems involve small devices and sensors, BLE has become the more common

protocol of the two (versus Bluetooth Classic) in IoT. In December 2016, the Bluetooth Special Interest Group (SIG), the governing body behind the Bluetooth standard, released Bluetooth version 5.0 (for marketing simplicity, the point number is removed and the official name is Bluetooth 5). A majority of the enhancements and features introduced in this version focused on BLE, not Bluetooth Classic.

You may have also heard of another term related to Bluetooth: **Bluetooth mesh**. Bluetooth mesh was released in July 2017. It builds on top of BLE and it requires a complete BLE stack (a software that acts as an interface for another piece of software or hardware) to work, but it's not part of the core Bluetooth specification.

To summarize, here's a figure showing the progression of BLE over the years:

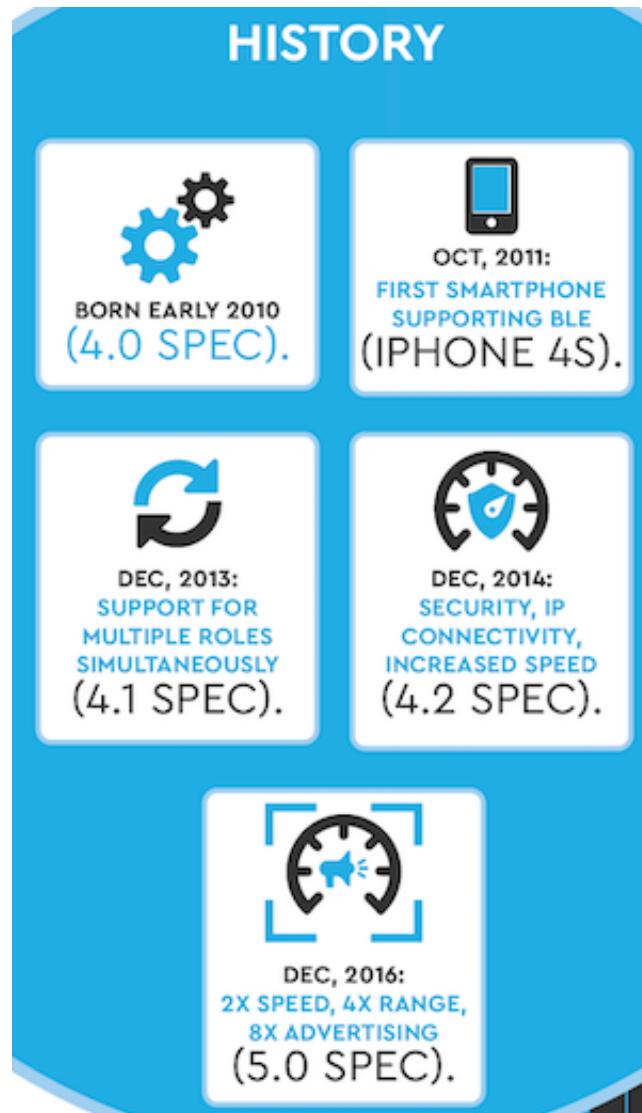


Figure 6: History of BLE

Technical Facts About BLE

Some of the most important technical facts about BLE include:

- The frequency spectrum occupied is **2.400 - 2.4835 GHz**.
- The frequency spectrum is segmented into **40 "2 MHz"-wide channels**.
- The maximum data rate supported by the radio (introduced in Bluetooth version 5) is **2 Mbps**.
- The range varies significantly depending on the environment surrounding the communicating BLE devices as well as the mode used (for example, in long-range mode, the range will be significantly longer than in the 2M/high-speed mode). A typical range is **10-30 meters (30-100 feet)**.
- Power consumption also varies widely. It depends on the implementation of the application, the different BLE parameters, and the chipset used. The peak current consumption of a BLE chipset during radio transmission is typically **under 15 mA**.
- Security is **optional** in BLE communication, and it is up to the device and applications developers to implement it. That said, though, there are also varying levels of security that can be implemented.
- For all encryption operations, BLE uses **AES CCM with a 128-bit key**.
- BLE is designed for **low-bandwidth data transfer** applications. Implementing BLE for high-bandwidth applications will significantly compromise the low power consumption promise. So, minimizing radio usage as much as possible achieves the optimal power consumption.
- **Bluetooth versions** are backwards compatible with each other. However, the communication may be limited to the features of the older version of the two communicating devices.

For example, a Bluetooth 5 BLE device can communicate with a Bluetooth 4.1 BLE device, but 5-specific features won't be supported. On the other hand, Connections, Primary Advertisements, discovering Services, discovering Characteristics, and reading/writing to these Characteristics are all possible between two BLE devices regardless of their supported Bluetooth version (since they were supported by the initial version of BLE).

Bluetooth Classic vs. BLE

It's important to note that there's a big difference between Bluetooth Classic and Bluetooth Low Energy in terms of technical specification, implementation, and the types of applications to which they're each suited. This is in addition to the fact that they are **incompatible** with each other.

Some of the notable differences are summarized in the following table:

Bluetooth Classic	BLE
Used for streaming applications such as audio streaming, file transfers, and headsets	Used for sensor data, control of devices, and low-bandwidth applications
Not optimized for low power, but has a higher data rate (3Mbps maximum compared to 2Mbps for BLE)	Meant for low power, low duty data cycles
Operates over 79 RF (radio frequency) channels	Operates over 40 RF channels.
Discovery occurs on 32 channels	Discovery occurs on 3 channels, leading to quicker discovery and connections than Bluetooth Classic

Table 1: Bluetooth Classic vs. BLE

BLE has gone through some major revisions and changes in the short time since its official release in 2010, with the most recent major update being Bluetooth 5 released in December 2016. Bluetooth 5 introduced many important upgrades to the Bluetooth specification, most of which were focused on BLE. Some of the most important enhancements include twice the speed, four times the range, and eight times the advertising data capacity.

Advantages and Limitations of BLE

Every technology has its limitations, and BLE is no exception. As we mentioned earlier, BLE is most suitable for applications with relatively short range and infrequent low-bandwidth data transfers.

Limitations of BLE

Data Throughput

The data throughput of BLE is limited by the physical radio data rate, which is the rate at which the radio transmits data. This rate depends on the Bluetooth version used. For Bluetooth 4.2 and earlier, the rate is fixed at 1 Mbps. For Bluetooth 5 and later, however, the rate varies depending on the mode and **PHY** (discussed later in the Physical Layer section) being used. The rate can be at 1 Mbps like earlier versions, or 2 Mbps when utilizing the high-speed feature. When utilizing the long-range feature, the rate drops to either 500 or 125 Kbps. We'll discuss each of these in more

detail in the section on Bluetooth 5.

At the application layer and for the end-user, the data rate is much lower than the radio data rate due to the following factors:

- **Gaps in between packets:** The Bluetooth specification defines a gap of 150 microseconds between packets being transmitted as a requirement for adhering to the specification. This gap is time lost with no data being exchanged between two devices.
- **Packet overhead:** All packets include header information and data handled at levels lower than the application level, which count towards the data being transmitted but are not part of the data utilized by your application.
- **Slave data packets requirement:** The requirement to send back data packets from the slave, even when no data needs to be sent back and empty packets are sent.
- **Retransmission of data packets:** In the case of packet loss or interference from devices in the surrounding environment, the lost or corrupted data packets get resent by the sender.

Range

BLE was designed for short range applications and hence its range of operation is limited. There are a few factors that limit the range of BLE including:

- It operates in the 2.4 GHz ISM spectrum which is greatly affected by obstacles that exist all around us such as metal objects, walls, and water (especially human bodies).
- Performance and design of the antenna of the BLE device.
- Physical enclosure of the device which affects the antenna performance, especially if it's an internal antenna.
- Device orientation, which effectively relates to the positioning of the antenna (e.g., in smartphones).

Gateway Requirement for Internet Connectivity

In order to transfer data from a BLE-only device to the Internet, another BLE device that has an IP connection is needed to receive this data and then, in turn, relay it to another IP device (or to the internet).

Advantages of BLE

Even with the previously mentioned limitations of BLE, it has presented some significant advantages and benefits over other similar technologies in the IoT space. Some of these advantages include:

- **Lower power consumption**
Even when compared to other low-power technologies, BLE achieves a lower power consumption than its competitors. It's optimized, and less power consumed, by turning the radio off as much as possible, in addition to sending small amounts of data at low transfer speeds.
- **No cost to access the official specification documents**
With most other wireless protocols and technologies, you would have to become a member of the official group or

consortium for that standard in order to access the specification. Becoming a member of those groups can cost a significant amount (up to thousands of dollars per year). With BLE, the major version (4.0, 4.1, 4.2, 5) specification documents are available to download from the Bluetooth website for free.

- **Lower cost of modules and chipsets** when compared to other similar technologies.
- Last but not least, **its existence in most smartphones in the market**. This is probably the biggest advantage BLE has over its competitors such as ZigBee, Z-Wave, and Thread.

Applications Most Suitable for BLE

Based on the limitations and benefits we mentioned earlier, there are a number of use cases where BLE makes the most sense:

- **Low-bandwidth data**

For cases where a device transfers small amounts of data representing sensor data or for controlling actuators, BLE has proven to be a suitable wireless protocol to utilize.

- **Device Configuration**

Even in cases where BLE doesn't satisfy the main requirements of a system, it can still be used as a secondary interface to configure a device before the main wireless connection is established.

For example, some WiFi-enabled devices are adding BLE as a means to configure and establish the WiFi connection of the device instead of using a technology such as WiFi direct (a technology that allows two WiFi devices to connect directly without going through a WiFi router. You can Learn more about it at its Wikipedia page [here](#)).

- **Using a smartphone as an interface**

Small, low-power devices usually don't have large screens and are only capable of displaying limited amounts of data to the end user. Due to the proliferation of smartphones nowadays, BLE can be utilized to offer an alternate, much richer user interface to these small devices (even if just for this sole purpose). Another by-product benefit of using a smartphone is that the data can be relayed up to the cloud.

- **Personal and wearable devices**

For use cases where a device is portable and can be located in areas where no other persistent wireless connections exist (such as WiFi), BLE can be used (since it's a direct peer-to-peer connection).

- **Broadcast-only devices**

You've probably heard of, and maybe seen, **Beacon** devices before. These devices have one simple task: to broadcast data so other devices may discover them and read their data. There are other technologies that have been used for this kind of application. However, BLE is becoming more and more popular because most people carry smartphones which already support BLE out-of-the-box.

These are all great use cases that could benefit from utilizing BLE. On the other hand, use cases that are not (generally) suitable for BLE include:

- Video streaming.
- High-quality audio streaming.
- Large data transfers for prolonged periods of time (if battery consumption is a concern).

Architecture of BLE

The following figure shows the different layers within the architecture of BLE. The three main blocks in the architecture of a BLE device are: the **application**, the **host**, and the **controller**.

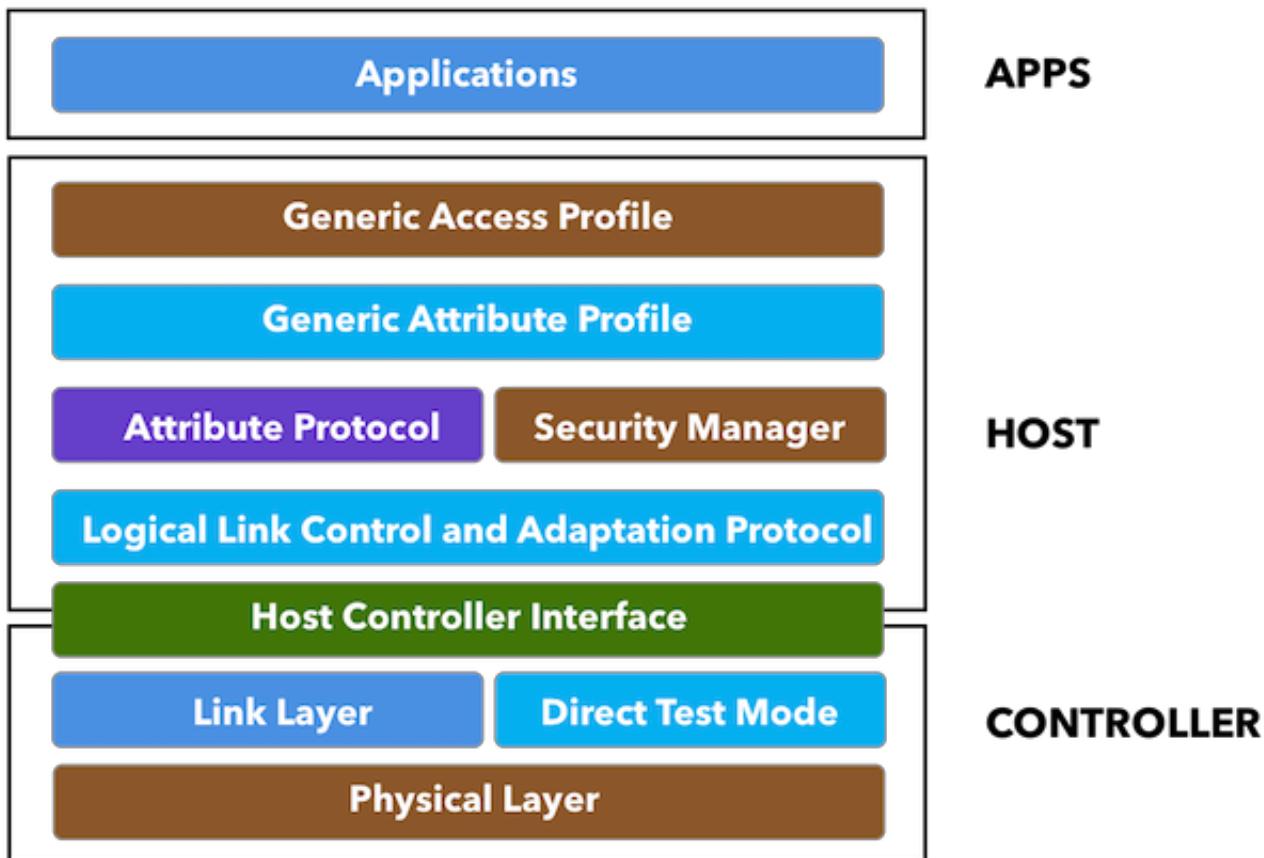


Figure 7: Architecture of BLE

In this book, we'll focus on the upper level layers of the architecture, while briefly covering the lower levels of the architecture. We'll go over each of the lower-level layers in this chapter and then look at each of the upper layers (the **Generic Access Profile**, the **Generic Attribute Profile**, the **Attribute Protocol**, and the **Security Manager**) each in their own chapter.

Application

The **application** layer is use-case dependent and refers to the implementation on top of the Generic Access Profile and Generic Attribute Profile — it's how your application handles data received from and sent to other devices and the logic behind it.

This portion is the code that you would write for your specific BLE application and is generally not part of the BLE stack for the platform which you develop. This part will not be covered in the book, since it depends on the specifics of your application and use case.

Host

The **host** contains the following layers:

- Generic Access Profile (GAP)
- Generic Attribute Profile (GATT)
- Attribute Protocol (ATT)
- Security Manager (SM)
- Logical Link Control and Adaptation Protocol (L2CAP)
- Host Controller Interface (HCI) — Host side

Controller

The **controller** contains the following layers:

- Physical Layer (PHY)
- Link Layer
- Direct Test Mode
- Host Controller Interface (HCI) — Controller side

Layers of the BLE Architecture

Physical Layer (PHY)

The **physical layer (PHY)** refers to the radio hardware used for communication and for modulating/de-modulating the data. BLE operates in the ISM band (2.4 GHz spectrum), which is segmented into 40 RF channels, each separated by 2 MHz (center-to-center), as shown in the following figure:

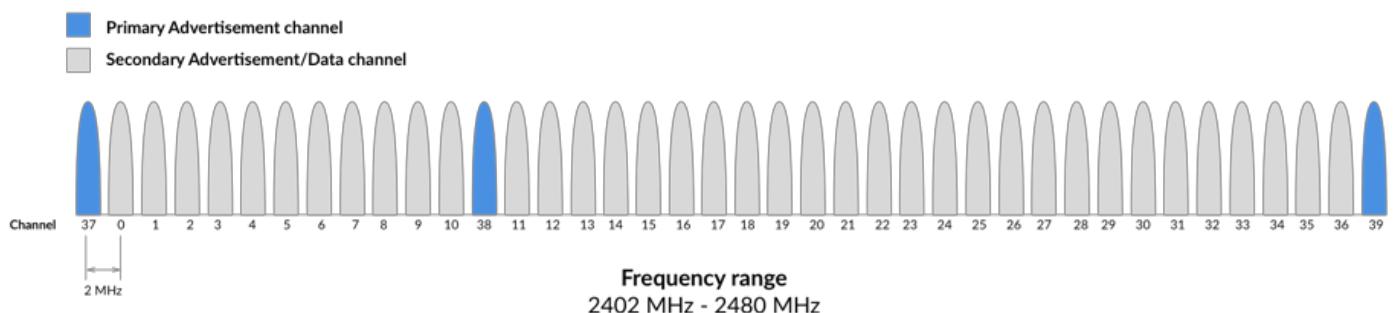


Figure 8: Frequency spectrum and RF channels in BLE

Three of these channels are called the **Primary Advertising Channels**, while the remaining 37 channels are used for **Secondary Advertisements** and for data transfer during a connection. We'll cover these concepts in detail in the chapter titled "**Advertising and Scanning**", but let's briefly cover the concepts here.

Advertising always starts with **advertisement packets** being sent on the three **Primary Advertising Channels** (or a subset of these channels). This allows the devices scanning for **advertisers** to find them and read their **advertisement data**. The **scanner** can then initiate a **connection** if the **advertiser** allows it. It can also request what's called a **scan request**, and if the advertiser supports this scan request functionality, it will respond with a **scan response**. Scan requests and scan responses allow the advertiser to send additional advertisement data to devices that are interested in receiving this data.

Here are some other important technical details pertaining to the Physical Radio:

- It uses Frequency Hopping Spread Spectrum (FHSS), which allows the two communicating devices to switch to randomly (agreed-on) selected frequencies for exchanging data. This greatly improves reliability and allows the devices to avoid frequency channels that may be congested and used by other devices in the surrounding environment.
- The transmission power levels are:
 - Maximum: 100mW (+20 dBm) for version ≥ 5 , 10mW (+10 dBm) for version ≤ 4.2
 - Minimum: 0.01 mW (-20 dBm)
- In older versions of Bluetooth (4.0, 4.1, and 4.2), the data rate was fixed at 1 Mbps. The physical layer radio (PHY) in this case is referred to as the **1M PHY** and is mandatory in all versions including Bluetooth 5. With Bluetooth 5, however, two new optional PHYs were introduced:
 - **2Mbps PHY**, used to achieve twice the speed of earlier versions of Bluetooth.
 - **Coded PHY**, used for longer range communication.

Note: We'll be covering these two new PHYs as well as the concept of **coding** in more detail in the chapter on Bluetooth 5.

Link Layer

The **link layer** is the layer that interfaces with the **physical layer (radio)** and provides the higher-level layers an abstraction and a way to interact with the radio (through an intermediary level called the **HCI layer** which we'll discuss shortly). It is responsible for managing the state of the radio as well as the timing requirements necessary for satisfying the BLE specification. It is also responsible for managing hardware accelerated operations such as: CRC, random number generation, and encryption.

The three main states in which a BLE device operates in are:

- The Advertising state

- The Scanning state
- The Connected state

When a device advertises, it allows other devices that are scanning to find the device and possibly **connect** to it. If the advertising device allows **connections** and a scanning device finds it and decides to connect to it, they each enter into the **connected** state. The link layer manages the different states of the radio, shown in the following figure:

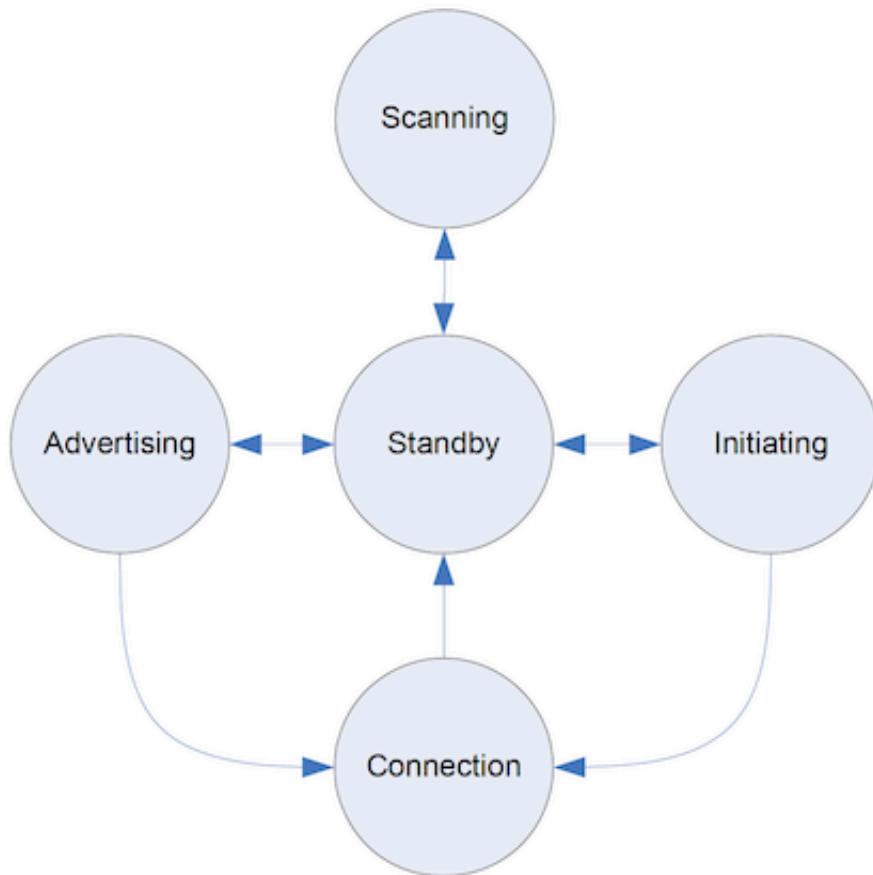


Figure 9: Link layer states
 (Source: Bluetooth 5.0 specification document)

- **Standby:** the default state in which the radio does not transmit or receive any data.
- **Advertising:** the state in which the device sends out advertising packets for other devices to discover and read.
- **Scanning:** the state in which the device scans for devices that are Advertising
- **Initiating:** the state in which a scanning device decides to establish a connection with a device that is advertising.
- **Connected:** the state in which a device has an established link with another device and regularly exchanges data with this other device. This applies to both a device that was in the advertising state or one that was scanning for advertisements and then decided to **initiate** a connection with the advertising device. In this connected state, the device that initiates the connection is called the **master**, and the device that was advertising is now called the

slave.

We'll be covering advertising, scanning, and connected states in more detail in the later chapters.

Bluetooth Address

Bluetooth devices are identified by a 48-bit address, similar to a MAC address. There are two main types of addresses: **Public Addresses** and **Random Addresses**.

Public Address

This is a fixed address that does not change and is factory-programmed. It must be registered with the IEEE (similar to a WiFi or Ethernet device MAC address).

Random Address

Since manufacturers have a choice on what type of **address** to use (Random vs. Public), Random addresses are more popular since they do not require registration with the [IEEE](#). A random address is programmed on the device or generated at runtime. It can be one of two sub-types:

- **Static Address**

- Used as a replacement for Public addresses.
- Can be generated at boot up OR stay the same during lifetime.
- Cannot change until a power cycle.

- **Private Address**

This one is also split up into two additional sub-types:

- Non-resolvable Private Address:
 - Random, temporary for a certain time.
 - Not commonly used.
- Resolvable Private Address:
 - Used for privacy.
 - Generated using **Identity Resolving Key (IRK)** and a random number.
 - Changes periodically (even during the lifetime of the connection).
 - Used to avoid being tracked by unknown scanners
 - Trusted devices (or **Bonded**, which is described later in the chapter on **Security**) can resolve it using the previously stored IRK.

Direct Test Mode

Direct Test Mode (DTM) is only needed for performing RF tests and used during manufacturing and for certification tests. This layer is beyond the scope of this book, so we won't get into it in any detail.

Host Controller Interface (HCI) Layer

The **HCI layer** is a standard protocol defined by the Bluetooth specification that allows the **host** layer to communicate with the **controller** layer. These layers could exist in separate chipsets, or they could exist in the same chipset. In this sense, it also allows interoperability between chipsets, so a device developer can choose two Bluetooth certified devices, a controller and a host, and be 100% confident that they are compatible with each other in terms of communication between the host and controller layers.

In the case where the host and controller are in separate chipsets, the HCI layer will be implemented over a physical communication interface. The three officially supported hardware interfaces by the spec are: UART, USB, and **SDIO (Secure Digital Input Output)**. In the case where the two layers (host and controller) live on the same chipset, the HCI layer will be a logical interface instead.

The job of the HCI layer is to relay commands from the host down to the controller and send events back up from the controller to the host. Following is an example of a capture of HCI commands, HCI events, and ATT commands being exchanged between the host and controller layers:

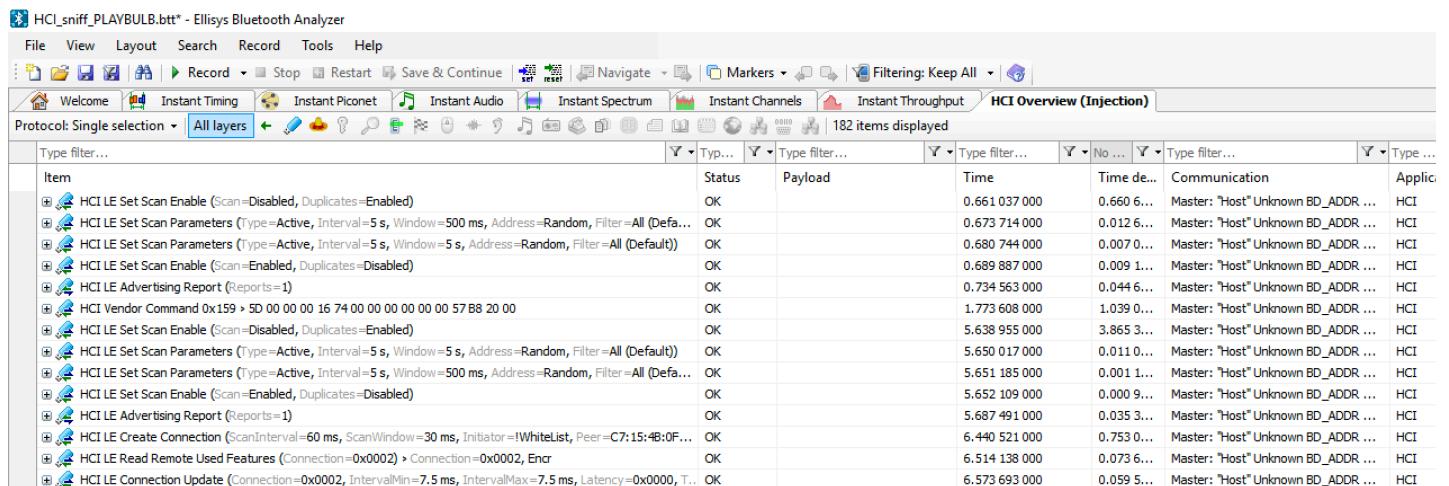


Figure 10: Capture of HCI packets

Examples of the messages include: command packets, configuring the controller, requesting actions, controlling the connection and connection parameters, event packets, command completion and status events.

Logical Link Control and Adaptation Protocol (L2CAP) Layer

The **Logical Link Control and Adaptation Protocol (L2CAP)** layer acts as a protocol-multiplexing layer. It is borrowed from the Bluetooth Classic standard, and performs the following tasks in the case of BLE:

- Takes multiple protocols from the upper layers and places them in standard BLE packets that are passed down to the lower layers beneath it.
- Handles fragmentation and recombination. It takes the larger packets from the upper layers and splits them into chunks that fit into the maximum BLE payload size supported for transmission. On the receiver side, it takes

multiple packets and combines them into one packet that can be handled by the upper layers.

For BLE, the L2CAP layer handles two main protocols: the Attribute Protocol (ATT) (covered in the chapter on GATT), and the Security Manager Protocol (SMP) (covered briefly in the chapter on Security).

Upper Level Layers

The Attribute Protocol (ATT), Generic Attribute Profile (GATT), Security Manager (SM) and Generic Access Profile (GAP) will all be covered in detail in the following chapters.

BLE Peripherals and Centrals

There are a few important terms that you'll come across while learning about BLE. Two of the most important are: **BLE central** and **BLE peripheral**. These two terms relate to the role of a BLE device, but they can be confusing sometimes. Let's go over each of these terms in a bit more detail.

Peripherals

A **peripheral** device is a device that announces its presence by sending out **advertising packets** and accepts a connection from another BLE device (*the BLE central — which will be explained shortly*). Another related term is a **BLE broadcaster**. A broadcaster is a device that sends out advertising packets as well, but with one difference from a peripheral: the broadcaster does not allow a connection from a central device. On the other hand, an **observer** device only discovers advertising devices, but does not have the capability to initiate a connection with the advertiser.

A typical example of an application that involves a broadcaster is in **Beacon** technologies. Beacons are devices that have the sole purpose of advertising and broadcasting their existence, while not accepting connections from other devices. They are becoming popular in two main use cases: retail marketing and indoor location services.

For example, some department stores utilize a smartphone app that can detect Beacons in certain locations within the store. If a customer who has the store's app installed on their smartphone (and has enabled location services) approaches a Beacon, the app displays a special offer to the customer on their phone.

The way a Broadcaster is differentiated from a peripheral device is by the advertising packets that get transmitted by the device. There are different types of advertising packets: some indicate the capability to accept a connection and others are simply for broadcasting presence. When the BLE central discovers the advertising packets of another BLE device (whether broadcaster or peripheral), it knows whether it can initiate a connection or not based on the type of advertising packets.

Once a peripheral gets connected to a BLE central, it also becomes known as the **slave** in that connection. The central device, in this case, gets called the **master**. These are roles defined within the link layer, whereas the peripheral and central roles are defined within the GAP layer.

Centrals

We've briefly mentioned the BLE Central, but to formally define it: A **Central** is a device that discovers and listens to other BLE devices that are advertising. It is also capable of establishing a connection to BLE peripherals (usually multiple at the same time).

An **Observer**, on the other hand, is a similar type of BLE device, but one that is not capable of initiating a connection with a peripheral device.

Observers and Broadcasters vs. Centrals and Peripherals

Let's go over some advantages and disadvantages of the four different types of device: Observers, Broadcasters, Centrals, and Peripherals.

Broadcaster	Peripheral	Observer	Central
No need for a radio receiver	Needs both a receiver and transmitter	No need for a transmitter	Needs both a receiver and transmitter
No bi-directional data transfer	Supports bi-directional data transfer	No bi-directional data transfer	Supports bi-directional data transfer
Reduced hardware, reduced BLE software stack	Requires the full BLE software stack	Reduced hardware, reduced BLE software stack	Requires the full BLE software stack

Table 2: Comparison between Observers, Broadcasters, Peripherals, and Centrals

Power Consumption and Processing Power Considerations

BLE is asymmetrical by design. Much of the heavy lifting regarding connection management, time management, and processing responsibilities lies on the central side. This helps reduce power consumption and processing power requirements on the peripheral side, thus, making it possible to integrate BLE into smaller and more resource-constrained devices (e.g., battery-powered devices).

A BLE central device can still be battery powered, but will usually have a relatively large battery that's rechargeable. Most commonly, in a BLE system, the central device is a smartphone, tablet, or a computer.

A central device also supports connecting to multiple Peripherals at the same time. A typical example of this is a smartphone that maintains a connection to a smartwatch, a smart-home thermostat, and a fitness tracker, all at the same time.

Multi-Role BLE Devices

In some use cases, a BLE device would benefit from acting in multiple roles simultaneously. For example, a device may want to monitor multiple sensors (peripheral devices), and at the same time be able to advertise its presence to a smartphone to allow access to sensor data from a mobile app interface.

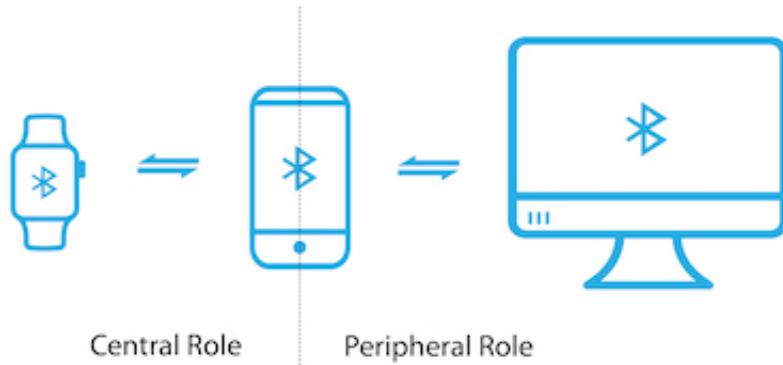


Figure 11: Multiple roles in BLE

The Role of Smartphones in BLE

One of the biggest advantages of BLE over other competing low-power wireless technologies (such as ZigBee, Z-Wave, Thread, etc.) is its existence in the majority of smartphones in the market. Most (if not all) smartphones already included Bluetooth Classic since the very early days, and most Bluetooth chipset vendors are now integrating BLE support along with Bluetooth Classic in their chipsets. The result is that the vast majority of smartphones nowadays support BLE.

Having the capability for a smartphone to interact and connect to BLE devices provides a couple of significant advantages:

- Smartphones provide a familiar user interface for consumers, offering a rich user experience when using a mobile app to interface with a BLE device (compared to interfacing with the BLE device directly).
- Smartphones are usually connected to the Internet. This means that the data transmitted from the BLE device can be sent up to the cloud and stored somewhere else for later access or analysis.

Challenges with BLE Development on Smartphones

There are two major mobile operating systems: Android and iOS. Android introduced native support for BLE APIs in Android 4.3 (released July 2012), while iOS provided native BLE support a bit earlier in iOS 5 (released October 2011).

One important thing to note is that this also depends on the hardware running the operating system. For iOS, this included all iOS devices starting with the iPhone 4s. For Android, it's a completely different story: Android runs on devices manufactured by many different vendors, so there's no easy way to determine which devices first started supporting BLE. This Android fragmentation problem introduces a big challenge with developing Android BLE applications that behave consistently across the dozens of existing Android phones.

Advertising and Scanning

Generic Access Profile (GAP)

The **Generic Access Profile (GAP)** provides the framework that defines how BLE devices interact with each other. This includes the following aspects:

- The Modes & Roles of BLE devices.
- Advertisements (advertising, scanning, advertising parameters, advertising data, and scanning parameters).
- Connection establishment (initiating, accepting, and connection parameters)
- Security (which we'll cover in its own chapter).

The implementation of this framework is mandatory per the official specification, and it is what allows two or more BLE devices to interoperate, communicate, and be able to exchange data with each other.

We talked briefly about the advertising and scanning states of a BLE device, and we mentioned that a BLE device always starts in the advertising state. This is the case even when it wants to operate in the connected state most of the time. In order for two BLE devices to discover each other, one of them has to advertise while the other scans the three Primary Advertising channels (RF channels 37, 38, and 39) looking for advertisement packets sent by the advertising device.

If the advertising device supports a connection and a central device discovers it, it may choose to establish a connection. In this chapter, we will focus on these initial states: **advertising** and **scanning**.

Advertising State

In the advertising state, a device sends out packets containing useful data for others to receive and process. The packets are sent at a fixed interval defined as the **advertising interval**. There are **40 RF channels** in BLE, each separated by 2 MHz (center-to-center), as shown in the following figure. Three of these channels are called the **Primary Advertising Channels**, while the remaining 37 channels are used for **Secondary Advertisements** and for data packet transfer during a connection.

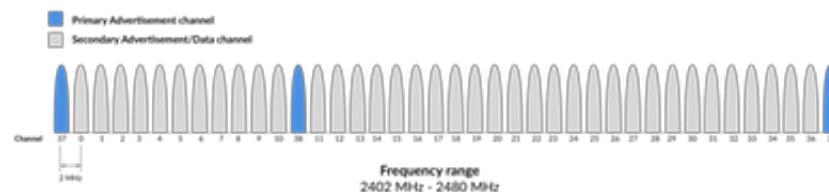


Figure 12: RF channels in BLE

Note: Since these are the three channels that a device starts by advertising on, and usually switches between them,

they are spread apart in the frequency spectrum to avoid radio interference between a device that's advertising on one channel and another that's advertising on a different channel. Also, the locations of these primary channels (RF channels 37, 38, and 39) were chosen within the spectrum to avoid interference with the most commonly used WiFi channels.

Advertisements always start with advertisement Packets sent on the three Primary Advertising Channels (or a subset of these channels). This allows centrals to find the advertising device (peripheral) and parse its advertisement packets. The central can then initiate a connection if the advertiser allows it. The central can also request what's called a **scan request**, and if the Advertiser supports it, it will respond with a **scan response**. Scan requests and responses allow the advertiser to send additional advertising data that would not fit in the initial advertisement packet.

Note: Primary advertisement data is limited to 31 bytes. Secondary advertisement data, on the other hand, supports up to 254 bytes of data.

As we've mentioned before, some devices (broadcasters) stay in the advertising state and do not accept connections (connectionless), while others (peripherals) allow the transition to the connected state if a central initiates a connection (connection-oriented). For example, most Beacons stay in the advertising state during the lifetime of the device.

The main advantage of staying in the advertising state is that multiple centrals can discover the advertising data without the need for a connection. However, the downsides are the lack of security and the inability for the advertiser to receive data from a central (data transfer is unidirectional).

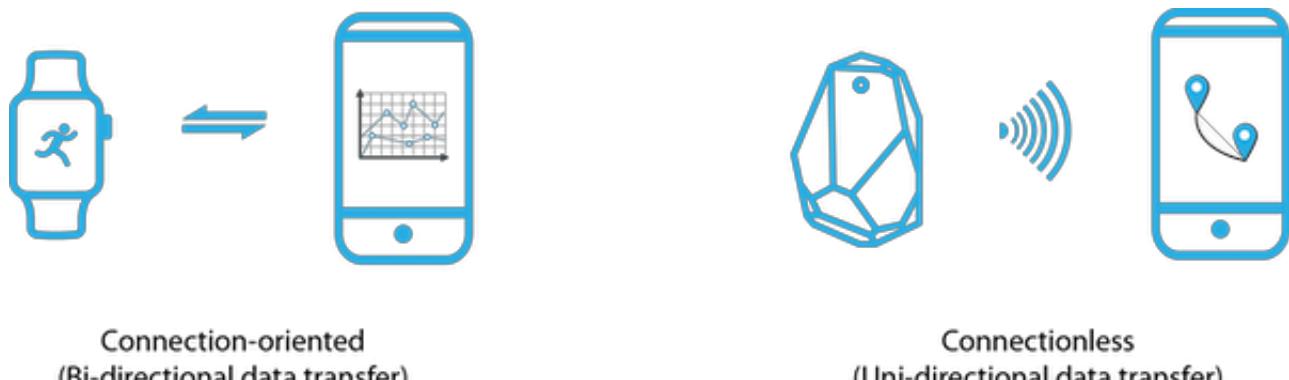


Figure 13: Connection-oriented vs. connectionless

Scanning State

Centrals tune to the three Primary Advertising Channels one at a time. So, in order for a central to discover a peripheral, the central has to be tuned to the same channel on which the peripheral is advertising at that given point. To increase the possibility of this happening, and in order to make it happen quickly, a few advertising and scanning parameters can be adjusted.

A device that listens for advertisements, and then sends scan Requests from the advertisers is defined to be in the **active scanning** mode, while a device that passively listens to advertising packets and does not send scan requests is said to be in the **passive scanning** mode.

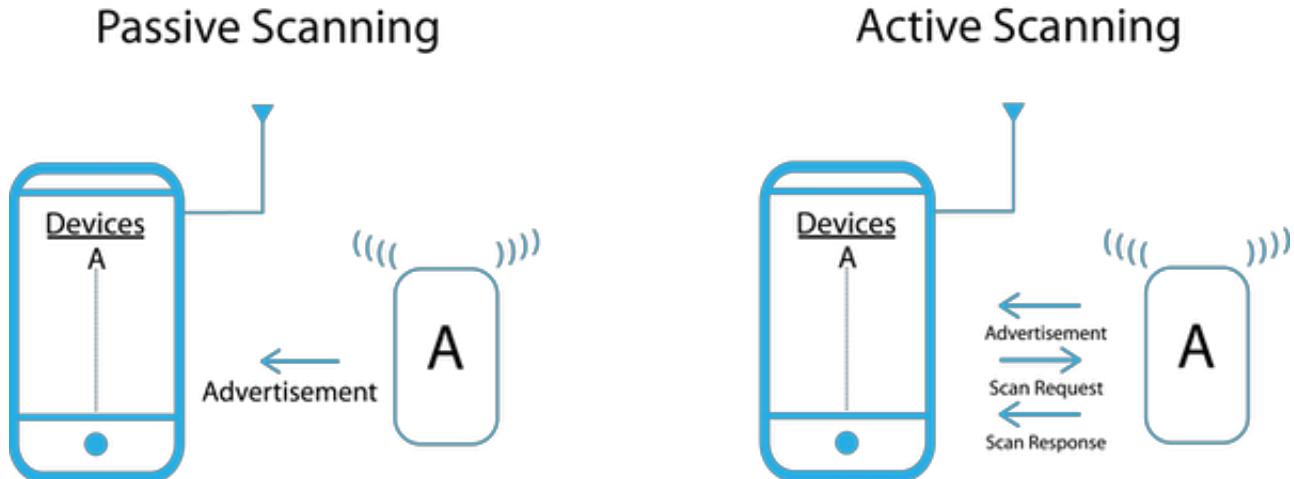


Figure 14: Passive vs. Active Scanning

Advertising Events

An **advertising event** is made up of multiple advertising packets being sent on all, or a subset of, the three Primary Advertising Channels (37, 38, and 39). There are **seven** types of advertising events (think of these as the different types of advertising packets):

- **Connectable and Scannable Undirected Event**

This type allows other devices to receive the advertisement packets, send a scan request to the advertiser, and establish a connection with it.

- **Connectable Undirected Event**

This type allows other devices to receive the advertisement packets and establish a connection with the advertiser.

- **Connectable Directed Event**

This type allows a **specific** device to receive the advertisement packets and establish a connection with the advertiser.

- **Non-Connectable and Non-Scannable Undirected Event**

This type allows other devices to receive its advertisement packets. However, it does **not** allow scan requests or the establishment of a connection with the advertiser.

- **Non-Connectable and Non-Scannable Directed Event**

This type allows a **specific** device to receive the advertisements without the ability to establish a connection with the advertiser or to send scan requests.

- **Scannable Undirected Event**

This type allows other devices to send a scan request to the advertiser to receive additional advertisement data.

- **Scannable Directed Event**

This type allows a **specific** device to send a scan request to the advertiser to receive additional advertisement data.

Advertising Parameters

The different **advertising parameters** are:

- **Advertising Interval**

The most important parameter related to advertisements is the **advertising interval**. The advertising interval value ranges all the way from **20 milliseconds** up to **10.24 seconds** in small increments of **625 microseconds**. The advertising interval greatly impacts battery life and should be chosen carefully. It's recommended to choose the longest advertising interval that provides a balance between fast connectivity and reduced power consumption.

- **Advertising/Scan Response Data**

Let's take a look at what fields are usually included in an advertisement packet, and what the packet format looks like. Note that scan responses share the same format.

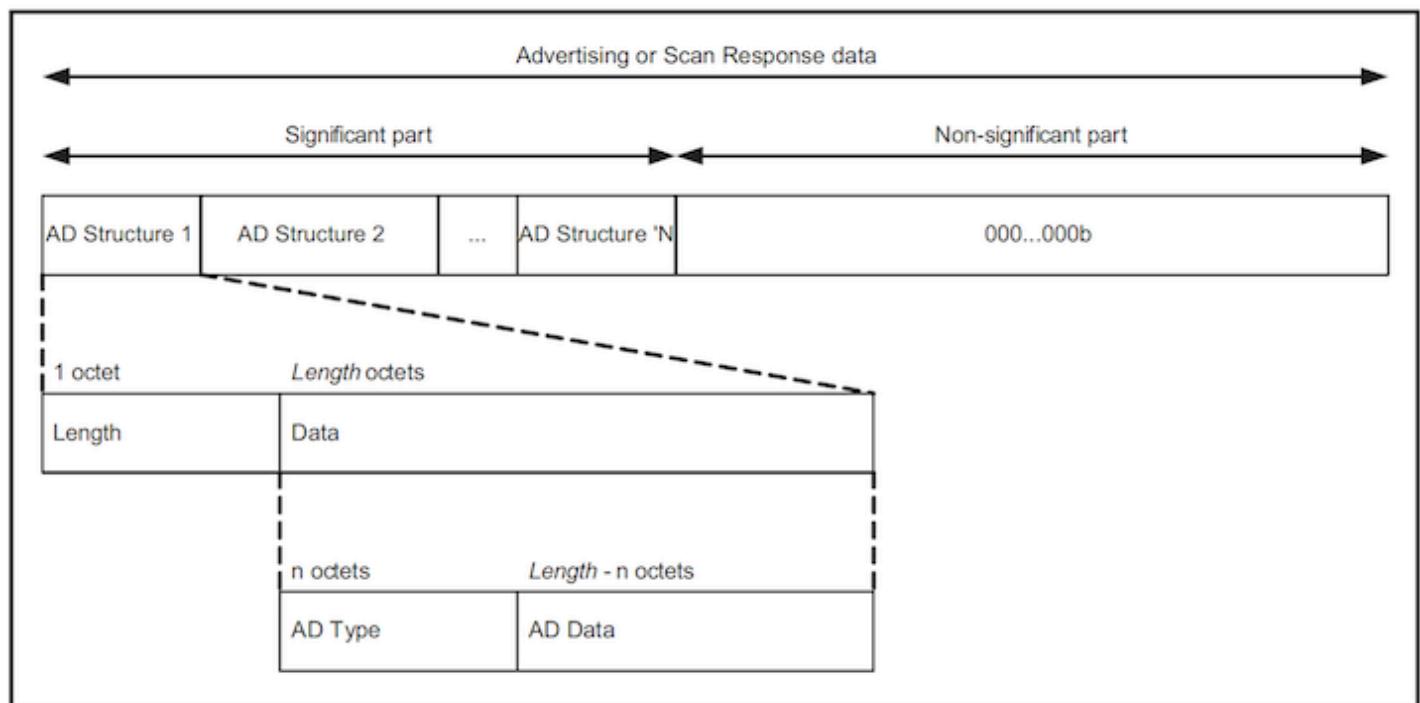


Figure 15: Advertising data packet format
(Source: Bluetooth 5 specification document)

The **advertising data** follows a format for organizing data similar to TLV (Type-Length-Value) used in data

communications, except that the length comes before the type. The advertising data goes into the PDU portion of the BLE packet and contains the following:

- **Length:** The length of the data that follows the length value itself (includes the AD Type as well as the AD Data).
- **Advertising Data Type (AD Type):** The type of advertisement data included in this specific TLV.
- **Advertising Data:** The actual value of the advertisement data.

Advertising Data (AD) types are defined in the [Bluetooth Core Specification Supplement document](#) (not the Core Specification document).

Some of the most commonly used AD Types:

- **Local Name:** contains the device name that is read by scanners that discover the advertising device.
- **Tx Power Level:** transmission power level, defined in units of dBm.
- **Flags:** multiple one-bit boolean (a binary variable, having two possible values: **TRUE [1]** or **FALSE [0]**) flags, including:
 - Limited Discoverable Mode
 - General Discoverable Mode
 - BR/EDR Not Supported
 - Simultaneous LE and BR/EDR to Same Device Capable (controller)
 - Simultaneous LE and BR/EDR to Same Device Capable (host)

Note: BR/EDR refers to Bluetooth Basic Rate/Enhanced Data Rate (i.e. Bluetooth Classic).

- **Service Solicitation:** a list of one or more UUIDs indicating what **services** are supported and exposed by the device's GATT server. This helps central devices learn the available services (explained in a later chapter) exposed by a device before establishing a connection.
- **Appearance:** this defines the external appearance of the device according to the [Bluetooth Assigned Numbers](#). These include appearances such as phone, heart rate sensor, key ring and many more. If you cannot find an appearance that fits the nature of your device, you can use the **UNKNOWN APPEARANCE** value.

Scanning Parameters

The three main **scanning parameters** are:

- **Scan Type:** Passive vs. Active Scanning.
- **Scan Window:** indicates how long to be scanning for advertisements.
- **Scan Interval:** indicates how often to scan for advertisements.

The scanner will listen for the complete **scan window** at every **scan interval**, and in each scan window it will listen on a different Primary Advertising Channel. Scan window and scan interval are configurable aspects of a scanner's

behavior.

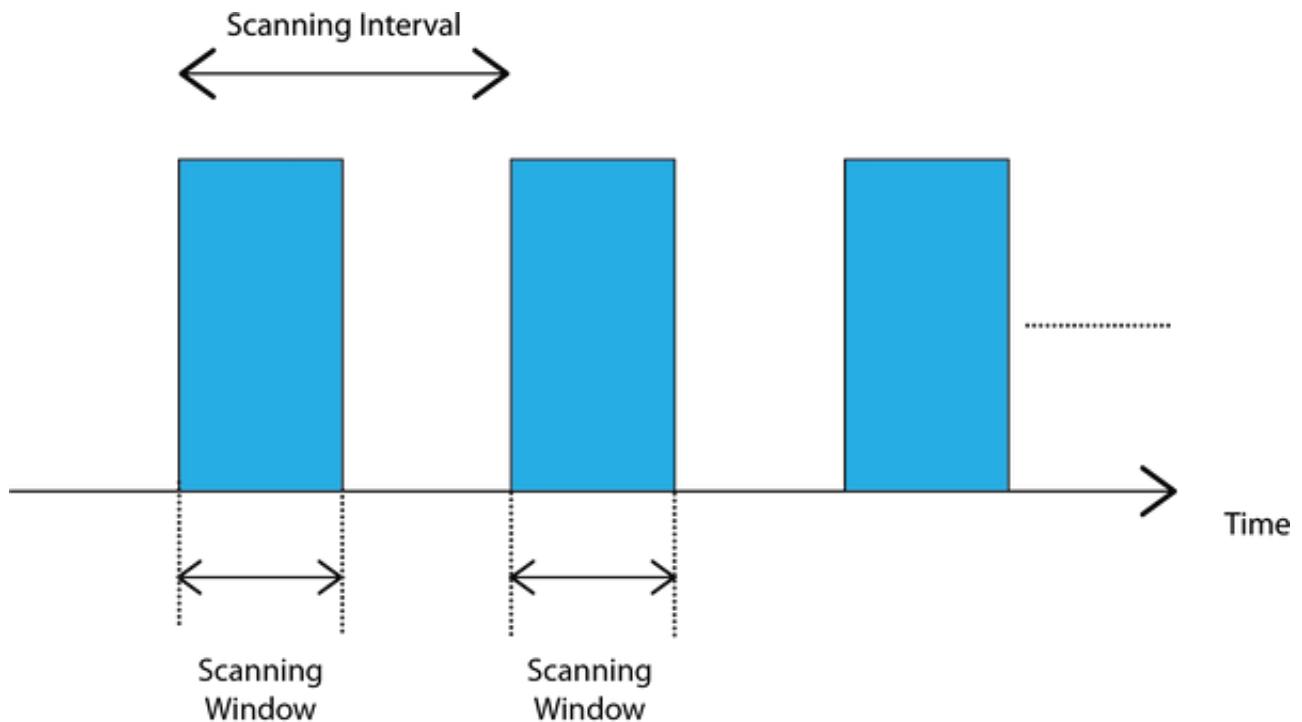


Figure 16: Scanning parameters

Connections

In order for two BLE devices to connect to each other, the following steps need to occur:

- The peripheral needs to start advertising and send out connectable advertisement packets.
- The central device needs to be scanning for advertisements while the peripheral is advertising.
- If the central happens to be listening on an advertising channel that the peripheral is advertising on, then the central device discovers the peripheral. It is then able to read the advertisement packet and all the necessary information in order to establish a connection.
- The central then sends a **CONNECT_IND** packet (also known as a **connection request** packet).
- The peripheral always listens for a short interval on the same advertising channel after it sends out the advertising packet. This allows it to receive the connection request packet from the central device — which triggers the forming of the connection between the two devices.

After this occurs, the connection is considered **created**, but not yet **established**. A connection is considered **established** once the device receives a packet from its peer device. After a connection becomes established, the central becomes known as the **master**, and the peripheral becomes known as the **slave**. The master is responsible for managing the connection, controlling the connection parameters, and the timing of the different events within a connection.

Connection Events

During what's called a **connection event**, the master and slave alternate sending data packets to each other until neither side has more data left to send. Here are a few aspects of connections that are **very important** to know:

- A **connection event** occurs periodically and continuously until the connection is closed or lost.
- A connection event contains at least one packet sent by the master.
- The slave always sends a packet back if it receives a packet from the master
- If the master does not receive a packet back from the slave, the master will close the connection Event — *it resumes sending packets at the next connection Event*.
- The connection Event can be closed by either side.
- The starting points of consecutive Connection Events are spaced by a period of time called the **connection interval**.

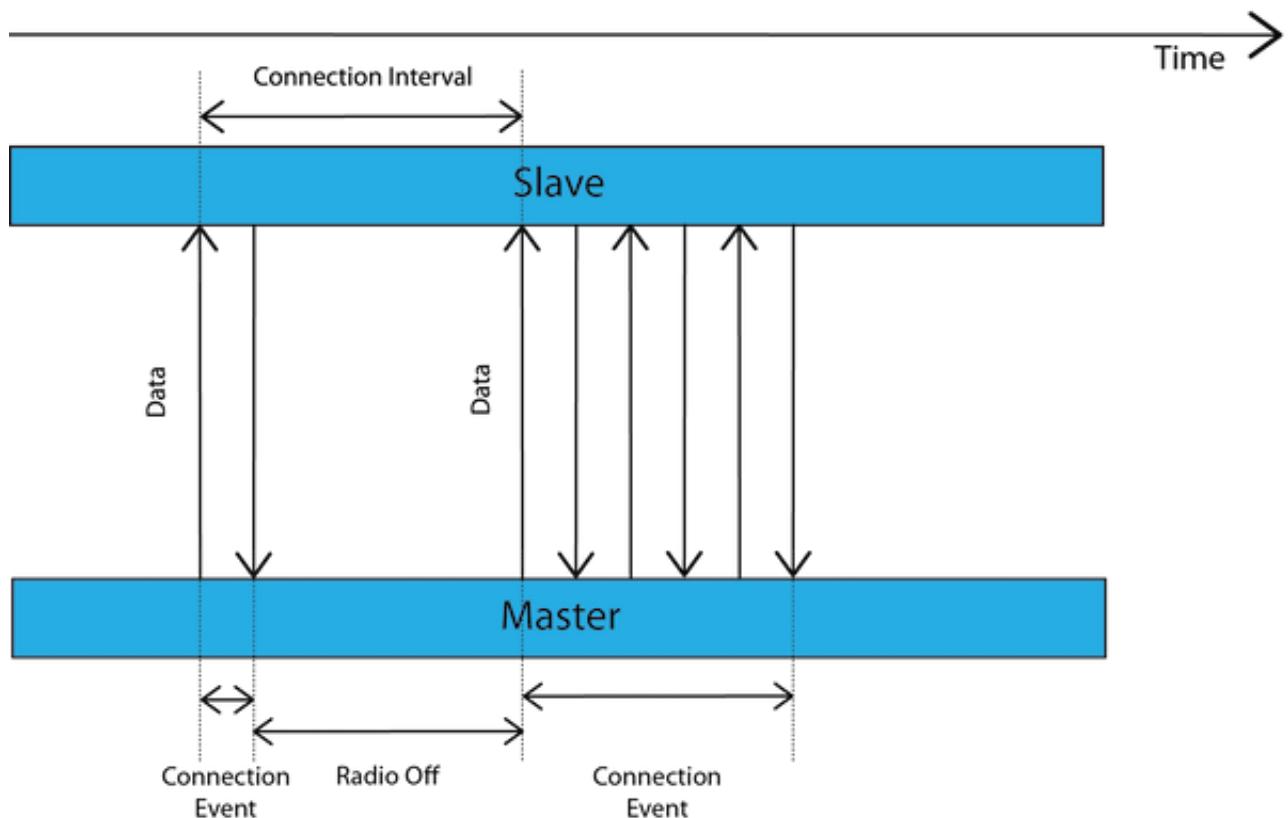


Figure 17: The connection interval and connection events

Connection Parameters

The parameters that define connections are:

- **Connection interval**

The Connection Interval value ranges between **7.5 milliseconds - 4.0 seconds** in increments of **1.25 milliseconds**. It is set by the central device in the connection request packet. The central may take into account the **Peripheral Preferred Connection Parameters (PPCP)**, which is a way for the peripheral to inform the central of a set of parameters that it prefers. In the end, though, it is up to the central to respect these values or ignore them.

- **Slave Latency**

The **slave latency** parameter allows the peripheral to skip a number of consecutive connection Events and not listen to the central at these connection events without compromising the connection. This allows the peripheral to sleep for longer periods of time, potentially reducing power consumption. The slave latency value defines the **number** of connection events it can safely skip.

For example, if the slave latency is set to **three**, then the peripheral may skip **three** consecutive connection events, but it then needs to wake up the radio and listen to the central a to listen — and respond — at every connection event.

- **Supervision Timeout**

The **supervision timeout** is used to detect a loss in connection. It is defined as the **maximum time between two received data packets before the Connection is considered lost**. Its value ranges between **100 milliseconds - 32 seconds** in increments of **10 milliseconds**. Another condition for this timeout value is:

$$\text{SupervisionTimeout} > (1 + \text{connSlaveLatency}) * \text{connInterval} * 2$$

The one exception — where the supervision timeout does not apply — is after a connection is **created**, but not yet established. In this case, the master will consider the connection to be lost if it does not receive the first packet from the slave within:

$$6 * \text{connInterval}$$

- **Data Length Extension (DLE)**

This is a setting that can be enabled or disabled. It allows the packet size to hold a larger amount of payload (up to 251 bytes vs. 27 when disabled). This feature was introduced in version 4.2 of the Bluetooth specification.

- **Maximum Transmission Unit (MTU)**

MTU stands for **Maximum Transmission Unit** and is used in computer networking to define the maximum size of a Protocol Data Unit (PDU) that can be sent by a specific protocol. The **Attribute MTU (ATT_MTU** as defined by the specification) is the largest size of an ATT payload that can be sent between a client and a server.

The effective ATT_MTU gets determined by the minimum value of the maximum ATT_MTU values that the master and slave support. For example, if a master supports an ATT_MTU of **100 bytes** and the slave responds that it supports an ATT_MTU of **150 bytes**, then the master will decide that the ATT_MTU to be used for the connection from thereon is **100 bytes**.

Note: To achieve maximum throughput, make sure you enable DLE (that is, if you are running Bluetooth 4.2 or greater). This reduces the packet overhead and any unnecessary header data that gets transmitted with smaller packets.

Channel Hopping

As we discussed at the beginning of this chapter, there are 37 RF channels utilized for transmitting data packets during a connection. However, not all 37 channels are necessarily used during a connection. The **used** channels are defined by the **channel map**, which is included in the connection request packet sent by the central to the peripheral to initiate a connection. For each connection event, the data packets will be sent on a different channel within the channel map.

The sequence of channels used for each of the connection events is determined by the channel map as well as another value called the **hop increment**. The hop increment — like the channel map — is also included in the connection request packet. The combination of the channel map and hop increment determines which channel gets used at each connection interval.

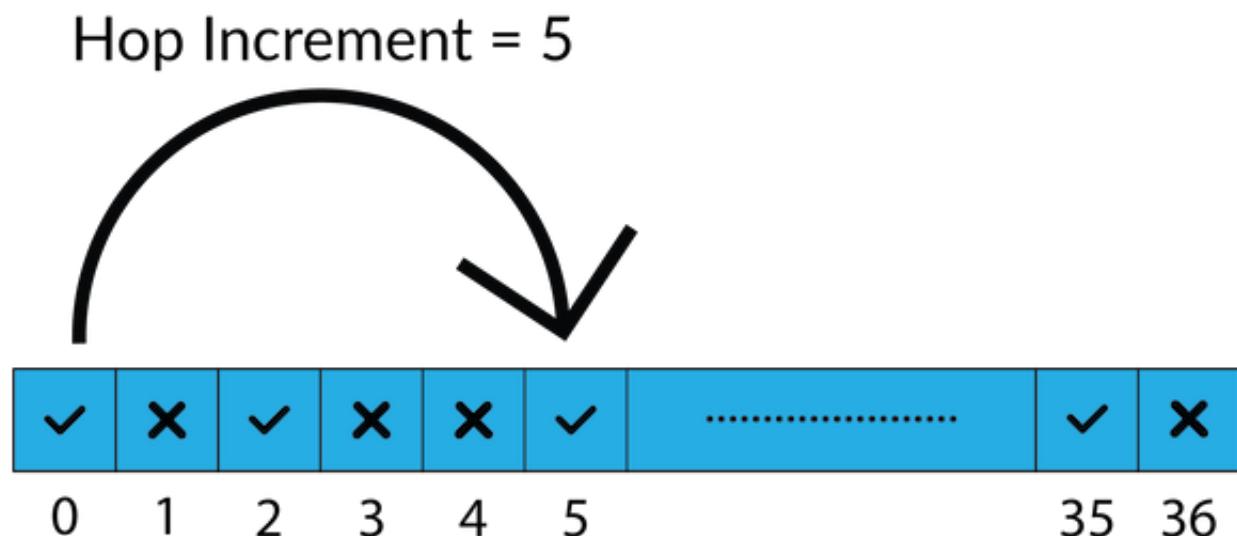


Figure 18: Channel map and hop increment

There are two **channel selection algorithms** used within BLE: **channel selection algorithm #1**, and **channel selection algorithm #2**. Covering the details of these algorithms is outside the scope of this book. To learn more about these algorithms and how they work, refer to the Bluetooth specification document (version 5.0 | Vol 6, Part B, Section 4.5.8.2).

White List & Device Filtering

BLE supports device filtering for procedures related to: the advertising state, the scanning state, and the initiating state (for establishing connections). A **white list** is a list of addresses and address types of specific devices. It is used for determining which peer devices a particular device is interested in. An entry for an **anonymous** device address type allows matching all advertisements sent with no address. Device Filtering gets processed at the link layer in the controller (the lower layer of the Bluetooth stack), which saves time and overhead from being performed at the host (the upper layer of the stack). However, the host is responsible for configuring the white list.

Here's a list of the different white list filter policies that apply to each of these states:

- **Advertising State Filter Policy** (peripheral side)

This filter policy defines how the advertiser processes both scan and connection requests. The different

configurations include:

- Process scan and connection requests only from devices in the white list.
- Process scan and connection requests from all devices (white list not used).
- Process scan requests only from devices in the white list, while processing connection requests from all devices.
- Process connection requests only from devices in the white list, while processing scan requests from all devices.

- **Scanning State Filter Policy** (central side)

This filter policy defines how the scanner processes advertising packets. The different configurations include:

- Process advertising packets from all devices (white list not used).
- Process advertising packets only from devices in the white list.

- **Initiating State Filter policy** (central side)

This filter policy defines how a connection initiator processes advertising packets. The different configurations include:

- Process and initiate a connection to all devices listed in the white list.
- Process and initiate a connection only to a device specified by the host.

Notice that it's not an option to process and connect to a connectable advertising device that's not in the white list.

Services and Characteristics

Before explaining what **services** and **characteristics** are, we first need to cover two very important concepts: the **Generic Attribute Profile (GATT)** and the **Attribute Protocol (ATT)**.

GATT stands for Generic Attribute Profile. To understand what GATT is, we first need to understand the underlying framework for GATT: the Attribute Protocol (ATT). The GATT only comes into play after a connection has been established between two BLE devices.

Attribute Protocol (ATT)

ATT defines how a server exposes its data to a client and how this data is structured. There are two roles within the ATT:

- **Server:**

This is the device that exposes the data it controls or contains, and possibly some other aspects of server behavior that other devices may be able to control. It is the device that accepts incoming commands from a peer device, and sends **responses**, **notifications** and **indications**.

For example, a thermometer device will behave as a server when it exposes the temperature of its surrounding environment, the unit of measurement, its battery level, and possibly the time intervals at which the thermometer reads and records the temperature. It can also **notify** the client (defined later) when a temperature reading has changed rather than have the client poll for the data waiting for a change to occur.

- **Client:**

This is the device that interfaces with the server with the purpose of reading the server's exposed data and/or controlling the server's behavior. It is the device that sends commands and requests and accepts incoming notifications and indications. In the previous example, a mobile device that connects to the thermometer and reads its temperature value is acting in the Client role.

The data that the server exposes is structured as **attributes**. An attribute is the generic term for any type of data exposed by the server and defines the structure of this data. For example, services and characteristics (both described later) are types of attributes. Attributes are made up of the following:

- **Attribute type (Universally Unique Identifier or UUID)**

This is a 16-bit number (in the case of Bluetooth SIG-Adopted Attributes), or 128-bit number (in the case of custom **attribute types** defined by the developer, also sometimes referred to as **vendor-specific UUIDs**).

For example, the UUID for a SIG-adopted temperature measurement value is **0xA1C** SIG-adopted attribute types (UUIDs) share all but 16 bits of a special 128-bit base UUID:

00000000-0000-1000-8000-00805F9B34FB

The published 16-bit UUID value replaces the 2 bytes in **bold** in the base UUID.

A custom UUID, on the other hand, can be any 128-bit number that does not use the SIG-adopted base UUID. For example, a developer can define their own attribute type (UUID) for a temperature reading as:

F5A1287E-227D-4C9E-AD2C-11D0FD6ED640

One benefit of using a SIG-adopted UUID is the reduced packet size since it can be transmitted as the 16-bit representation instead of the full 128-bit value.

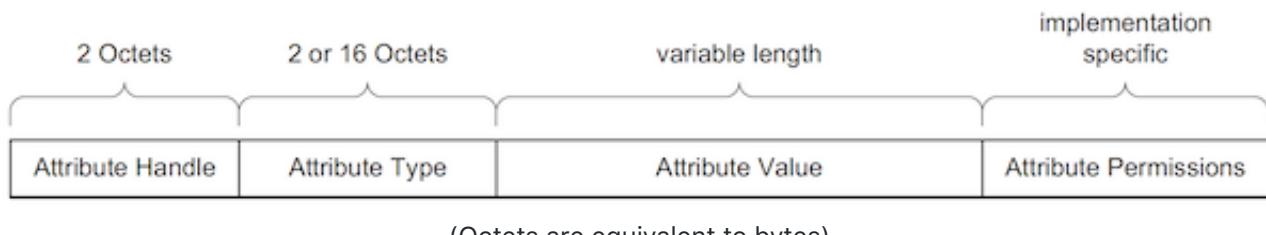
- **Attribute Handle**

This is a 16-bit value that the server assigns to each of its attributes — *think of it as an address*. This value is used by the client to reference a specific attribute, and is guaranteed by the server to uniquely identify the attribute during the life of the connection between two devices. The range of handles is **0x0001-0xFFFF**, where the value of **0x0000** is reserved.

- **Attribute Permissions**

Permissions determine whether an attribute can be **read** or **written** to, whether it can be **notified** or **indicated**, and what **security levels** are required for each of these operations. These permissions are not defined or discovered via the Attribute Protocol (ATT), but rather defined at a higher layer (GATT layer or Application layer).

The following figure shows a logical representation of an Attribute:



*Figure 19: Logical representation of an attribute
(Source: Bluetooth 5 specification document)*

Generic Attribute Profile (GATT)

Now that we've covered the concept of attributes, we'll go over three important concepts in BLE that you will come across very often:

- **Services**
- **Characteristics**
- **Profiles**

These concepts are used specifically to allow hierarchy in the structuring of the data exposed by the Server. Services and characteristics are types of attributes that serve a specific purpose. Characteristics are the lowest level attribute

within a database of attributes. Profiles are a bit different and are not discovered on a server — we will explain them later in this chapter.

The GATT defines the format of services and their characteristics, and the procedures that are used to interface with these attributes such as service discovery, characteristic reads, characteristic writes, notifications, and indications. GATT takes on the same **roles** as the Attribute Protocol (ATT). The roles are not set per device — rather they are determined per transaction (such as request ↔ response, indication ↔ confirmation, notification). So, in this sense, a device can act as a server serving up data for clients, and at the same time act as a client reading data served up by other servers (all during the same connection).

Services & Characteristics

Services

A **service** is a grouping of one or more attributes, some of which are characteristics. It's meant to group together related attributes that satisfy a specific functionality on the server. For example, the SIG-adopted **battery service** contains one characteristic called the **battery level**. A service also contains other attributes (non-characteristics) that help structure the data within a service (such as **service declarations**, **characteristic declarations**, and others). Here's what a service looks like:

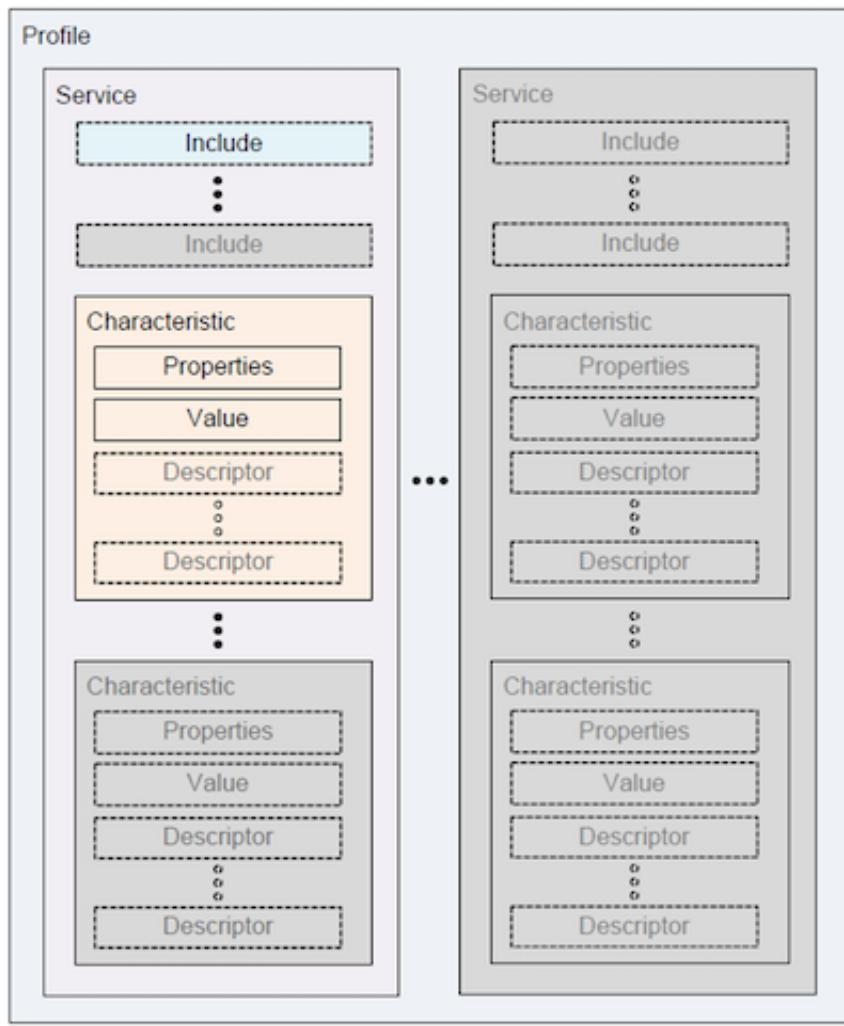


Figure 20: Profiles, Services, and Characteristics
(Source: Bluetooth 5 specification document)

From the figure, we can see the different attributes that a service is made up of:

- One or more **include services**
- One or more characteristics
 - Characteristic properties
 - A characteristic value
 - Zero or more characteristic descriptors

An **include** service allows a service to refer to other services for purposes such as extending the included service.

There are two types of services:

- **Primary Service:** represents primary functionality of a device.

- **Secondary Service:** provides auxiliary functionality of a device and is referenced (included) by at least one other primary service on the device (it is rarely used and won't be referenced in this book).

Characteristics

A **characteristic** is always part of a service and it represents a piece of information/data that a server wants to expose to a client. For example, the battery level characteristic represents the remaining power level of a battery in a device which can be read by a client. The characteristic contains other attributes that help define the value it holds:

- **Properties:** represented by a number of bits and which defines how a **characteristic value** can be used. Some examples include: **read**, **write**, **write without response**, **notify**, **indicate**.
- **Descriptors:** used to contain related information about the characteristic Value. Some examples include: **extended properties**, **user description**, fields used for subscribing to notifications and indications, and a field that defines the presentation of the value such as the format and the unit of the value.

Understanding these concepts is important, however, as an application developer you'll probably interface with APIs provided by the chipset or mobile operating system SDK that abstract out many of these concepts.

For example, you may have an API for enabling notifications on a certain characteristic that you can simply call (you don't necessarily need to know that the stack ends up writing a value of **0x0001** to the characteristic's **Characteristic Configuration Descriptor (CCCD)** on a server to enable notifications).

It's important to keep in mind that while there are no restrictions or limitations on the characteristics contained within a service, services are meant to group together related characteristics that define a specific functionality within a device.

For example, even though it's technically possible — it does not make sense to create a service called the **humidity service** that includes both a **humidity** characteristic and a **temperature** characteristic. Instead, it would make more sense to have two separate services specific to each of these two distinct functionalities (temperature reading, and humidity reading).

It's worth mentioning that the Bluetooth SIG has adopted quite a few services and characteristics that satisfy a good number of common use cases. For these adopted services, specification documents exist to help developers implement them along with ensuring conformance and interoperability with this service.

If a device claims conformance to a service, it must be implemented according to the service specification published by the Bluetooth SIG. This is essential if you want to develop a device that is guaranteed to be connectable with third-party devices from other vendors. The Bluetooth SIG-adopted services make the connection specification "pre-negotiated" between different vendors.

You can find the list of **adopted services** [here](#), and their respective **specifications** [here](#). **Adopted characteristics** can be found [here](#).

Profiles

Profiles are much broader in definition than services. They are concerned with defining the behavior of both the client and server when it comes to services, characteristics and even connections and security requirements. Services and their specifications, on the other hand, deal with the implementation of these services and characteristics on the server side only.

Just like in the case of services, there are also **SIG-adopted** profiles that have published specifications. In a profile specification, you will generally find the following:

- The definition of roles and relationship between the GATT server and client.
- Required Services.
- Service requirements.
- How the required services and characteristics are used.
- Details of connection establishment requirements including advertising and connection parameters.
- Security considerations.

Following is an example of a diagram taken from the Blood Pressure Profile specification document. It shows the relationship between the roles (server, client), services, and characteristics within the profile.

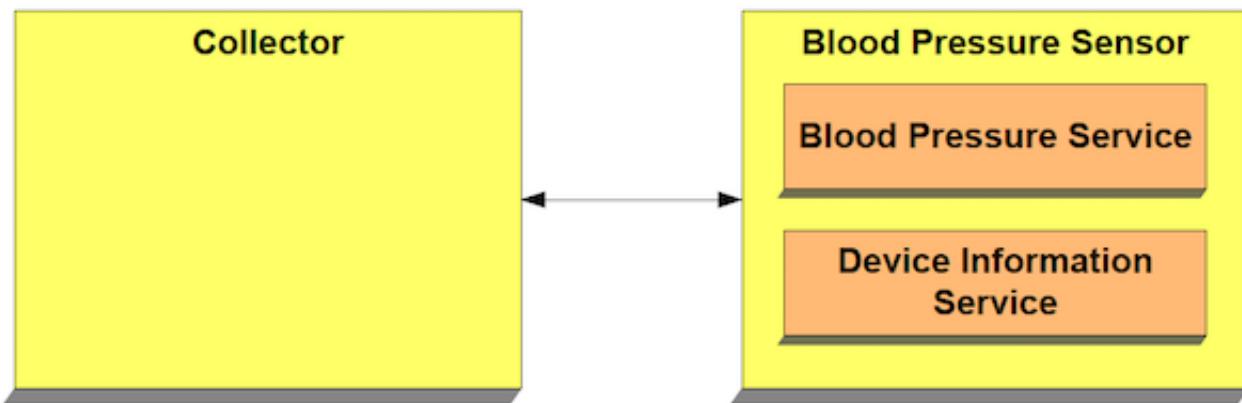


Figure 21: Blood pressure profile

(Source: *Blood Pressure Profile specification document*)

The roles are represented by the yellow boxes, whereas the services are represented by the orange boxes. You can find the list of **SIG-adopted profiles** [here](#).

A GATT Example

Let's look at an example of a GATT implementation. For this example, we'll look at an example **GATT.xml** file that's used by the Silicon Labs Bluetooth Low Energy development framework (BGLib).

```

<?xml version="1.0" encoding="UTF-8" ?>
<configuration>

    <service uuid="1800">
        <description>Generic Access Profile</description>

        <characteristic uuid="2a00">
            <properties read="true" const="true" />
            <value>Bluegiga CR Demo</value>
        </characteristic>

        <characteristic uuid="2a01">
            <properties read="true" const="true" />
            <value type="hex">4142</value>
        </characteristic>
    </service>

    <service uuid="0bd51666-e7cb-469b-8e4d-2742f1ba77cc" advertise="true">
        <description>Cable replacement service</description>

        <characteristic uuid="e7add780-b042-4876-aae1-112855353cc1" id="xgatt_data">
            <description>Data</description>
            <properties write="true" indicate="true" />
            <value variable_length="true" length="20" type="user" />
        </characteristic>
    </service>

</configuration>

```

Figure 22: GATT.xml example from Silicon Labs sample application

In this XML, you'll notice the following:

- There are two services defined:
 - Generic Access Profile (GAP) service with UUID: **0x1800** (SIG-adopted service).
 - Cable Replacement service with UUID: **0bd51666-e7cb-469b-8e4d-2742f1ba77cc** (a custom or vendor-specific service).
- The Generic Access Profile service is mandatory per the spec, and it includes the following mandatory characteristics:
 - **Name** with UUID **0x2a00** and value: **Bluegiga CR Demo**.
 - **Appearance** with UUID **0x2a01** and value **0x4142**.
Appearance value definitions can be found [here](#).

Note: the creation and inclusion of this Service is usually handled by the chipset's SDK, and usually APIs are provided to simply set the Name and Appearance values.

- The Cable Replacement service has one characteristic named **data**
 - The **data** characteristic has a UUID: **e7add780-b042-4876-aae1-112855353cc1**
 - It has both **writes** and **indications** enabled.

Attribute Operations

There are six different types of attribute operations. They are:

- **Commands:** sent by the client to the server and do not require a **response** (*defined below*).
- **Requests:** sent by the client to the server and require a response. There are two types of requests:
 - Find Information Request
 - Read Request
- **Responses:** sent by the server in response to a request.
- **Notifications:** sent by the server to the client to let the client know that a specific characteristic value has changed. In order for this to be triggered and sent by the server, the client has to enable notifications for the characteristic of interest. Note that a notification does not require a response from the client to acknowledge its receipt.
- **Indications:** sent by the server to the client. They are very similar to notifications, but require an acknowledgment to be sent back from the client to let the server know that the indication was successfully received.

Note: Notifications and indications are exposed via the **Client Characteristic Configuration Descriptor (CCCD)** attribute. Writing a "1" to this attribute value enables notifications, whereas writing a "2" enables indications. Writing a "0" disables both notifications and indications.

- **Confirmations:** sent by the client to the server. These are the acknowledgment packets sent back to the server to let it know that the client successfully received an indication.

Flow Control and Sequence of Attribute Operations

Requests are sequential in nature and require a response from the server before a new request can be sent. Indications have the same requirement: a new indication cannot be sent before a confirmation for the previous indication is received by the server.

Requests and indications, however, are mutually exclusive in terms of the sequence requirement. So, an indication can be sent by the server before it responds to a request that it had received earlier.

Commands and notifications are different, and do not require any flow control — they can be sent at any time. Because of this — and because a server or client may not be able to handle these packets (due to buffer or processing limitations) — they are considered unreliable. When reliability is a concern, requests and indications should be used instead.

Reading Attributes

Reads are **requests** by nature since they require a response. There are different types of reads. Here we list the two

most important ones:

- **Read Request:** a simple request referencing the attribute to be read by its **handle**.
- **Read Blob Request:** similar to the read request but adds an offset to indicate where the read should start, returning a portion of the value. This type of read is used for reading only part of a characteristic's value.

Writing To Attributes

Writes can be either commands or requests. Here are the most common types of writes:

- **Write Request:** as the name suggests, this requires a response from the server to acknowledge that the attribute has been successfully written to.
- **Write Command:** this has no response from the server.
- **Queued Writes (atomic operation behavior):** these are classified as requests and require a Response from the server. They are used whenever a large value needs to be written and does not fit within a single message. Instead of writing parts of the value and risking someone else reading the incorrect (partial) value, two types of write requests are used to make sure the operation completes safely:
 - **One or more Prepare Write Requests:** each includes an offset at which the sent value should be written within the attribute value. The sent values are also referred to as **prepared values**, and they get stored in a buffer on the server side — not written to the attribute yet. This operation requires a response from the server.
 - **One Execute Write Request:** used to request from the server to either execute or cancel the Write operation of the prepared values. It requires a response from the server and once a response has been received from the server, the client can now be sure that the attribute holds the complete value it sent to the server.

Exchange MTU Request

The server and the client agree on a common value that is used for both data transfer directions. The client is the side that sends this exchange MTU request packet, and can only send it once per connection (per the Bluetooth specification, version 5.0, Vol 3, Part F, Section 3.4.2.1).

The server then responds with an exchange MTU response packet indicating the ATT_MTU it can support. The agreed-on value then becomes the minimum of the ATT_MTU values exchanged between the client and server.

It's important to know that different BLE stacks have different maximum values of ATT_MTU that they can support.

Designing your GATT

General Guidelines

While GATT is a pretty flexible framework, there are a few general guidelines to follow when designing it and creating the services and characteristics within it. Here are some recommendations:

- Make sure to implement the following mandatory service and its characteristics:
 - **Generic Access Profile (GAP) service.**
 - **Name and Appearance** characteristics within the GAP service.
- One thing to keep in mind is that vendor SDKs usually do not require you to explicitly implement this service, but rather they provide APIs for setting the **name** and **appearance**. The SDK then handles creating the GAP service and setting the characteristics according to the user-provided values.
- Utilize the Bluetooth SIG-adopted profiles, services, and characteristics in your design whenever possible. This has the following benefits:
 - You get the benefit of reducing the size of data packets involving UUIDs for services and characteristics (including advertisement packets, discovery procedures, and others) — since 16-bit UUID values are used instead of 128-bit values.
 - Bluetooth chipset and module vendors usually provide implementations of these profiles, services, and characteristics in their SDKs — reducing development time considerably.
 - Interoperability with other third-party devices and applications, allowing more devices to interface with your device and provide a richer user experience.
- Group characteristics that serve related functionality within a single service.
- Avoid having services with too many characteristics. A good separation of services makes it faster to discover certain characteristics and leads to a better GATT design that's modular and user-friendly.

In the next chapter, we'll go over a practical example showing how to design the GATT for our main project (BLE home automation system).

GATT Design Guidelines

Designing the GATT for your BLE device can be a challenge. To make this task easier, let's go through a complete exercise of designing the GATT for a simple BLE home automation system.

The home automation system is a hypothetical one, but one that will help you better understand the steps taken in designing the GATT of a real-life application, rather than some other generic, abstract system. Here's a diagram showing the different elements of the home automation system and how they interact with each other.

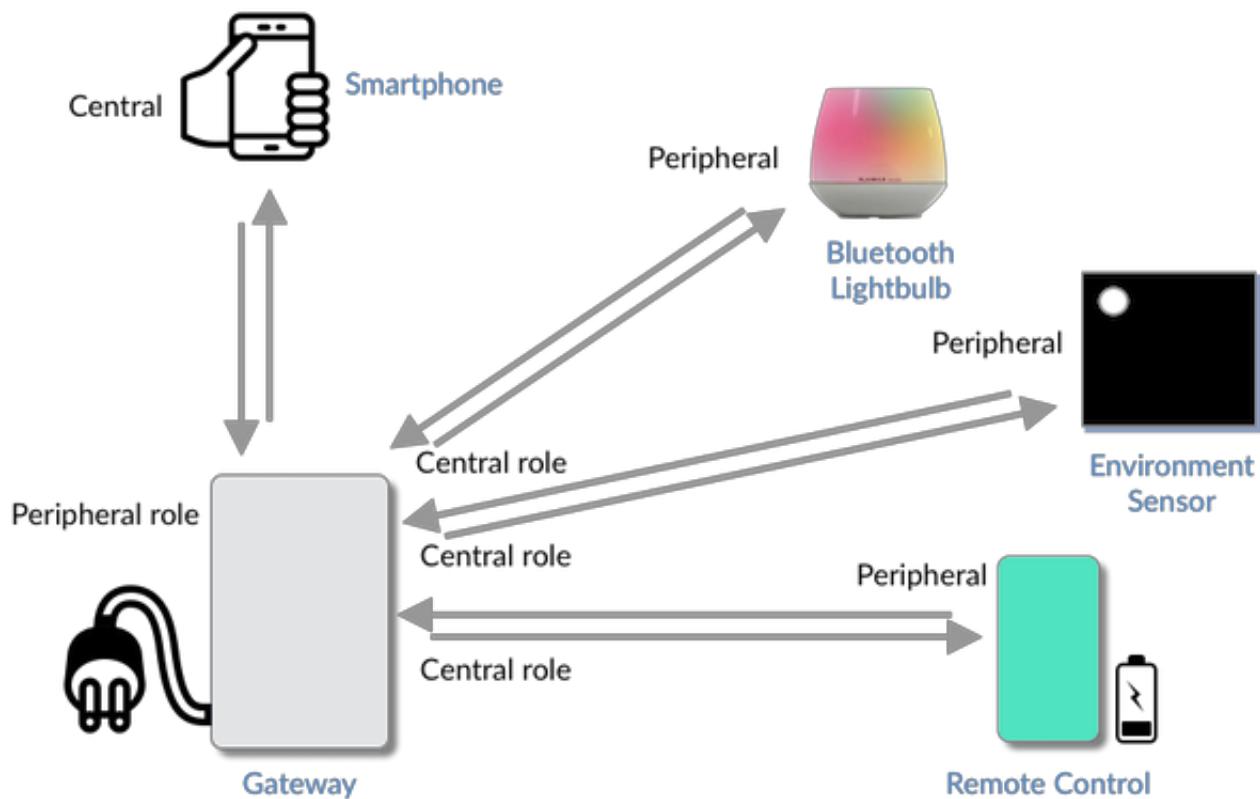


Figure 23: BLE home automation project example

The system consists of multiple elements (devices). Some of these are off-the-shelf components that we don't have control over, while others are devices whose firmware we do have control over and to which we will design their GATT structure.

General System Description

Let's go ahead and describe the main user scenarios of the system:

1. The homeowner can use the **remote control** to turn on/off the **Bluetooth lightbulb**.

2. The homeowner can monitor changes in the temperature and humidity of the **environment sensor**.
3. The homeowner is notified of the battery levels of the **remote control**, **Bluetooth lightbulb**, and **environment sensor**.

System Elements

Now, let's go over the different elements within the system.

1. Gateway

The **gateway** will act as a BLE central when communicating with all the other devices except the smartphone, where it will act as a peripheral. We have control over this device and we will be designing its GATT.

The commands for controlling the Bluetooth lightbulb will be routed from the remote control through the gateway and to the Bluetooth lightbulb.

2. Remote Control

The **remote control** is a device that will act as a peripheral only, and one that we will be designing the GATT for.

3. Environment Sensor

This is an off-the-shelf device over whose GATT design we have no control, so we will simply be interested in reading the data from it (temperature and humidity readings). In my case, I am placing it in my garage area.

4. Bluetooth Lightbulb

This is another off-the-shelf device over whose GATT we have no control.

5. Smartphone

This is also another existing device over whose GATT we have no control.

GATT Design

Let's go through the GATT design process, step-by-step:

Step 1: Documenting the Different User Scenarios and Data Points

Even though the GATT is usually more focused on the peripheral role (since a peripheral is usually the server exposing the data), the central can still act as the server in some cases for specific data points it needs to expose. Also, since we are designing both sides of the system (peripheral and central on the gateway), it helps to think in terms of what needs to happen from each side since this could affect some aspects of the system and GATT design.

Gateway

The gateway device acts in both the central and peripheral role. Each of these roles are used to enable communication with different devices within the system. The main purpose of the gateway is to act as a central device to read data from multiple peripherals. It then exposes this information via the peripheral role to another central device (the smartphone) that is able to relay this data to a **cloud server**. Let's go through the user scenarios from the

gateway's perspective for each of the central and peripheral roles.

Peripheral Role

- The remote control notifies the gateway when specific buttons are pressed to turn on/off the Bluetooth lightbulb.
- The following data points within the system need to be reported up to a cloud server via the gateway. These data points get exposed as a GATT Server in the peripheral role to the central device (the smartphone) that has an Internet connection (which allows it to relay this data up to the cloud server).
 - Environment sensor temperature reading
 - Environment sensor humidity reading
 - Bluetooth lightbulb status (on/off)
 - Individual battery levels for the environment sensor, Bluetooth lightbulb, and the remote control.

Central Role

The gateway device needs to read some of the data exposed by devices within the system and get notified of other data points exposed by these devices.

Remote Control

The remote control provides one main function: turning the Bluetooth lightbulb on or off. It acts strictly in the peripheral role and needs to expose the following data points:

- **On button press:** the gateway needs to be notified of this event when it occurs.
- **Off button press:** the gateway needs to be notified of this event when it occurs.
- **Battery level:** the gateway needs to be able to read this data and be notified when it changes.

Step 2: Define the Services, Characteristics, and Access Permissions

The next step is to group the **characteristics** into meaningful groups (**services**) based on their functionalities and define the access permissions for each of these characteristics.

Gateway

We have one GATT Server for the gateway, and it exists for the peripheral role. By looking at the data points (characteristics) we listed previously, we can group them into the following services:

- **Environment Sensor Service:**
 - Environment sensor temperature reading characteristic: "Temperature".
Access permissions: Read, notify.
 - Environment sensor humidity reading characteristic: "Humidity".
Access permissions: Read, notify.

- Battery level characteristic: "Battery Level".
Access permissions: Read, notify.
- **Playbulb Service:**
 - Light status characteristic: "Light Status".
Access permissions: Read, notify.
 - Battery level characteristic: "Battery Level".
Access permissions: Read, notify.
- **Remote Control Service:**
 - Battery level characteristic: "Battery Level".
Access permissions: Read, notify.

In addition to these services, it is mandatory (per the Bluetooth specification) to implement the following service:

- **GAP Service:**
 - Name characteristic: the device name.
Access: Read.
 - Appearance characteristic: a description of the device.
Access: Read.

Remote Control

We have one GATT server for the remote control. We can define the following services and characteristics:

- **GAP Service (mandatory):**
 - Device name characteristic: "Device Name".
Access permissions: Read.
 - Appearance characteristic.
Access permissions: Read.
- **Battery Service:**
 - Battery level characteristic : "Battery Level".
Access: Read, notify.
- **Button Service:**
 - On button characteristic: "On Button Press".
Access permissions: Notify.
 - Off button characteristic: "Off Button Press".
Access permissions: Notify.

Step 3: Re-use Bluetooth SIG-Adopted Services & Characteristics

Gateway

The environment sensor, Bluetooth lightbulb, and remote control services are all custom services, since there are no

SIG-adopted ones that can be utilized for them. We have three devices for which we need to expose the battery level, so we can reuse the SIG-adopted battery level characteristic. We will reuse it for each device within each device's service in the gateway GATT. We'll also re-use the mandatory GAP service.

Remote Control

For the **remote control**, we can reuse both the battery service and the mandatory GAP service.

Step 4: Assign UUIDs to Custom Services and Characteristics

For any **custom services and characteristics** within the GATT, we can use an online tool to generate UUIDs such as the [Online GUID Generator](#).

A common practice is to choose a base UUID for the custom service and then increment the **3rd and 4th Most Significant Bytes (MSB)** within the UUID of each included characteristic.

For example, we could choose the UUID: 00000001-1000-2000-3000-111122223333 for a specific service and then 0000000[N]-1000-2000-3000-111122223333, (where **N > 1**) for each of its characteristics.

The only restriction for choosing UUIDs for custom services and characteristics is that they must not collide with the Bluetooth SIG base UUID:

XXXXXXXX-0000-1000-8000-00805F9B34FB

However, following the previously mentioned common practice where the service and its characteristics employ sequential UUIDs makes it a bit easier to relate services and their characteristics to one another. The following tables list the services and characteristics along with their UUIDs for each of the gateway and remote control devices:

Service	Characteristic	UUID	UUID Type
Environment Sensor		13BB0001-5884-4C5D-B75B-8768DE741149	Custom
	Temperature	13BB0002-5884-4C5D-B75B-8768DE741149	Custom
	Humidity	13BB0003-5884-4C5D-B75B-8768DE741150	Custom
	Battery level	0x2A19	SIG-adopted
Playbulb		19210001-D8A0-49CE-8038-2BE02F099430	Custom
	Light status	19210002-D8A0-49CE-8038-2BE02F099430	Custom
	Battery level	0x2A19	SIG-adopted
Remote Control		B49B0001-37C8-4E16-A8C4-49EA4536F44F	Custom
	Battery level	0x2A19	SIG-adopted
GAP		0x1800	SIG-adopted
	Device Name	0x2A00	SIG-adopted
	Appearance	0x2A01	SIG-adopted

Table 3: Gateway GATT design

Service	Characteristic	UUID	UUID Type
Button		E54B0001-67F5-479E-8711-B3B99198CE6C	Custom
	ON button press	E54B0002-67F5-479E-8711-B3B99198CE6C	Custom
	OFF button press	E54B0003-67F5-479E-8711-B3B99198CE6C	Custom
Battery		0x180F	SIG-adopted
	Battery level	0x2A19	SIG-adopted
GAP		0x1800	SIG-adopted
	Device Name	0x2A00	SIG-adopted
	Appearance	0x2A01	SIG-adopted

Table 4: Remote control GATT design

Step 5: Implement the Services and Characteristics Using the Vendor SDK APIs

Each platform, whether it's an embedded or mobile one, has its own APIs for implementing services and characteristics. This is left to the reader to implement for the specific platform they choose for their BLE device or application.

GAP Design Guidelines

As we've discussed before, there are a number of parameters that affect a device's behavior depending on which state it's in. We need to design the different parameters for each of these states (Advertising, Scanning, Connected) while keeping in mind the specific use case of our application.

For example, if we have a Peripheral that gathers time-sensitive data that it needs to relay to a Central, then we may want to make sure that Connections can be established very quickly (*short Advertising Interval*) and that we have a low latency in sending data during a Connection (*short Connection Interval*).

Now, consider the case where the Peripheral does not always have data it needs to send to the Central, and it also runs on a small battery. In this case, we can utilize the **Slave Latency** value and still keep a **short Connection Interval**. This allows the Peripheral to stay "asleep" when it doesn't have any data to transmit, but still be able to send data frequently — if necessary.

Here are the different BLE parameters that are based on the state of the device:

- **Advertising state parameters**
 - Advertising Interval
 - Advertising Timeout
 - Advertising Data & Data Size
- **Scanning state parameters**
 - Scanning Window
 - Scanning Interval
 - Scanning Timeout
 - Scanning Type
- **Connected state parameters**
 - Connection Interval (**minimum** and **maximum** values)

Minimum and maximum Connection Interval values are used in order to give the peer device flexibility when negotiating the Connection Parameters.
 - Slave Latency
 - Supervision Timeout
 - MTU Size
 - Amount of Data Sent

Central vs. Peripheral vs. Multi-Role

For devices that play only in the Peripheral Role (hence operating in only one State at any given time), the optimization of the relevant parameters can be relatively simple. One example is the **Remote Control** device in our **Main Project**. However, when you consider the case of a device that operates in multiple Roles, such as the **Gateway** device in our

Main Project (*acting as a Peripheral and a Central at the same time*), then the optimization and design of these parameters can get quite complex.

In BLE, Central devices have a lot more responsibility — in terms of data processing and timing management — than Peripherals do. This usually requires the Central to operate on larger batteries and have more processing power. Having said that, power consumption is still a concern for Centrals, but it is a much more critical factor for Peripherals — which usually run on smaller batteries, have a smaller footprint, and lack the processing power that Centrals have. In our GAP design exercise, we will focus on **power consumption optimization** for the Peripheral, and on **connection management/availability** and **connection stability** on the Central side.

GAP Parameters Design Exercise

In this exercise, we will go through the process of designing the different GAP parameters for our Main Project nRF-based devices: the *Remote Control* and the *Gateway*.

Peripheral-Only Use Case: The Remote Control

When designing the different parameters for a Peripheral device, we need to keep in mind what Central devices will connect to the Peripheral. This is important due to the fact that Centrals control all aspects of a Connection. The most common use case for a Central device is a **smartphone**.

However, in our case the Remote Control will be connected to via the Gateway (not directly by the end user), so we have control on both sides. Let's list the parameters that are relevant to the Remote Control:

- **Advertising state parameters**
 - Advertising Interval
 - Advertising Timeout
 - Advertising Data & Data Size
- **Connected state parameters**
 - Connection Interval
 - Slave Latency
 - Supervision Timeout
 - MTU Size
 - Amount of data being sent

Notice we did not include the Scanning state parameters, since the Remote Control is acting in the Peripheral Role only.

Advertising State Parameters

- Advertising Interval & Timeout: to reduce power consumption, we will choose to Advertise for a **very short** period (*Advertising Timeout = 30 seconds*). However, we will use a **short** Advertising Interval (*Advertising Interval =*

187.5 msec).

We don't expect the Remote Control to be Advertising often since it will normally be connected to the Gateway. If it loses the connection, it can go back to Advertising and then stop after the configured Advertising Timeout. If Advertising times out, we can utilize a button press to trigger the Advertising again.

- Advertising Data & Advertising Data Size: since the Remote Control is Advertising specifically for the Gateway device to discover it, we don't need to include any data other than the Local Name and Appearance. To optimize power consumption even more, we could shorten the Local Name string to reduce the amount of data being transmitted in the Advertising packet.

Connected State Parameters

In BLE, the Central dictates the Connection parameters. However, the Peripheral can still **suggest** to the Central a set of Connection parameters. When developing for nRF52 applications, the developer sets the "preferred" parameters and the SoftDevice takes care of "negotiating" them with the Central. Since we control the Central (the Gateway) firmware, we can make sure the parameters match.

- **Connection Interval & Slave Latency:** to reduce power consumption, we will choose a **short** Connection Interval along with a **non-zero** Slave Latency value.

Minimum Connection Interval = Maximum Connection Interval = 30 msec

Slave Latency = 9

- **Supervision Timeout:** this parameter does not affect power consumption as much, so we will choose the default value used in most nRF examples.

Supervision Timeout = 4,000 msec

- **MTU Size & Amount of Data Sent:** since we do not have a lot of data to transmit, we can stick with the default/minimum MTU size. This will also help reduce power consumption.

MTU Size = 23 (in bytes)

Multi-Role Use Case: The Gateway

In the case of the Gateway, we are not as concerned about power consumption (since it will be running on live power). However, we need to make sure that it can perform all the operations for the different Roles without sacrificing the stability and availability of Connections. We will address the parameters for each Role separately.

The Peripheral Role

In the Peripheral Role, we want to be able to connect to the Gateway from different devices, especially mobile devices. iOS and Android — the two most popular mobile device Operating Systems at the time of this writing — run different

chipsets and BLE stacks, so they (may) behave differently in terms of BLE device discovery and interaction. For the developer of a Peripheral device, this is a crucial aspect that needs to be taken into account.

Apple provides a strict guideline for designing BLE Peripherals that need to interface with iOS devices. Android, on the other hand, does not have any guidelines since it runs on many devices from different vendors (that may be also running on different Bluetooth chipsets). You can find the iOS guidelines listed in the [Bluetooth Accessory Design Guidelines](#) document. For our Gateway device, we will focus on designing the parameters to respect the iOS guidelines to allow connect-ability and connection stability. The parameters we will optimize:

- **Advertising Interval & Advertising Timeout:** following the iOS guidelines for the Advertising Interval shown here, we can use the suggested Advertising Interval and set the Timeout to zero.

Advertising Interval = 20 milliseconds

Advertising Timeout = 0 milliseconds

3.5 Advertising Interval

The advertising interval of the accessory should be carefully considered, because it affects the time to discovery and connect performance. For a battery-powered accessory, its battery resources should also be considered.

To be discovered by the Apple product, the accessory should first use the recommended advertising interval of 20 ms for at least 30 seconds. If it is not discovered within the initial 30 seconds, Apple recommends using one of the following longer intervals to increase chances of discovery by the Apple product:

- 152.5 ms
- 211.25 ms
- 318.75 ms
- 417.5 ms
- 546.25 ms
- 760 ms
- 852.5 ms
- 1022.5 ms
- 1285 ms

Figure 24: iOS advertising interval guidelines

(Source: Apple Bluetooth Accessory Design Guidelines)

- **Connection Interval, Slave Latency, Supervision Timeout:** here are the iOS guidelines for all connection parameters:

3.6 Connection Parameters

The accessory is responsible for the connection parameters used for the Low Energy connection. The accessory should request connection parameters appropriate for its use case by sending an L2CAP Connection Parameter Update Request at the appropriate time. See the *Bluetooth 4.0* specification, Volume 3, Part A, Section 4.20 for details.

The connection parameter request may be rejected if it does not comply with all of these rules:

- Slave Latency ≤ 30
- 2 seconds $\leq \text{connSupervisionTimeout} \leq 6$ seconds
- Interval Min modulo 15 ms $\equiv 0$
- Interval Min ≥ 15 ms
- One of the following:
 - Interval Min + 15 ms \leq Interval Max
 - Interval Min == Interval Max == 15 ms
- Interval Max * (Slave Latency + 1) ≤ 2 seconds
- Interval Max * (Slave Latency + 1) * 3 $<$ connSupervisionTimeout

Figure 25: iOS connection parameters guidelines
 (Source: Apple Bluetooth Accessory Design Guidelines)

Based on these guidelines, we will choose the following values for our Peripheral Preferred Connection Parameters:

- Minimum Connection Interval = Maximum Connection Interval = 15 milliseconds
- Slave Latency = 0
- Supervision Timeout = 4,000 ms
- MTU Size: since we do not have much data to send (mostly a few bytes for the Notifications), we will choose the default MTU Size.

MTU Size = 23 (bytes)

The Central Role

Since the Gateway operates in both the Peripheral and the Central role, we need to define the parameters that relate to scanning and connections as well.

Scanning Parameters

Power consumption is usually not a concern on the Central side (since it's usually connected to live power). This means we can scan more frequently and keep the radio on for longer periods of time without having to worry about battery life (at least in our Gateway implementation).

The three main scanning parameters are: the Scan Window, the Scan Interval, and the Scan Timeout.

We define the parameter values as following:

- Scan Window = 50 ms
- Scan Interval = 100 ms
- Scan Timeout = 0 (no timeout)

With this Scan Window and Scan Interval, we are basically scanning for half the time. We could increase it even more, but it may have an impact on the connections with other Peripherals and on its connection as a Peripheral to another Central device.

The other thing to keep in mind is that by our design of the Gateway, we stop scanning once a connection to all Peripherals are established (Thingy:52, Remote Control, and Playbulb candlelight).

Connection Parameters

As we've discussed before, the Central dictates the timing for the connections of all Peripherals connected to it. This means that the connection parameters play an important role on the Gateway side.

We define the different connection parameter values as following:

- Minimum Connection Interval = 15 ms
- Maximum Connection Interval = 15 ms
- Slave Latency = 9
- Supervision Timeout = 4,000 ms (4 seconds)

There's more flexibility with these parameters since we only have 3 Peripheral devices to main a connection with, so they can be relaxed a bit. In my testing, these values have proven to be sufficient for maintaining a connection with each of the Peripheral devices.

Summary

Defining your GAP Parameters is a very important step in the design of your BLE application, regardless of the device's Role. However, in the end you need to perform "real-life" tests to make sure the parameters chosen are satisfactory. This is true when it comes to both power consumption and the user experience.

Note: One other aspect that adds to the complexity of optimizing the different GAP parameters is the BLE stack (the

software/firmware that implements the Bluetooth Low Energy specification) used. Usually, the stack has additional “*non BLE-specification*” parameters that need to be configured — making the task even more difficult. For more information on nRF52 and SoftDevice configurations and limitations, refer to the Scheduling section of the S140 SoftDevice located [here](#).

GitHub Repository

This book is accompanied by a GitHub repository that contains the full source code for all examples used in the exercises. It also contains other useful files that can be used to learn more about Bluetooth Low Energy (such as the Ellisys Tracker capture files). The GitHub repository is located at this link:

<https://github.com/mafaneh/ble-ebook-project>

You can download the repository or clone it by clicking on the button highlighted in the following screenshot:

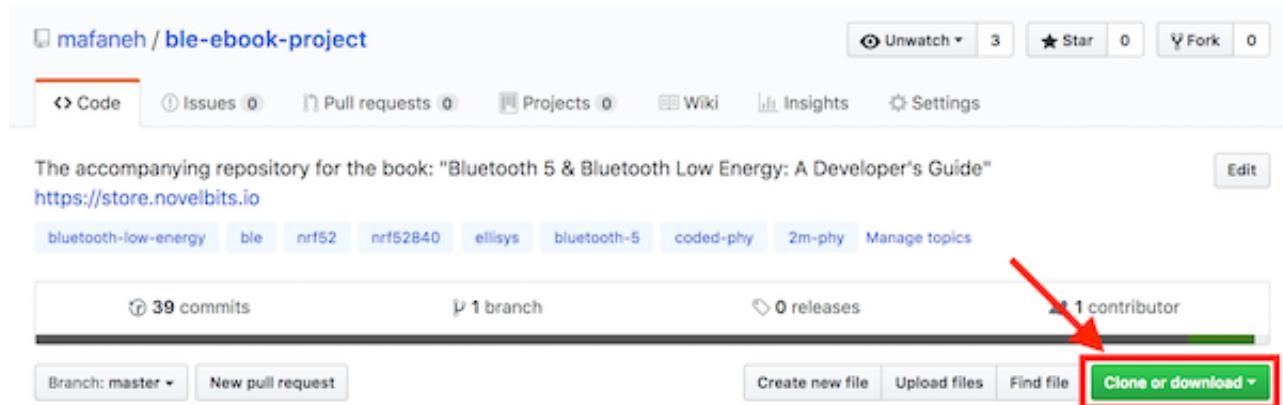


Figure 24: GitHub repository Clone/Download button

Introduction to the nRF5 Software Development Kit

In this chapter, we will go over some general concepts and information on the development platform used for the book's project: the nRF52 series and the nRF5x SDK.

SoftDevice

From [Nordic's SoftDevice InfoCenter documentation](#):

A SoftDevice is a wireless protocol stack library for building System on Chip (SoC) solutions. SoftDevices are precompiled into a binary image and functionally verified according to the wireless protocol specification, so that all you have to think about is creating the application. The unique hardware and software framework provide runtime memory protection, thread safety, and deterministic real-time behavior. The Application Programming Interface (API) is declared in header files for the C programming language. These characteristics make the interface similar to a hardware driver abstraction where device services are provided to the application, in this case, a complete wireless protocol.

More specifically, from the S140 SoftDevice product specification:

The S140 SoftDevice is a Bluetooth® Low Energy Central and Peripheral protocol stack solution. The S140 SoftDevice supports running up to twenty connections concurrently, with an additional observer role and broadcaster role. The S140 SoftDevice integrates a Bluetooth Low Energy Controller and Host, and provides a full and flexible API for building Bluetooth Low Energy nRF52 System on Chip solutions.

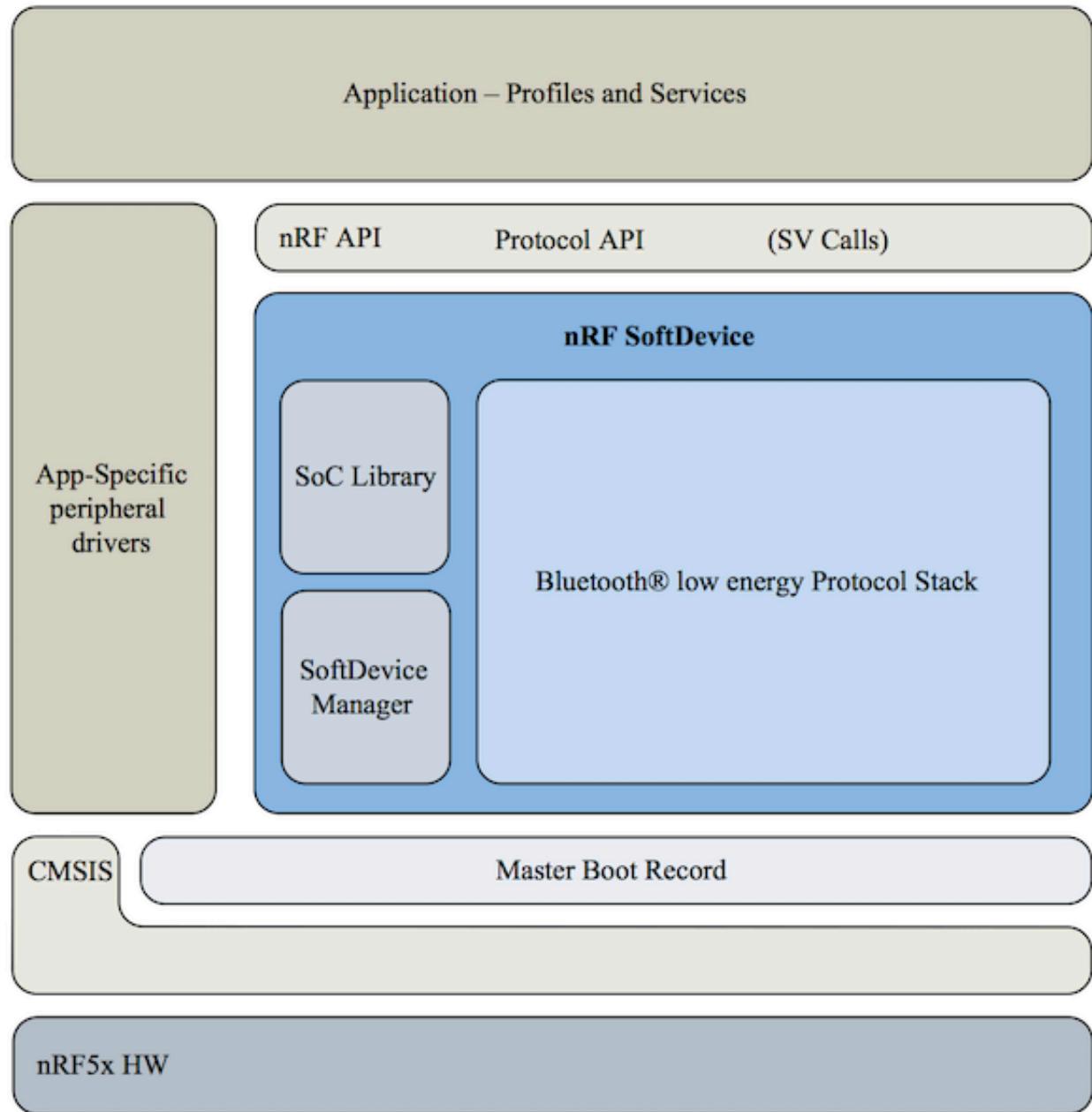
In this book, we will focus on the SoftDevice S140 since it applies to the development kit in the Main Project, the **nRF52840**. In the case of the nRF52832 development kit, the S132 is used instead. The two SoftDevices are very similar and it should not take much effort to adapt the provided project and source code to run on the nRF52832 development kit. The main difference between the two development kits, especially relating to Bluetooth Low Energy is the support for Bluetooth 5's Long-Range mode, which is only supported on the nRF52840.

Important Note

If you are planning on developing for the nRF52832 instead of the nRF52840 chipset, then refer to the section titled "["Migrating from nRF52840 to nRF52832"](#)" at the end of this chapter.

Architecture

Following is a diagram taken from [Nordic's Documentation on the SoftDevice](#):

*Figure 25: nRF5 SDK architecture*

We won't get too deep into the SoftDevice, and instead we'll focus on what applies to the application level and how we use the SoftDevice (via the APIs) to implement a BLE application. The application that you write will call specific APIs, initially to instantiate and initialize the Soft Device, and then to send data through the Soft Device API to the BLE stack. To receive data after initialization, your application will be "called-back" by the SoftDevice in response to different events (including BLE events). In this way, it is "event driven".

Important Note:

I highly recommend referring to the following links before continuing with this chapter. Otherwise, some of the content may feel like it has gaps and missing some (presumed) knowledge.

- [Nordic S140 SoftDevice Specification Documentation](#)
- [Nordic S140 SoftDevice v6.1.0 API](#)

From the S140 specification:

A SoftDevice consists of three main components:

- **SoC Library:** implementation and nRF Application Programming Interface (API) for shared hardware resource management (application coexistence)
- **SoftDevice Manager (SDM):** implementation and nRF API for SoftDevice management (enabling/disabling the SoftDevice, etc.)
- **Bluetooth 5 Low Energy protocol stack:** implementation of protocol stack and API

Here are some notable sections from the [SoftDevice specification document](#) that are important to refer to:

- **Chapter 4:** Application Programming Interface
- **Chapter 5:** SoftDevice Manager
- **Chapter 8:** Flash memory API
- **Chapter 10:** Bluetooth Low Energy protocol stack
- **Chapter 14:** SoftDevice memory usage
- **Chapter 17:** Bluetooth low energy data throughput
- **Chapter 18:** Bluetooth low energy power profiles

Since we will be using the SoftDevice stack in our example project and the other examples, the binary (pre-compiled) SoftDevice file has to be flashed to the board before our application binary is flashed. However, this is done automatically as part of the project building and flashing process, so we don't need to worry about it.

Development on the nRF52 Series (nRF5 SDK)

SDK Version

For the project developed throughout this book, we will be using the nRF5x SDK version **15.2.0** along with SoftDevice **S140 v6.1.0**.

Here are a few useful links:

- [nRF5 SDK Homepage](#)
- [S140 SoftDevice Homepage](#)
- [nRF5 SDK 15.2.0 download link](#)
- [SoftDevice S140 v6.1.0 download link](#)

Installing the SDK

The SDK and all its files are included with the project folder provided along the e-book Downloads and there is no need to download and install it. It is also included automatically by the SES Project files and requires no configuration in order to build the Gateway and Remote Control Projects (that is, when using the code provided in the GitHub repository).

SDK Configuration

The nRF5 SDK is configurable for a certain project via two main methods:

- The `sdk_config.h` file
- Adding/modifying compiler flags in the Project configuration

`sdk_config.h`

This file includes configurations such as:

- Flags to enable certain modules to be included in an application (e.g., Central role, Peripheral role, Battery Service Client, and more).
- Flags to configure and enable different hardware peripherals.
- Flags to configure and enable logging and debugging.
- Many others.

The file can be modified directly from the SES IDE Editor, using any generic text editor or via a plugin called the CMSIS Configuration Wizard (a Java-based add-on, explained later on in the chapter on Development Environment Setup).

Compiler Flags

Some examples of compiler flags defined for a project:

- **BLE_STACK_SUPPORT_REQD** (*enables the BLE stack*)
- **BOARD_PCA10056** (*defines which development board is used. nRF52840 Preview Development Kit is the PCA10056*)
- **NRF52840_XXAA** (*which chipset is used*)
- **NRF_SD_BLE_API_VERSION=6** (*version of the SoftDevice BLE API*)
- **S140** (*which SoftDevice is used. For nRF52840, the S140 is used*)
- **SOFTDEVICE_PRESENT** (*enables availability of the SoftDevice, specifically for memory allocation purposes*)
- For most (or all) of these settings, the default values stored in the Project configuration file will work just fine. Example applications included with the SDK each have their own SES Project configuration files appropriately configured for the application.

APIs, Libraries & Hardware Drivers

The nRF5 SDK provides us with APIs for interacting with the BLE functionality of the SoftDevice as well as the Hardware interfaces such as the clock, SPI, timers, ADC, etc. The SDK is split up into different modules called “Libraries” which can be enabled and configured via the `sdk_config.h` file we mentioned earlier. Some examples of the libraries include: BLE libraries, BLE services, the Bootloader, Board Support Package (BSP), Logger module, and a Cryptographic library. We will discuss selected libraries as we use them in the example project.

Logger Module

The Logger module enables us to print debug statements that help us track and inspect the state of the SoftDevice’s and the application at runtime. There are two **backend** interfaces that can be used by the Logger module: the **UART interface**, the **RTT interface**. In our example project, we will be using the RTT backend. RTT stands for RealTime Transfer, and it is a protocol developed by Segger (more on the protocol can be found [here](#)). It provides us with the following benefits:

- Very high transfer speed without affecting real-time behavior of the application.
- Supported by the J-Link module included in the development kits provided by Nordic.
- Bi-directional communication with the target device.

The RTT debug output will show up in the Debug Terminal window in Segger Embedded Studio while debugging. The Logger module is highly configurable. It allows different levels of debug output, different coloring for each application module and for each level of debug statements. Log messages can be processed in two different ways:

- In-place processing
- Deferred processing

In our application, we will configure the Logger module to use the deferred processing method. This mainly allows us to reduce any impact on real-time behavior and the timing of the application flow. The different levels of debug output are:

- ERROR (highest)
- WARNING
- INFO
- DEBUG (lowest)

Caution should be exercised with logging as it impacts the application performance. It is always good practice is to make sure log messages are output at the appropriate level to their function. For example, enabling output of large amounts of raw data should be done at the **DEBUG** level, whereas critical errors should be output with **ERROR** level. Each level of debug message logging can be executed with its own macro:

- **NRF_LOG_ERROR** for ERROR level debugging
- **NRF_LOG_WARNING** for WARNING level debugging
- **NRF_LOG_INFO** for INFO level debugging

- **NRF_LOG_DEBUG** for DEBUG level debugging

The SDK also allows us to define new logging modules for different parts of our application. This allows log messages to be prefaced with the name of the module from which these logs are called. It also allows us to define other attributes for each specific module such as: the highest level of debug messages that are output by the module, the colors of the different levels of messages.

We will be defining two new logging modules for the Gateway project: the Central logging module, and the Peripheral logging module. As for the Remote Control project, we will be adding one logging module: the Remote Control module. For example, to add our Central logging module, we would add the following to **each** file we want to include in this logging module. The one exception is the `NRF_LOG_MODULE_REGISTER()` call which should only be called once from one of the files.

```
#define NRF_LOG_MODULE_NAME Central
#ifndef CENTRAL_CONFIG_LOG_ENABLED
#define NRF_LOG_LEVEL CENTRAL_CONFIG_LOG_LEVEL
#define NRF_LOG_INFO_COLOR CENTRAL_CONFIG_INFO_COLOR
#define NRF_LOG_DEBUG_COLOR CENTRAL_CONFIG_DEBUG_COLOR
#else //CENTRAL_CONFIG_LOG_ENABLED
#define NRF_LOG_LEVEL 0
#endif //CENTRAL_CONFIG_LOG_ENABLED
#include "nrf_log.h"
NRF_LOG_MODULE_REGISTER();
```

BSP Library

The [Board Support Package \(BSP\) library](#) provides an abstraction for interfacing with the physical aspects of the Nordic development kits including buttons and LEDs.

We will be using this library only to define how the LEDs are used on the development kit board. For button press/release detection, we'll be using the [App Button library](#) instead.

Gateway

- Button assignments: None
- LEDs: we'll be using the [default states defined by the BSP module](#):

BSP state	PCA10040/PCA10056
BSP_INDICATE_IDLE	All LEDs are off
BSP_INDICATE_SCANNING	LED 1 is blinking (period 2 sec, duty cycle: 10%)
BSP_INDICATE_ADVERTISING	LED 1 is blinking (period 2 sec, duty cycle: 10%)
BSP_INDICATE_ADVERTISING_WHITELIST	LED 1 is blinking fast (period 1 sec, duty cycle: 20%)
BSP_INDICATE_ADVERTISING_SLOW	LED 1 is blinking slowly (period 4.4 sec, duty cycle: 10%)
BSP_INDICATE_ADVERTISING_DIRECTED	LED 1 is blinking very fast (period 0.6 sec, duty cycle: 50%)
BSP_INDICATE_BONDING	LED 1 is blinking (period 200 msec, duty cycle: 50%)
BSP_INDICATE_CONNECTED	LED 1 is on
BSP_INDICATE_SENT_OK	LED 2 is inverted for 100 msec
BSP_INDICATE_SEND_ERROR	LED 2 is inverted for 500 msec
BSP_INDICATE_RCV_OK	LED 2 is inverted for 100 msec
BSP_INDICATE_RCV_ERROR	LED 2 is inverted for 500 msec
BSP_INDICATE_FATAL_ERROR	All LEDs are on
BSP_INDICATE_USER_STATE_OFF	All LEDs are off
BSP_INDICATE_USER_STATE_0	LEDs 2, 3, and 4 are off, LED 1 is on
BSP_INDICATE_USER_STATE_1	LEDs 1, 3, and 4 are off, LED 2 is on
BSP_INDICATE_USER_STATE_2	LEDs 3 and 4 are off, LEDs 1 and 2 are on
BSP_INDICATE_USER_STATE_3	All LEDs are on
BSP_INDICATE_USER_STATE_ON	All LEDs are on

Figure 26: Default LED states BSP Module

Remote Control

- Button assignments:
 - Button 1: ON button that turns ON the Playbulb candle
 - Button 2: OFF button that turns OFF the Playbulb candle
- LEDs: we'll be using the [default states defined by the BSP module](#) (shown in the table above).

Application Template

Let's take a look at the general structure of an nRF52 application (taken from the `ble_app_template` example provided in the nRF5 SDK at the `examples/ble_peripheral/` folder):

```
/**@brief Function for application main entry.
 */
int main(void)
{
    bool erase_bonds;
```

```

// Initialize.
log_init();
timers_init();
buttons_leds_init(&erase_bonds);
ble_stack_init();
gap_params_init();
gatt_init();
advertising_init();
services_init();
conn_params_init();
peer_manager_init();

// Start execution.
NRF_LOG_INFO("Template example started.");
application_timers_start();

advertising_start(erase_bonds);

// Enter main loop.
for (;;)
{
    if (NRF_LOG_PROCESS() == false)
    {
        power_manage();
    }
}

```

Let's analyze each line of code to see how it all comes together:

`log_init();` defined as:

```

static void log_init(void)
{
    ret_code_t err_code = NRF_LOG_INIT(NULL);
    APP_ERROR_CHECK(err_code);

    NRF_LOG_DEFAULT_BACKENDS_INIT();
}

```

- `NRF_LOG_INIT()` initializes the Logger module.
 - `NRF_LOG_DEFAULT_BACKENDS_INIT()` sets the default backends for the Logger to use the enabled backends from `sdk_config.h` (UART and/or RTT)
-

`timers_init();` defined as:

```
/**@brief Function for the Timer initialization.
 *
 * @details Initializes the timer module. This creates and starts application timers.
 */
static void timers_init(void)
{
    // Initialize timer module.
    ret_code_t err_code = app_timer_init();
    APP_ERROR_CHECK(err_code);

    // Create timers.

    /* YOUR_JOB: Create any timers to be used by the application.
       Below is an example of how to create a timer.
       For every new timer needed, increase the value of the macro APP_TIMER_MAX_TIMERS by
       one.
       ret_code_t err_code;
       err_code = app_timer_create(&m_app_timer_id, APP_TIMER_MODE_REPEAT, timer_timeout_handler);
       APP_ERROR_CHECK(err_code); */
}
```

- Initializes the timer
- Checks the returned error code
- Provides some template code to create your own timer based on the application
- Timers here are generally used for non-BLE purposes such as detecting button presses, lighting up LEDs, and any other application-specific tasks you may have.

`buttons_leds_init(&erase_bonds);` defined as:

```
/**@brief Function for initializing buttons and leds.
 *
 * @param[out] p_erase_bonds Will be true if the clear bonding button was pressed to wake the
application up.
 */
static void buttons_leds_init(bool * p_erase_bonds)
{
    ret_code_t err_code;
    bsp_event_t startup_event;

    err_code = bsp_init(BSP_INIT_LED | BSP_INIT_BUTTONS, bsp_event_handler);
    APP_ERROR_CHECK(err_code);

    err_code = bsp_btn_ble_init(NULL, &startup_event);
    APP_ERROR_CHECK(err_code);
```

```
*p_erase_bonds = (startup_event == BSP_EVENT_CLEAR_BONDING_DATA);
}
```

- Takes in an argument that's enabled if the user holds down the button assigned to clear the Bonds during boot-up.
- `bsp_init(BSP_INIT_LED | BSP_INIT_BUTTONS, bsp_event_handler)` enables both the LEDs and the buttons via the Nordic BSP library. This assigns default behavior to the buttons and LEDs according to the BSP library documentation found [here]. It also assigns the event callback `bsp_event_handler` for BSP related events reported by the BSP library.
- `bsp_btn_ble_init(NULL, &startup_event)` handles specific button presses when called (during boot-up) by extracting the event and reporting it in `startup_event``. It also handles assigning the actions to the different buttons, defined as follows (from [this Nordic InfoCenter page]):

BSP BLE Button Assignments

This information applies to the following SoftDevices: S132, S140

Most BLE examples use the following standard button assignments as configured by the [BSP BLE Button Module](#):

During advertising or scanning:

- **Button 1:** Sleep (if not also in a connection)
- **Button 2 long push:** Turn off whitelist.

During sleep:

- **Button 1:** Wake up.
- **Button 2:** Wake up and delete bond information.

During connection:

- **Button 1 long push:** Disconnect.
- **Push and release on all buttons:** Application-specific.

Figure 27: BSP Button Assignments

- `*p_erase_bonds = (startup_event == BSP_EVENT_CLEAR_BONDING_DATA)` assigns the value of `p_erase_bonds` based on whether the specific button was pressed or not.

`ble_stack_init();` defined as:

```
/**@brief Function for initializing the BLE stack.
 *
 * @details Initializes the SoftDevice and the BLE event interrupt.
```

```
/*
static void ble_stack_init(void)
{
    ret_code_t err_code;

    err_code = nrf_sdh_enable_request();
    APP_ERROR_CHECK(err_code);

    // Configure the BLE stack using the default settings.
    // Fetch the start address of the application RAM.
    uint32_t ram_start = 0;
    err_code = nrf_sdh_ble_default_cfg_set(APP_BLE_CONN_CFG_TAG, &ram_start);
    APP_ERROR_CHECK(err_code);

    // Enable BLE stack.
    err_code = nrf_sdh_ble_enable(&ram_start);
    APP_ERROR_CHECK(err_code);

    // Register a handler for BLE events.
    NRF_SDH_BLE_OBSERVER(m_ble_observer, APP_BLE_OBSERVER_PRIO, ble_evt_handler, NULL);
}
```

- `nrf_sdh_enable_request()` enables the SoftDevice
- `nrf_sdh_ble_default_cfg_set(APP_BLE_CONN_CFG_TAG, &ram_start)` configures the stack with the default settings. It also returns the correct `ram_start` value that is needed for enabling the BLE stack.
- `nrf_sdh_ble_enable(&ram_start)` enables the BLE stack.
- `NRF_SDH_BLE_OBSERVER(m_ble_observer, APP_BLE_OBSERVER_PRIO, ble_evt_handler, NULL)` registers for BLE events.
- Keep in mind that `ble_evt_handler()` is a function defined by the application developer, and its job is to handle the different BLE events coming from the SoftDevice.

`gap_params_init();` defined as:

```
/**@brief Function for the GAP initialization.
 *
 * @details This function sets up all the necessary GAP (Generic Access Profile) parameters of the
 *          device including the device name, appearance, and the preferred connection parameters.
 */
static void gap_params_init(void)
{
    ret_code_t           err_code;
    ble_gap_conn_params_t   gap_conn_params;
    ble_gap_conn_sec_mode_t sec_mode;

    BLE_GAP_CONN_SEC_MODE_SET_OPEN(&sec_mode);

    err_code = sd_ble_gap_device_name_set(&sec_mode,
                                         (const uint8_t *)DEVICE_NAME,
```

```

        strlen(DEVICE_NAME));
APP_ERROR_CHECK(err_code);

/* YOUR_JOB: Use an appearance value matching the application's use case.
   err_code = sd_ble_gap_appearance_set(BLE_APPEARANCE_);
   APP_ERROR_CHECK(err_code); */

memset(&gap_conn_params, 0, sizeof(gap_conn_params));

gap_conn_params.min_conn_interval = MIN_CONN_INTERVAL;
gap_conn_params.max_conn_interval = MAX_CONN_INTERVAL;
gap_conn_params.slave_latency     = SLAVE_LATENCY;
gap_conn_params.conn_sup_timeout = CONN_SUP_TIMEOUT;

err_code = sd_ble_gap_ppcp_set(&gap_conn_params);
APP_ERROR_CHECK(err_code);
}

```

- `BLE_GAP_CONN_SEC_MODE_SET_OPEN(&sec_mode);` sets the `sec_mode` variable's permissions to be open with no security requirements. This variable is then used in the following API call to be set for the Device Name.
- `sd_ble_gap_device_name_set(...);` sets the Device Name and is passing the `sec_mode` variable as the Write Permission, meaning the Device Name can be modified by another (connected) device. (*Read permissions are always enabled for Device Name, so the API does not accept any changes to Read Permissions of the Device Name*).
- `sd_ble_gap_appearance_set(BLE_APPEARANCE_)` is commented out and left to the developer to enable with the proper Appearance type based on the application.
- `sd_ble_gap_ppcp_set(&gap_conn_params)` sets the Peripheral's Preferred Connection Parameters (PPCP) which get sent to the Central.

`gatt_init();` defined as:

```

/**@brief Function for initializing the GATT module.
 */
static void gatt_init(void)
{
    ret_code_t err_code = nrf_ble_gatt_init(&m_gatt, NULL);
    APP_ERROR_CHECK(err_code);
}

```

- `nrf_ble_gatt_init(&m_gatt, NULL)` initializes the GATT module allowing the application to add Services and Characteristics.

`advertising_init();` defined as:

```
/**@brief Function for initializing the Advertising functionality.
 */
static void advertising_init(void)
{
    ret_code_t          err_code;
    ble_advertising_init_t init;

    memset(&init, 0, sizeof(init));

    init.advdata.name_type          = BLE_ADVDATA_FULL_NAME;
    init.advdata.include_appearance = true;
    init.advdata.flags              = BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE;
    init.advdata.uuids_complete.uuid_cnt = sizeof(m_adv_uuids) / sizeof(m_adv_uuids[0]);
    init.advdata.uuids_complete.p_uuids = m_adv_uuids;

    init.config.ble_adv_fast_enabled = true;
    init.config.ble_adv_fast_interval = APP_ADV_INTERVAL;
    init.config.ble_adv_fast_timeout = APP_ADV_TIMEOUT_IN_SECONDS;

    init.evt_handler = on_adv_evt;

    err_code = ble_advertising_init(&m_advertising, &init);
    APP_ERROR_CHECK(err_code);

    ble_advertising_conn_cfg_tag_set(&m_advertising, APP_BLE_CONN_CFG_TAG);
}
```

- `ble_advertising_init(&m_advertising, &init)` initializes the Advertising module with the parameters defined prior in the `init` variable.
- The `init` data structure is where you configure the Advertising mode for your device. nRF52 SDK lists the following supported modes:

Advertising mode	Behavior
Directed	After disconnecting, the application immediately attempts to reconnect to the peer that was connected most recently. This advertising mode is very useful to stay connected to one peer and seamlessly recover from accidental disconnects. This is only allowed very briefly since it has a high chance of blocking other wireless traffic.
Directed Slow	After the initial "Directed" advertising burst, The application can continue advertising directly to its last peer, but with lower duty cycle compared to "Directed" above.
Fast	The application rapidly advertises to surrounding devices for a short time.
Slow	The application advertises to surrounding devices, but with a longer advertising interval than in fast advertising mode. This advertising mode conserves power and causes less traffic for other wireless devices that might be present. However, finding a device and connecting to it might take more time in slow advertising mode.
Idle	The application stops advertising.

Figure 28: Advertising types table

nRF5 SDK v14.2.0
Data Fields

ble_adv_modes_config_t Struct Reference
Advertising Module

Options for the different advertisement modes. [More...](#)

```
#include <ble_advertising.h>
```

Data Fields

bool	ble_adv_on_disconnect_disabled
bool	ble_adv_whitelist_enabled
bool	ble_adv_directed_enabled
bool	ble_adv_directed_slow_enabled
bool	ble_adv_fast_enabled
bool	ble_adv_slow_enabled
uint32_t	ble_adv_directed_slow_interval
uint32_t	ble_adv_directed_slow_timeout
uint32_t	ble_adv_fast_interval
uint32_t	ble_adv_fast_timeout
uint32_t	ble_adv_slow_interval
uint32_t	ble_adv_slow_timeout

Figure 29: Advertising types struct

- More information can be found here at Nordic's InfoCenter:
 - [BLE Advertising Module documentation](#)

- [BLE Advertising Module API documentation](#)
 - `ble_advertising_conn_cfg_tag_set(&m_advertising, APP_BLE_CONN_CFG_TAG)` keeps track of which Connection Parameters get used if the device ends up connecting as a result of its Advertising Events. (*Tag is an nRF-specific term that's not BLE-related*).
-

`services_init();` defined as:

```
/**@brief Function for initializing services that will be used by the application.
 */
static void services_init(void)
{
    /* YOUR_JOB: Add code to initialize the services used by the application.
     *           ret_code_t           err_code;
     *           ble_xxs_init_t        xxs_init;
     *           ble_yys_init_t        yys_init;

     // Initialize XXX Service.
     memset(&xxs_init, 0, sizeof(xxs_init));

     xxs_init.evt_handler          = NULL;
     xxs_init.is_xxx_notify_supported = true;
     xxs_init.ble_xx_initial_value.level = 100;

     err_code = ble_bas_init(&m_xxs, &xxs_init);
     APP_ERROR_CHECK(err_code);

     // Initialize YYY Service.
     memset(&yys_init, 0, sizeof(yys_init));
     yys_init.evt_handler          = on_yys_evt;
     yys_init.ble_yy_initial_value.counter = 0;

     err_code = ble_yy_service_init(&yys_init, &yy_init);
     APP_ERROR_CHECK(err_code);
    */
}
```

- This is where the code goes that makes your application special. You'll include and initialize standard Services you need in your application as well as any custom Services you build. It is left to the developer to define and write its content based on the Services to be added.
-

`conn_params_init();` defined as:

```
/**@brief Function for initializing the Connection Parameters module.
```

```
/*
static void conn_params_init(void)
{
    ret_code_t           err_code;
    ble_conn_params_init_t cp_init;

    memset(&cp_init, 0, sizeof(cp_init));

    cp_init.p_conn_params          = NULL;
    cp_init.first_conn_params_update_delay = FIRST_CONN_PARAMS_UPDATE_DELAY;
    cp_init.next_conn_params_update_delay = NEXT_CONN_PARAMS_UPDATE_DELAY;
    cp_init.max_conn_params_update_count = MAX_CONN_PARAMS_UPDATE_COUNT;
    cp_init.start_on_notify_cccd_handle = BLE_GATT_HANDLE_INVALID;
    cp_init.disconnect_on_fail      = false;
    cp_init.evt_handler            = on_conn_params_evt;
    cp_init.error_handler          = conn_params_error_handler;

    err_code = ble_conn_params_init(&cp_init);
    APP_ERROR_CHECK(err_code);
}
```

- `ble_conn_params_init(&cp_init)` sets the connection parameters based on the initialized `cp_init` variable. These are used by the Peripheral to request an update to the Connection Parameters from the Central. To learn more about this, refer to [this section of the Nordic InfoCenter](#).

`peer_manager_init();` defined as:

```
/**@brief Function for the Peer Manager initialization.
*/
static void peer_manager_init(void)
{
    ble_gap_sec_params_t sec_param;
    ret_code_t           err_code;

    err_code = pm_init();
    APP_ERROR_CHECK(err_code);

    memset(&sec_param, 0, sizeof(ble_gap_sec_params_t));

    // Security parameters to be used for all security procedures.
    sec_param.bond          = SEC_PARAM_BOND;
    sec_param.mitm          = SEC_PARAM_MITM;
    sec_param.lesc          = SEC_PARAM_LESC;
    sec_param.keypress       = SEC_PARAM_KEYPRESS;
    sec_param.io_caps        = SEC_PARAM_IO_CAPABILITIES;
    sec_param.oob            = SEC_PARAM_OOB;
    sec_param.min_key_size   = SEC_PARAM_MIN_KEY_SIZE;
    sec_param.max_key_size   = SEC_PARAM_MAX_KEY_SIZE;
```

```

sec_param.kdist_own.enc  = 1;
sec_param.kdist_own.id   = 1;
sec_param.kdist_peer.enc = 1;
sec_param.kdist_peer.id  = 1;

err_code = pm_sec_params_set(&sec_param);
APP_ERROR_CHECK(err_code);

err_code = pm_register(pm_evt_handler);
APP_ERROR_CHECK(err_code);
}

```

- `pm_init()` initializes the Peer Manager module.
 - `pm_sec_params_set(&sec_param)` sets the Security Requirements for the connection.
 - `pm_register(pm_evt_handler)` registers the event handler for any Peer Manager events. `pm_evt_handler` is provided and implemented by the application.
 - The Peer Manager module is used to handle Security-related (Bonding, Pairing, Whitelisting, and others).
 - You can learn more about it at Nordic's InfoCenter [section on Peer Manager](#).
-

`application_timers_start();` defined as:

```

/**@brief Function for starting timers.
 */
static void application_timers_start(void)
{
    /* YOUR_JOB: Start your timers. below is an example of how to start a timer.
       ret_code_t err_code;
       err_code = app_timer_start(m_app_timer_id, TIMER_INTERVAL, NULL);
       APP_ERROR_CHECK(err_code); */

}

```

- Left to the developer to start any timers the application needs (such as reading from a sensor, measuring battery level, etc.).
-

`advertising_start(erase_bonds);` defined as:

```

/**@brief Function for starting advertising.
 */
static void advertising_start(bool erase_bonds)
{

```

```

if (erase_bonds == true)
{
    delete_bonds();
    // Advertising is started by PM_EVT_PEERS_DELETED_SUCCEEDED eventnt
}
else
{
    ret_code_t err_code = ble_advertising_start(&m_advertising, BLE_ADV_MODE_FAST);

    APP_ERROR_CHECK(err_code);
}
}

```

- `delete_bonds()` deletes any previously stored Bonds with other devices based on the if-condition when `erase_bonds` is enabled.
- `ble_advertising_start(&m_advertising, BLE_ADV_MODE_FAST)` starts Advertising in Fast Advertising mode.

```

// Enter main loop.
for (;;)
{
    if (NRF_LOG_PROCESS() == false)
    {
        power_manage();
    }
}

```

- `NRF_LOG_PROCESS()` processes any log messages only when the device is in Idle mode (not processing any events).
- When there are no log messages left to be processed, the device calls `power_manage()` which puts the application in “sleep” mode until an event occurs.

Migrating an Application From nRF52840 to nRF52832

Here are the steps needed to migrate an SES project from using the nRF52840 to the nRF52832 development kit:

- Replace the `flash_placement.xml` file in your project with a copy from an example for the nRF52832 in the nRF5 SDK. Here are the contents for the nRF52832 `flash_placement.xml` file:

```

<!DOCTYPE Linker_Placement_File>
<Root name="Flash Section Placement">
    <MemorySegment name="FLASH" start="$(FLASH_PH_START)" size="$(FLASH_PH_SIZE)">
        <ProgramSection load="no" name=".reserved_flash" start="$(FLASH_PH_START)" size="$(FLASH_START)-$(FLASH_PH_START)" />

```

```

<ProgramSection alignment="0x100" load="Yes" name=".vectors" start="$(FLASH_START)" />
<ProgramSection alignment="4" load="Yes" name=".init" />
<ProgramSection alignment="4" load="Yes" name=".init_rodata" />
<ProgramSection alignment="4" load="Yes" name=".text" />
<ProgramSection alignment="4" keep="Yes" load="Yes" name=".sdh_soc_observers" inputsections="*$(SORT(.sdh_soc_observers*))" address_symbol="__start_sdh_soc_observers" end_symbol="__stop_sdh_soc_observers" />
<ProgramSection alignment="4" keep="Yes" load="Yes" name=".pwr_mgmt_data" inputsections="*$(SORT(.pwr_mgmt_data*))" address_symbol="__start_pwr_mgmt_data" end_symbol="__stop_pwr_mgmt_data" />
<ProgramSection alignment="4" keep="Yes" load="Yes" name=".sdh_ble_observers" inputsections="*$(SORT(.sdh_ble_observers*))" address_symbol="__start_sdh_ble_observers" end_symbol="__stop_sdh_ble_observers" />
<ProgramSection alignment="4" keep="Yes" load="Yes" name=".sdh_stack_observers" inputsections="*$(SORT(.sdh_stack_observers*))" address_symbol="__start_sdh_stack_observers" end_symbol="__stop_sdh_stack_observers" />
<ProgramSection alignment="4" keep="Yes" load="Yes" name=".sdh_req_observers" inputsections="*$(SORT(.sdh_req_observers*))" address_symbol="__start_sdh_req_observers" end_symbol="__stop_sdh_req_observers" />
<ProgramSection alignment="4" keep="Yes" load="Yes" name=".sdh_state_observers" inputsections="*$(SORT(.sdh_state_observers*))" address_symbol="__start_sdh_state_observers" end_symbol="__stop_sdh_state_observers" />
<ProgramSection alignment="4" keep="Yes" load="Yes" name=".nrf_queue" inputsections="*$(.nrf_queue*)" address_symbol="__start_nrf_queue" end_symbol="__stop_nrf_queue" />
<ProgramSection alignment="4" keep="Yes" load="Yes" name=".nrf_balloc" inputsections="*$(.nrf_balloc*)" address_symbol="__start_nrf_balloc" end_symbol="__stop_nrf_balloc" />
<ProgramSection alignment="4" keep="Yes" load="Yes" name=".cli_command" inputsections="*$(.cli_command*)" address_symbol="__start_cli_command" end_symbol="__stop_cli_command" />
<ProgramSection alignment="4" keep="Yes" load="Yes" name=".crypto_data" inputsections="*$(SORT(.crypto_data*))" address_symbol="__start_crypto_data" end_symbol="__stop_crypto_data" />
<ProgramSection alignment="4" keep="Yes" load="Yes" name=".log_const_data" inputsections="*$(SORT(.log_const_data*))" address_symbol="__start_log_const_data" end_symbol="__stop_log_const_data" />
<ProgramSection alignment="4" keep="Yes" load="Yes" name=".log_backends" inputsections="*$(SORT(.log_backends*))" address_symbol="__start_log_backends" end_symbol="__stop_log_backends" />
<ProgramSection alignment="4" keep="Yes" load="No" name=".nrf_sections" address_symbol="__start_nrf_sections" />
<ProgramSection alignment="4" keep="Yes" load="Yes" name=".cli_sorted_cmd_ptrs" inputsections="*$(.cli_sorted_cmd_ptrs*)" runin=".cli_sorted_cmd_ptrs_run"/>
<ProgramSection alignment="4" keep="Yes" load="Yes" name=".fs_data" inputsections="*$(.fs_data*)" runin=".fs_data_run"/>
<ProgramSection alignment="4" keep="Yes" load="Yes" name=".log_dynamic_data" inputsections="*$(SORT(.log_dynamic_data*))" runin=".log_dynamic_data_run"/>
<ProgramSection alignment="4" keep="Yes" load="Yes" name=".log_filter_data" inputsections="*$(SORT(.log_filter_data*))" runin=".log_filter_data_run"/>
<ProgramSection alignment="4" load="Yes" name=".dtors" />
<ProgramSection alignment="4" load="Yes" name=".ctors" />
<ProgramSection alignment="4" load="Yes" name=".rodata" />
<ProgramSection alignment="4" load="Yes" name=".ARM.exidx" address_symbol="__exidx_start" end_symbol="__exidx_end" />
<ProgramSection alignment="4" load="Yes" runin=".fast_run" name=".fast" />
<ProgramSection alignment="4" load="Yes" runin=".data_run" name=".data" />
<ProgramSection alignment="4" load="Yes" runin=".tdata_run" name=".tdata" />
</MemorySegment>
<MemorySegment name="RAM" start="$(RAM_PH_START)" size="$(RAM_PH_SIZE)">
```

```

<ProgramSection load="no" name=".reserved_ram" start="$(RAM_PH_START)" size="$(RAM_START)-$(RAM_PH_START)" />
<ProgramSection alignment="0x100" load="No" name=".vectors_ram" start="$(RAM_START)" address_symbol="__app_ram_start__"/>
<ProgramSection alignment="4" keep="Yes" load="No" name=".nrf_sections_run" address_symbol="__start_nrf_sections_run" />
<ProgramSection alignment="4" keep="Yes" load="No" name=".cli_sorted_cmd_ptrs_run" address_symbol="__start_cli_sorted_cmd_ptrs" end_symbol="__stop_cli_sorted_cmd_ptrs" />
<ProgramSection alignment="4" keep="Yes" load="No" name=".fs_data_run" address_symbol="__start_fs_data" end_symbol="__stop_fs_data" />
<ProgramSection alignment="4" keep="Yes" load="No" name=".log_dynamic_data_run" address_symbol="__start_log_dynamic_data" end_symbol="__stop_log_dynamic_data" />
<ProgramSection alignment="4" keep="Yes" load="No" name=".log_filter_data_run" address_symbol="__start_log_filter_data" end_symbol="__stop_log_filter_data" />
<ProgramSection alignment="4" keep="Yes" load="No" name=".nrf_sections_run_end" address_symbol="__end_nrf_sections_run" />
<ProgramSection alignment="4" load="No" name=".fast_run" />
<ProgramSection alignment="4" load="No" name=".data_run" />
<ProgramSection alignment="4" load="No" name=".tdata_run" />
<ProgramSection alignment="4" load="No" name=".bss" />
<ProgramSection alignment="4" load="No" name=".tbss" />
<ProgramSection alignment="4" load="No" name=".non_init" />
<ProgramSection alignment="4" size="__HEAPSIZE__" load="No" name=".heap" />
<ProgramSection alignment="8" size="__STACKSIZE__" load="No" place_from_segment_end="Yes" name=".stack" address_symbol="__StackLimit" end_symbol="__StackTop" />
<ProgramSection alignment="8" size="__STACKSIZE_PROCESS__" load="No" name=".stack_process" />
</MemorySegment>
</Root>

```

- Edit the SES project/solution file (*.emProject file) in a text editor of choice and make the following modifications:
 - Replace the following within the `arm_target_device_name` definition:
 - `nRF52840_xxAA` with `nRF52832_xxAA` .
 - `NRF52840_XXAA` with `NRF52832_XXAA` .
 - `S140` with `S132` .
 - Change `c_preprocessor_definitions="BOARD_PCA10056` to `c_preprocessor_definitions="BOARD_PCA10040` .
 - In the `c_user_include_directories` definition, replace:
 - `components/softdevice/s140/headers` with `components/softdevice/s132/headers` .
 - `components/softdevice/s140/headers/nrf52` with `components/softdevice/s132/headers/nrf52` .
 - Replace


```
debug_additional_load_file="..../nRF5_SDK_current/components/softdevice/s140/hex/s140_nrf52_6.1.0_softdevice.hex" with
debug_additional_load_file="..../nRF5_SDK_current/components/softdevice/s132/hex/s132_nrf52_6.1.0_softdevice.hex".
```

- Replace `debug_register_definition_file=".../nRF5_SDK_current/modules/nrfx/mdk/nrf52840.svd"` with `debug_register_definition_file=".../nRF5_SDK_current/modules/nrfx/mdk/nrf52.svd"`.
- Replace
 - `linker_section_placement_macros="FLASH_PH_START=0x0;FLASH_PH_SIZE=0x100000;RAM_PH_START=0x20000000;RAM_PH_SIZE=0x40000;FLASH_START=0x26000;FLASH_SIZE=0xda000;..."`
 - with
 - `linker_section_placement_macros="FLASH_PH_START=0x0;FLASH_PH_SIZE=0x80000;RAM_PH_START=0x20000000;RAM_PH_SIZE=0x10000;FLASH_START=0x26000;FLASH_SIZE=0x5a000;..."`
- Make sure you do not change the portion of `linker_section_placement_macros` at the end which lists RAM_start and RAM_size (*indicated by "..." above*). Those can stay the same.
- Replace `linker_section_placements_segments="FLASH RX 0x0 0x100000;RAM RWX 0x20000000 0x40000"` with `linker_section_placements_segments="FLASH RX 0x0 0x80000;RAM RWX 0x20000000 0x10000"`.
- In the file listings, change:
 - `ses_startup_nrf52840.s` to `ses_startup_nrf52.s`.
 - `system_nrf52840.c` to `system_nrf52.c`.

Development Environment Setup

Segger Embedded Studio (SES)

Segger Embedded Studio (SES) is a professional cross-platform IDE for ARM Core microcontrollers. Segger is a well-known company in the embedded space that provides software, hardware, and development tools for embedded systems. In our project, we will be working with the Nordic nRF52 microcontroller and development kit.

Segger Embedded Studio has many advantages over other IDEs in the space, especially when used for nRF52 development. Some of which are:

- A recently announced partnership between Nordic Semiconductor and Segger provides a FREE commercial and unlimited license for nRF51 and nRF52 series SoCs development. To learn more, refer to [this announcement](#).
- It provides a turnkey solution for development on the nRF52 series platform with tight and seamless integration for on-target (on the device) debugging using J-Link.
- It has integrated GCC C/C++ and Clang/LLVM compilers.
- It provides cross-platform support: runs on Windows, Linux and macOS.
- There are no restrictions on code size and compiler optimization
- It integrates analysis tools for memory use, static code analysis and more.
- It has tools for importing projects from other IDEs such as Keil and IAR.

For nRF52 developers, this is the most cost-efficient solution and provides features that compete with other professional IDEs such as ARM Keil and IAR Embedded Workbench for ARM (EWARM) IDE.

Steps for Setup

Here are the steps to set up SES for our project:

- Download Segger Embedded Studio (SES) at: <https://www.segger.com/downloads/embedded-studio/>
- Pick the download that matches your platform (look for the Embedded Studio for ARM version).
- Launch the downloaded install file, and follow the steps for installation.
- There's no need to download and install the nRF5x SDK - it's already included in GitHub folder along with the SES Projects provided with the e-book.nRF5x SDK used: **nRF5_SDK_15.2.0**
- Download, or clone the GitHub project and place the folder as close to the root folder of your computer as possible refer to the chapter titled "**The Accompanying GitHub Repository**".

- After download and installation, launch SES.
- Enable showing all toolbars in SES to have a better user interface and to be able to perform operations more efficiently:

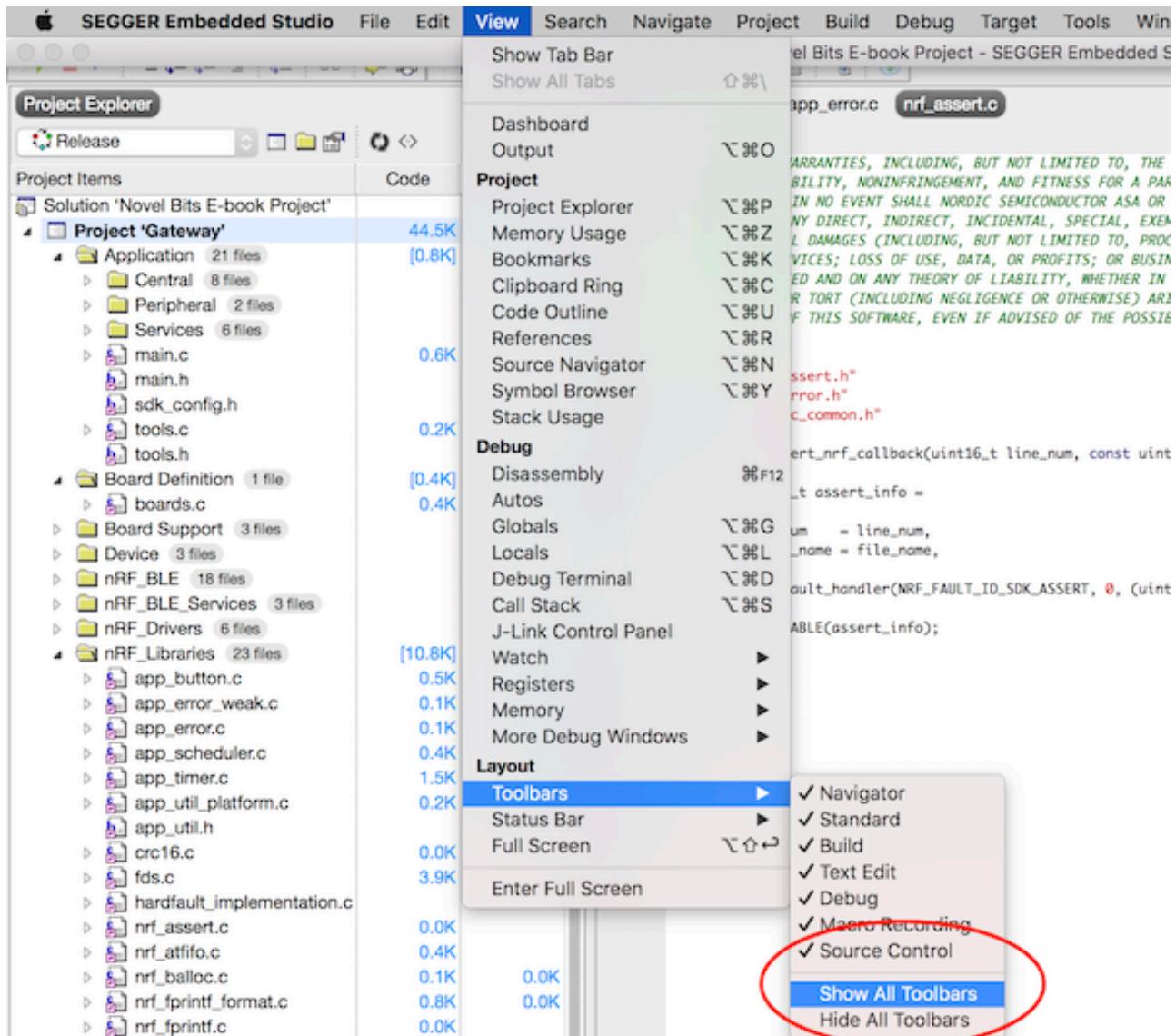


Figure 30: Show All Toolbars (SES)

- Open the SES Solution file from the (previously downloaded) GitHub folder located at (./Main Project SES files/NovelBits.emProject):

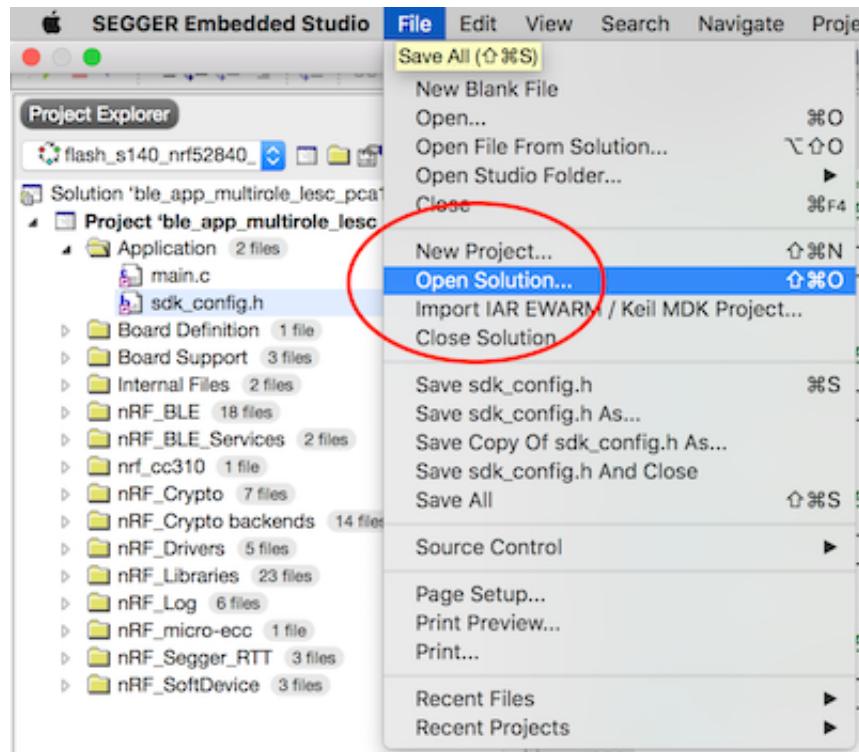


Figure 31: Open Solution (SES)

- We will attempt to build the Solution next (right-click on the “Solution” name in the Project Explorer, then click Build):

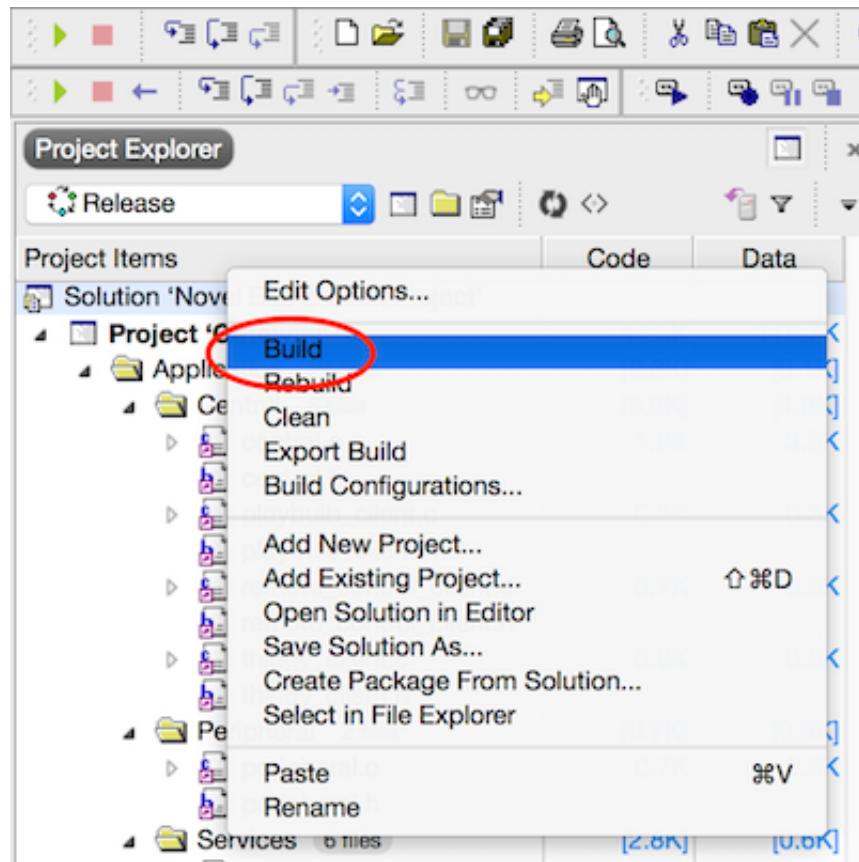


Figure 32: Build Solution (SES)

- If you haven't yet activated your FREE license for nRF development, you will be prompted to get a license:

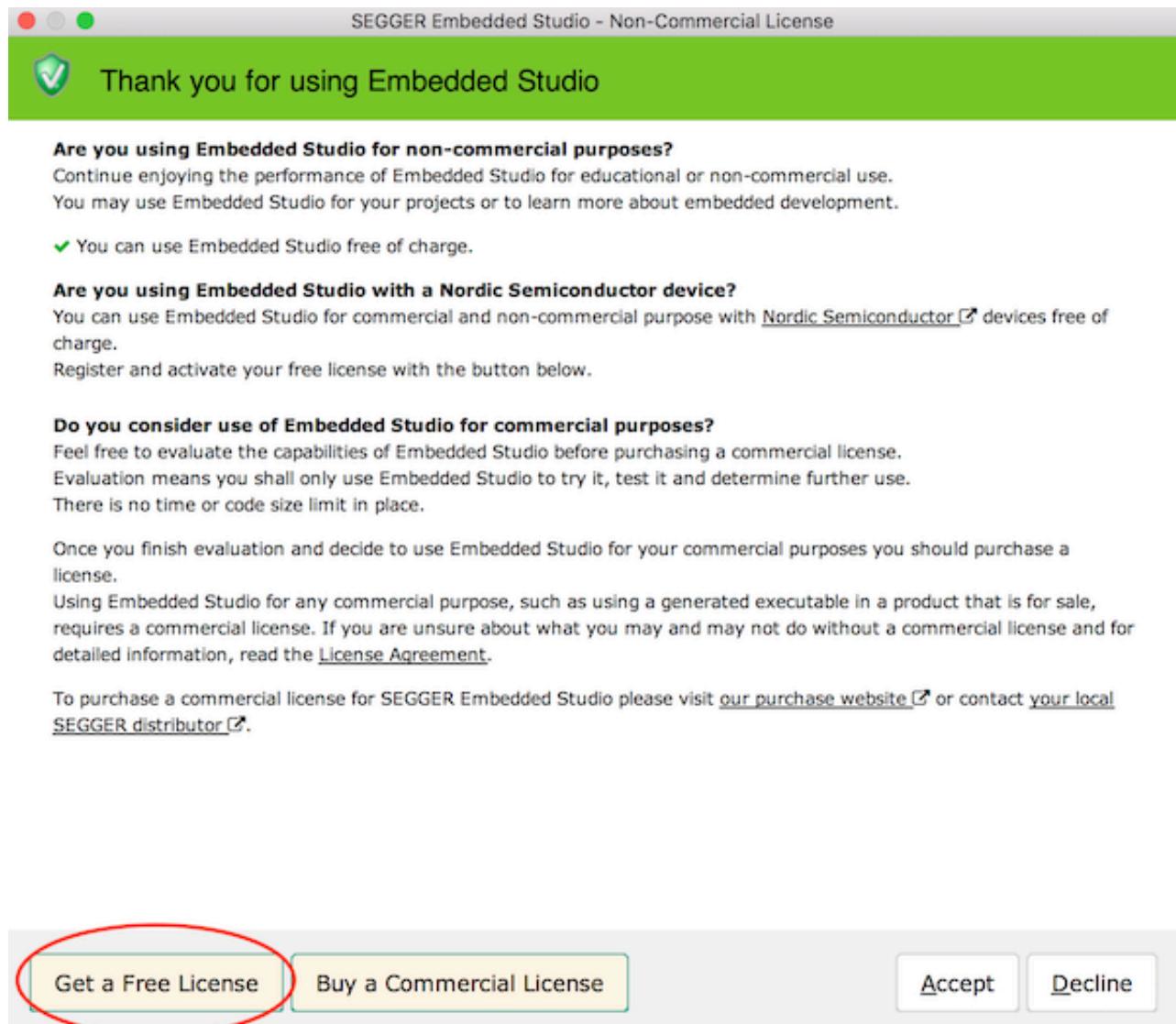


Figure 33: Free License (SES)

- Choose the "Get a Free License" option

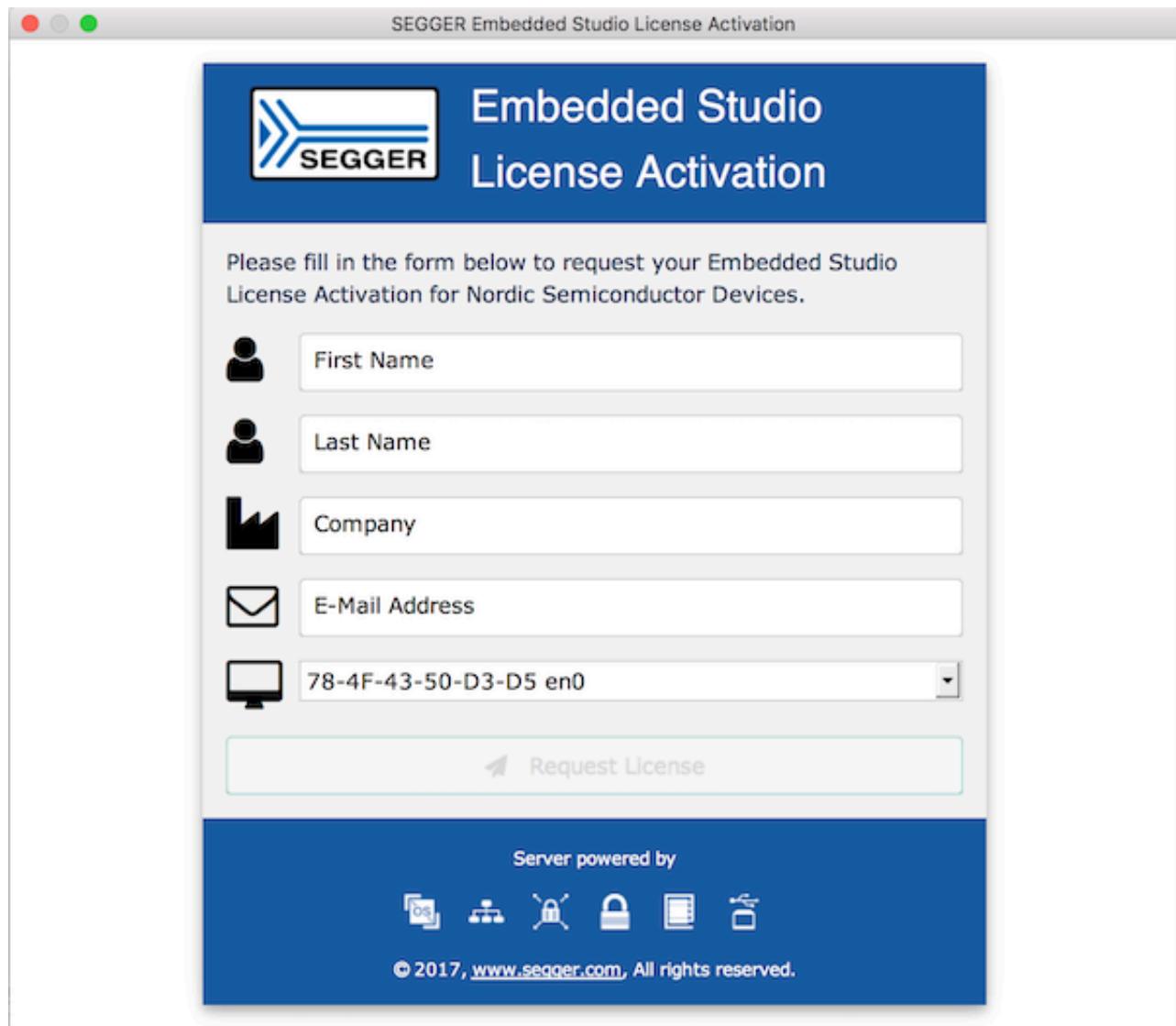


Figure 34: Get Free License (SES)

- Once you have filled out the information needed and clicked “Request License”, you will receive a license activation key in your email.
- Copy/paste the license key you received:

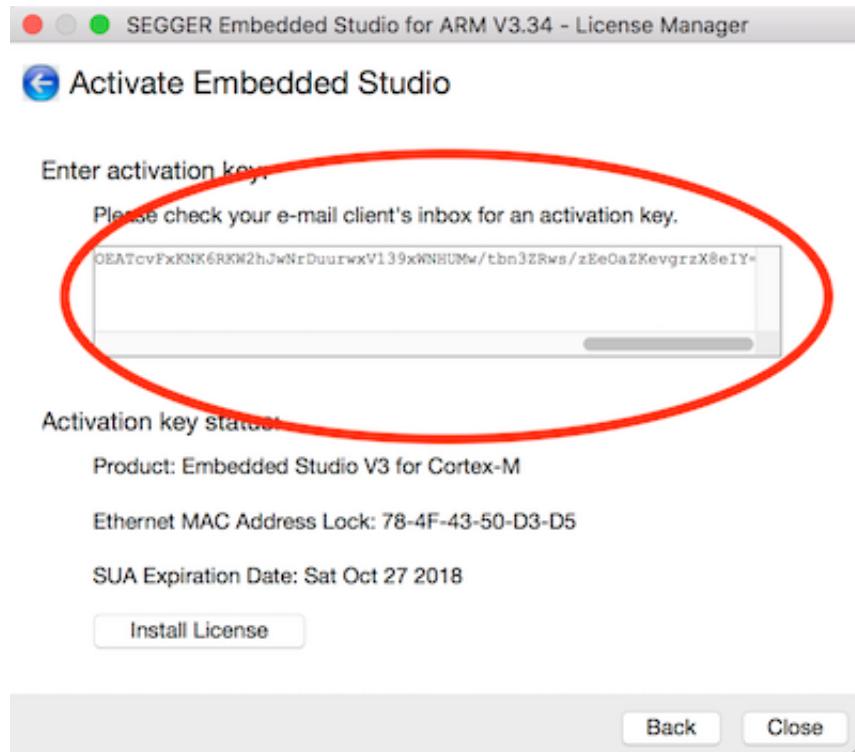


Figure 35: Activate License (SES)

- Once completed, the solution will start building
- If you run into issues obtaining the license, check out the following YouTube videos:
 - [SEGGER Embedded studio – Getting started](#)
 - [SEGGER Embedded Studio – Requesting license in browser](#)
- To build a specific Project (**Remote Control** or **Gateway**), right click the Project name in the Project Explorer, set Active Project, then you can use the “Build” button.

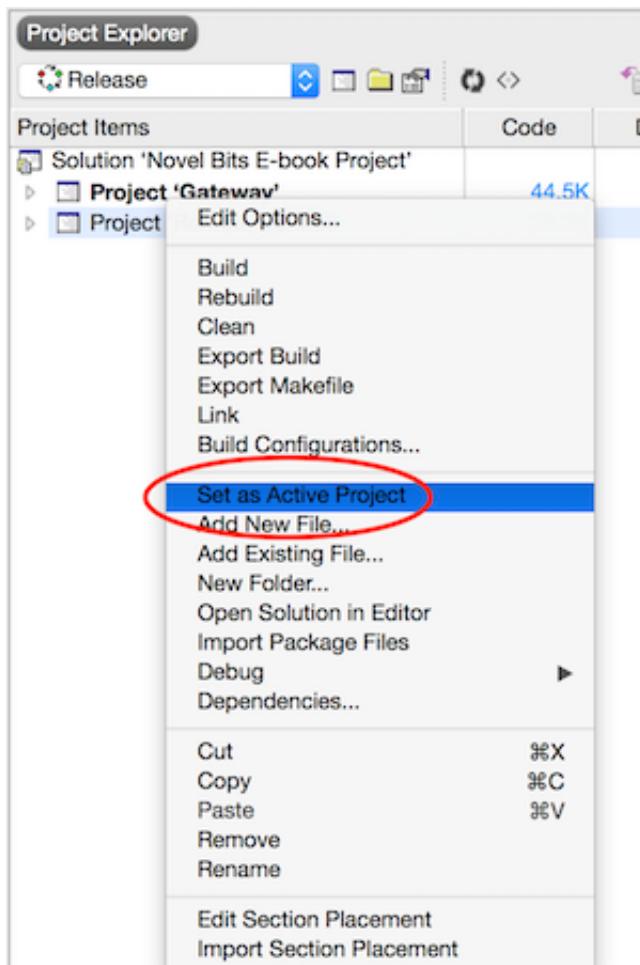


Figure 36: Set as Active Project (SES)

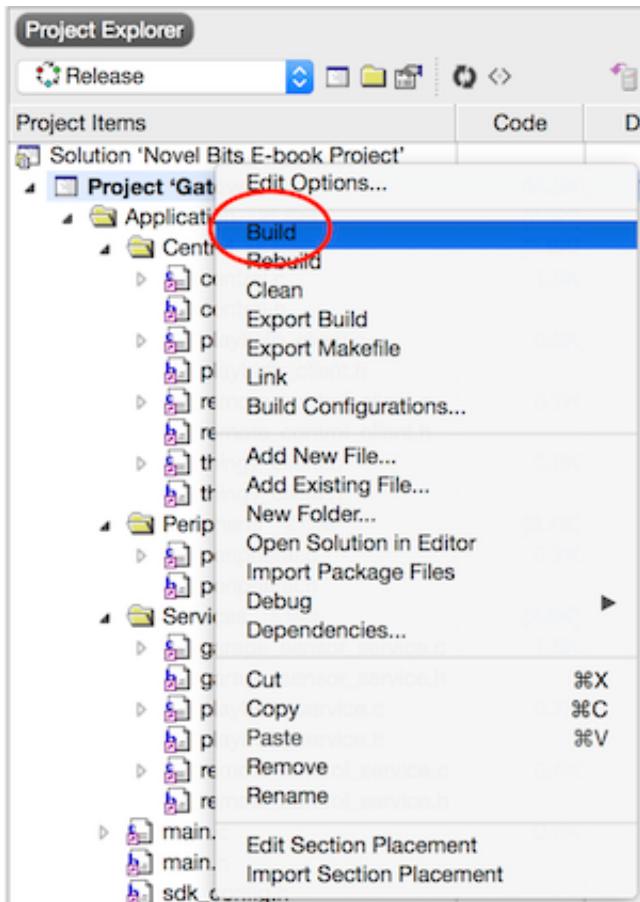


Figure 37: Build Project (SES)

- Let's test running the Gateway project on the Target. First, make sure that (see image below):
 - The nRF52 development kit is connected to your computer via USB.
 - LED 5 is turned ON
 - Power is switched to ON position
 - nRF Switch set to Default

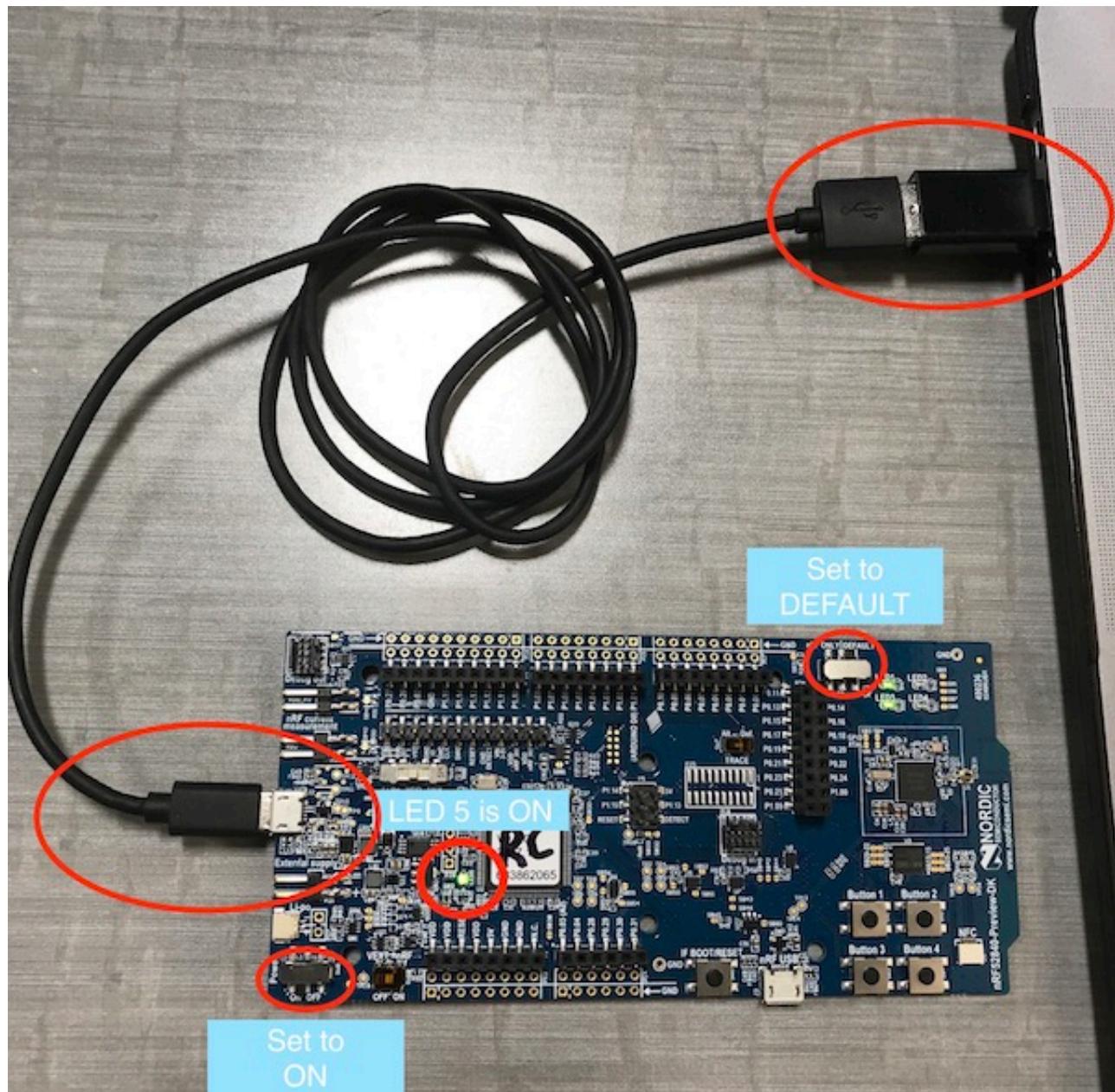


Figure 38: Development Kit Configuration

Important Note:

I recommend connecting **only one** nRF development kit to your computer at a time - specifically when flashing the board or debugging. This is because it can be hard to figure out which serial port corresponds to with development kit you have connected.

Alternatively, you can have more than one connected to your computer, but only switch the **Power** on for the one you're debugging or flashing the firmware to.

- Also, make sure you make the Gateway Project the “Active Project”.
- Press the “Start Debugging” button in the top toolbar

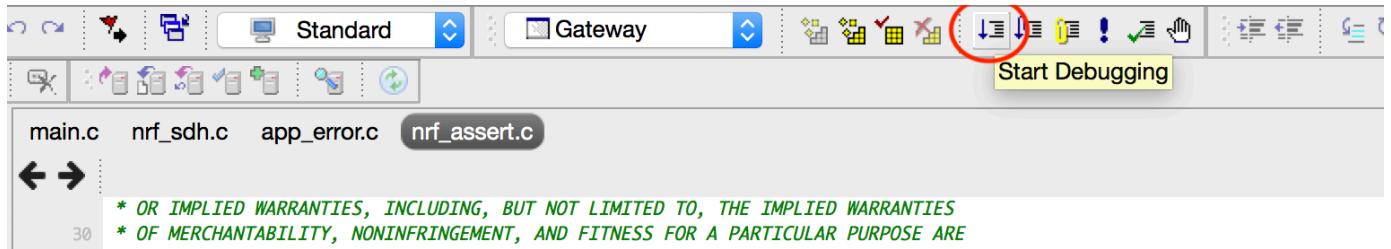


Figure 39: Start Debugging

- The application will start and will stop in the main function waiting for you to continue running

The screenshot shows a debugger interface with two main windows. The top window displays the source code for `main.c`. A red circle highlights the opening brace of the `main` function at line 409. The bottom window shows the `Output` and `Call Stack` panes.

```

main.c
int main()
{
    NRF_LOG_DEFAULT_BACKENDS_INIT();
}

/**@brief Function for the Power manager.
 */
static void power_manage(void)
{
    ret_code_t err_code = sd_app_evt_wait();
    APP_ERROR_CHECK(err_code);
}

/**@brief Function for application main entry.
 */
int main(void)
{
    // Initialize various services
    log_init();
    timers_init();
    buttons_leds_init();
    ble_stack_init();
    gap_params_init();
    gatt_init();
    conn_params_init();
    db_discovery_init();
    peer_manager_init();

    // Initialize the central clients (Remote Control, Thingy:52, Playbulb Candle)
    central_init();

    // Initialize services, then advertise
    services_init();
    advertising_init();
}

```

Output

Show:	Target	Tasks
Preparing target for download	Completed	
Downloading 's140_nrf52840_5.0.0-2.alpha_softdevice.hex' to J-Link	Download successful	0.0 KB in 0.5s 0.1 KB/s
Downloading 'Gateway.elf' to J-Link	Download successful	0.0 KB in 0.3s 0.2 KB/s

Call Stack

Function	Call Address
int ma...	0x0002BAE0
start()	0x000221AC

Figure 40: Debugger Stopped in main()

- Hit the Continue Execution (play) button (circled in the screenshot)

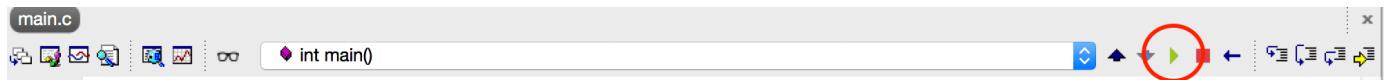


Figure 41: Continue Debugger Execution

- The Debug Terminal should now show the Gateway application running on your target with the debug terminal showing the following text: "Novel Bits Gateway started".

```

gatt_params_init(),
gatt_init();
conn_params_init();
db_discovery_init();
peer_manager_init();

// Initialize the central clients (Remote Control, Thingy:52, Playbulb

```

Debug Terminal

```

<info> app: Fast advertising.
<info> app: Novel Bits Gateway started.

```

Figure 42: Gateway Example Debug Output

- The final thing we want to do is enable an external tool called the "CMSIS Configuration Wizard". This tool enables you to edit the options within the sdk_config.h file from a user-friendly interface instead of editing it within the raw source code. You will only have to do this step once.
- Navigate to File -> Open Studio File -> External Tools Configuration

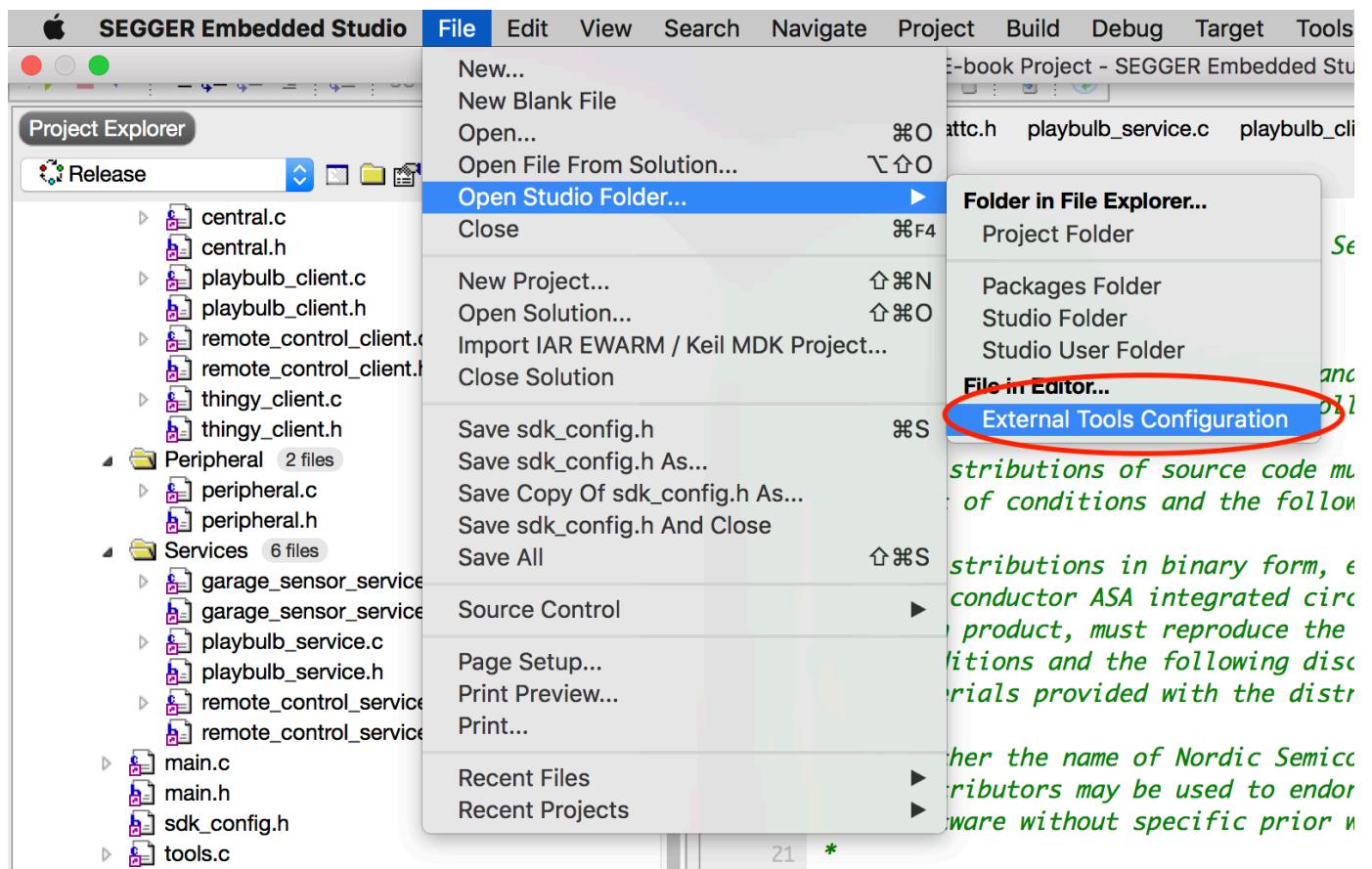


Figure 43: External Tools Configuration

- Now the file `tools.xml` should be open

```

thingy_client.c  ble_gattc.h  playbulb_service.c  playbulb_client.c  ble_gatts.h  playbulb_client.h  central.c  sdk_config.h  tools.xml
← → ⋮
1 <tools>
2
3  <!-- PC-lint - http://www.gimpel.com/html/pcl.htm -->
4
5  <if host_os="win">
6    <item name="Tool.PClint">
7      <menu>&PC-lint (Unit Check)</menu>
8      <text>PC-lint (Unit Check)</text>
9      <tip>Run a PC-lint unit checkout on the selected file or folder</tip>
10     <key>Ctrl+L, Ctrl+P</key>
11     <match>*.c;*.cpp</match>
12     <message>Linting</message>
13     <commands>
14       "$(LINTDIR)/lint-nt" -v -incvar(__CW_ARM) -i$(LINTDIR)/lnt co-gcc.lnt $(DEFINES)
15     </commands>
16   </item>
17 </if>
18
19 </tools>
20

```

Figure 44: Tools.xml

- After you have the tools.xml file open, add the following xml code before the `</tools>` tag:

```

<item name="Tool.CMSIS_Config_Wizard" wait="no">
  <menu>&CMSIS Configuration Wizard</menu>
  <text>CMSIS Configuration Wizard</text>
  <tip>Open a configuration file in CMSIS Configuration Wizard</tip>
  <key>Ctrl+Y</key>
  <match>*config*.h</match>
  <message>CMSIS Config</message>
  <commands>
    java -jar "$(CMSIS_CONFIG_TOOL)" "$(InputPath)";
  </commands>
</item>

```

Make sure to place it outside the `<if host_os=...>...</if>` statement:



```

1 <tools>
2
3 <!-- PC-lint - http://www.gimpel.com/html/pcl.htm -->
4
5 <if host_os="win">
6   <item name="Tool.PClint">
7     <menu>&PC-lint (Unit Check)</menu>
8     <text>PC-lint (Unit Check)</text>
9     <tip>Run a PC-lint unit checkout on the selected file or folder</tip>
10    <key>Ctrl+L, Ctrl+P</key>
11    <match>*.c,*.cpp</match>
12    <message>Linting</message>
13    <commands>
14      &quot;$(LINTDIR)/lint-nt&quot; -v -incvar(__CW_ARM) -i$(LINTDIR)/lnt co-gcc.lnt $(DEFINES) $(INCLUDES) -D__GNUC__ -u -b +ma
15    </commands>
16  </item>
17 </if>
18
19 <item name="Tool.CMSIS_Config_Wizard" wait="no">
20   <menu>&CMSIS Configuration Wizard</menu>
21   <text>CMSIS Configuration Wizard</text>
22   <tip>Open a configuration file in CMSIS Configuration Wizard</tip>
23   <key>Ctrl+Y</key>
24   <match>*config*.h</match>
25   <message>CMSIS Config</message>
26   <commands>
27     java -jar &quot;$(CMSIS_CONFIG_TOOL)&quot; &quot;$(InputPath)&quot;
28   </commands>
29 </item>
30 </tools>

```

Figure 45: Adding CMSIS Configuration Wizard

- Now, quit and restart the SES IDE
- After it restarts, you can now right-click on the `sdk_config.h` file within the Remote Control or Gateway project's **Application** folder and you will see the option "CMSIS Configuration Wizard".

Main Project File Structure

Introduction

Make sure you refer to the chapter titled "**Main Project**" before reading this chapter. It is important to understand the overall architecture and design of the Main Project before digging into this chapter.

A few notes regarding the Segger Embedded Studio Solution/Project structure:

- A complete "Solution" is provided for the nRF52-based code (under the **Main Project SES files** folder in the root folder of the provided source code).
- A Solution is a group of one or more Projects. In our case, we are bundling the Gateway and Remote Control projects in one solution.
- This makes it easier to have everything related to the book's project in one place. It also reduces the overhead of switching between two different solutions — one per device.
- The Solution file is named: `NovelBits.emProject`.
- The Solution can be opened directly from Segger Embedded Studio.
- The Solution contains two Projects:
 - Gateway
 - Remote Control
- You can build the Solution as a whole, but each Project produces a separate result that gets flashed to a development kit.
- You can choose to build each Project separately.
- All files were added to the Projects with relative paths to provide a better out-of-the-box experience (no modifications to the Solution and included Projects necessary to build).

The Remote Control Project

As we mentioned before, the Projects included in the GitHub folder are ready for you to use out-of-the-box. Here's a screenshot of the SES Remote Control project structure:

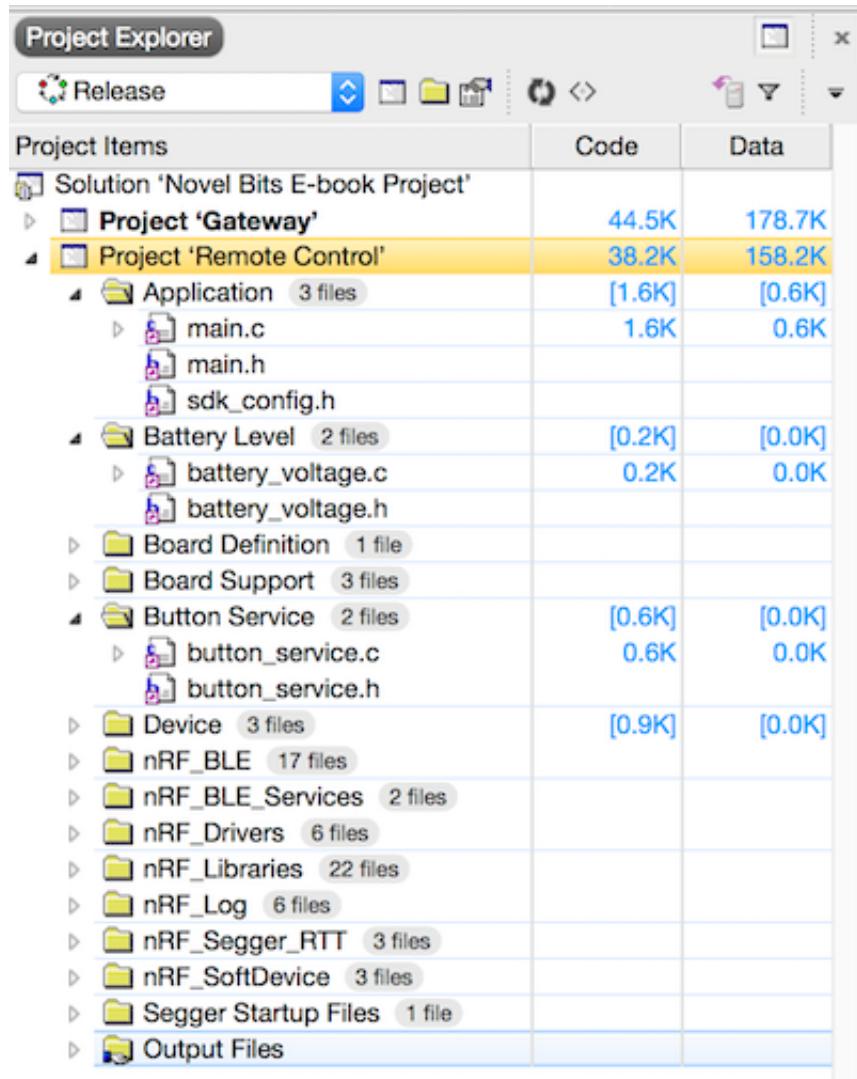


Figure 46: The Remote Control Example SES Project

The main parts of the project that involve our application code are:

- **Application:**
 - `main.h` : header file including definitions for the Remote Control main application file
 - `main.c` : Remote Control main application file
 - `sdk_config.h` : configuration file for the nRF5 SDK for the Remote Control application
- **Battery Level:**
 - `battery_voltage.h` : header file for the battery voltage reading module
 - `battery_voltage.c` : battery voltage reading module main file
- **Button Service:**
 - `button_service.h` : header file for the button service

- `button_service.c` : button service main file needed to detect ON and OFF button presses/releases on the nRF52 development kit

The Gateway Project

Following is a screenshot of the SES Gateway project structure:

Project Explorer		Code	Data
	Release		
Project Items			
Solution 'Novel Bits E-book Project'			
Project 'Gateway'		44.5K	178.7K
Application (21 files)		[0.8K]	[0.1K]
Central (8 files)		[3.9K]	[4.9K]
central.c		1.9K	3.5K
central.h			
playbulb_client.c		0.3K	0.3K
playbulb_client.h			
remote_control_client.c		0.7K	0.5K
remote_control_client.h			
thingy_client.c		0.8K	0.5K
thingy_client.h			
Peripheral (2 files)		[0.7K]	[0.5K]
peripheral.c		0.7K	0.5K
peripheral.h			
Services (6 files)		[2.8K]	[0.6K]
garage_sensor_service.c		1.5K	0.4K
garage_sensor_service.h			
playbulb_service.c		0.7K	0.2K
playbulb_service.h			
remote_control_service.c		0.4K	0.0K
remote_control_service.h			
main.c		0.6K	0.1K
main.h			
sdk_config.h			
tools.c		0.2K	
tools.h			
Board Definition (1 file)		[0.4K]	[0.0K]
Board Support (3 files)			
Device (3 files)			
nRF_BLE (18 files)			
nRF_BLE_Services (3 files)			

▷	nRF_Drivers 6 files	[10.8K]	[2.1K]
▷	nRF_Libraries 23 files		
▷	nRF_Log 6 files		
▷	nRF_Segger_RTT 3 files	[1.2K]	[0.7K]
▷	nRF_SoftDevice 3 files		
▷	Segger Startup Files 1 file		
▷	Output Files		
▷	Project 'Remote Control'	38.2K	158.2K

Figure 47: Adding CMSIS Configuration Wizard

The main parts that involve our application code are:

- **Application:**
 - `main.h` : header file including definitions for the Gateway main application file
 - `main.c` : Gateway main application file
 - `tools.h` : header file for utility functions module
 - `tools.c` : utility functions module for parsing advertisement packets and searching for device names and UUIDs
 - `sdk_config.h` : configuration file for the nRF5 SDK for the Gateway application
- **Central:**
 - `central.h` : header file for the central role module for the Gateway
 - `central.c` : central role module that handles operations relating to the central role of the Gateway such as connecting to the different peripherals (Thingy:52, Remote Control, and Playbulb candle)
 - `playbulb_client.h` : header file for the playbulb client module
 - `playbulb_client.c` : playbulb client module that handles discovering the services and characteristics, and subscribing to and handling notifications from the Playbulb device
 - `remote_control_client.h` : header file for the Remote Control client module
 - `remote_control_client.c` : Remote Control client module handles discovering the services and characteristics, and subscribing to and handling notifications from the Remote Control device
 - `thingy_client.h` : header file for the Thingy:52 client module
 - `thingy_client.c` : Thingy:52 client module handles discovering the services and characteristics, and subscribing to and handling notifications from the Thingy:52 device
- **Peripheral:**
 - `peripheral.h` : header file for the peripheral role module for the Gateway
 - `peripheral.c` : peripheral role module that relays the data from the different connected peripheral devices such as the Thingy:52, Playbulb candle, and Remote Control to other BLE devices
- **Services:**
 - `garage_sensor_service.h` : header file for the Garage Sensor service
 - `garage_sensor_service.c` : Garage Sensor service module that handles exposing the Thingy:52 temperature, humidity, motion and battery level information

- `playbulb_service.h` : header file for the Playbulb candle service module
- `playbulb_service.c` : Playbulb candle service module that handles exposing the Playbulb candle light status (ON or OFF)
- `remote_control_service.h` : header file for the Remote Control service module
- `remote_control_service.c` : Remote Control service module that handles exposing the Remote Control battery level information

Note: You don't need to explicitly include the header files in your project structure. You simply need it to add them (or their respective folders) to the Project settings. However, we include them in the Projects to make it easier to refer to during development and while navigating the source code.

Other Included Files and Folders

Other included files and folders present in the Projects come from the nRF5 SDK and are located in the `nRF5_SDK_current` folder in the Main Project root folder (similar to other example applications included in the SDK folder).

nRF Development and Troubleshooting Tips

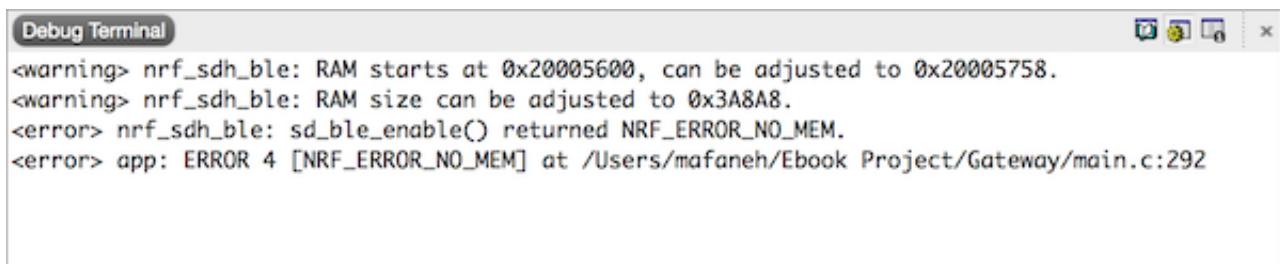
Tip #1: RAM Start and RAM Size

For some of the modifications to the `sdk_config.h` file, the RAM size used by the SoftDevice will change causing the RAM offset and size of the user application to shift. This causes the application to halt and not work properly (sometimes causing an early exception).

Fortunately, the SoftDevice prints out an error message in the debug messages log indicating the correct RAM start and size for the application. Make sure you have logging enabled with the following settings:

- `NRF_LOG_DEFAULT_LEVEL` set to ≥ 3 .
- `NRF_SDH_BLE_LOG_ENABLED` set to 1.
- `NRF_SDH_BLE_LOG_LEVEL` set to ≥ 2 .
- Logger configured to use the Segger RTT backend:
`NRF_LOG_BACKEND_RTT_ENABLED` set to 1.

The error message will look like this:



```
Debug Terminal
<warning> nrf_sdh_ble: RAM starts at 0x20005600, can be adjusted to 0x20005758.
<warning> nrf_sdh_ble: RAM size can be adjusted to 0x3A8A8.
<error> nrf_sdh_ble: sd_ble_enable() returned NRF_ERROR_NO_MEM.
<error> app: ERROR 4 [NRF_ERROR_NO_MEM] at /Users/mafaneh/Ebook Project/Gateway/main.c:292
```

Figure 48: RAM Start and Size Error

This can be fixed by modifying the RAM start and size of the application from within SES:

- Right click on the Project, the click on "Edit options"

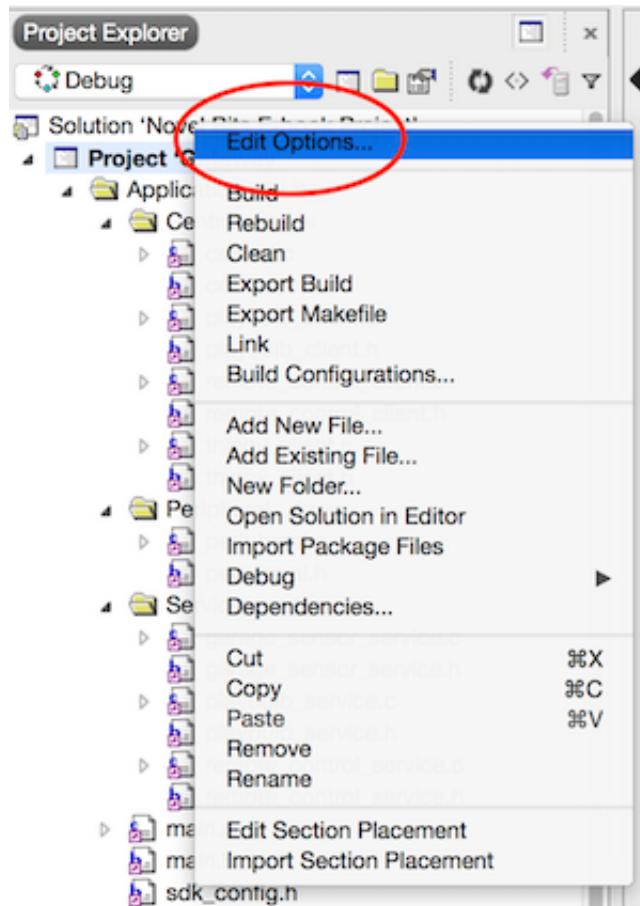
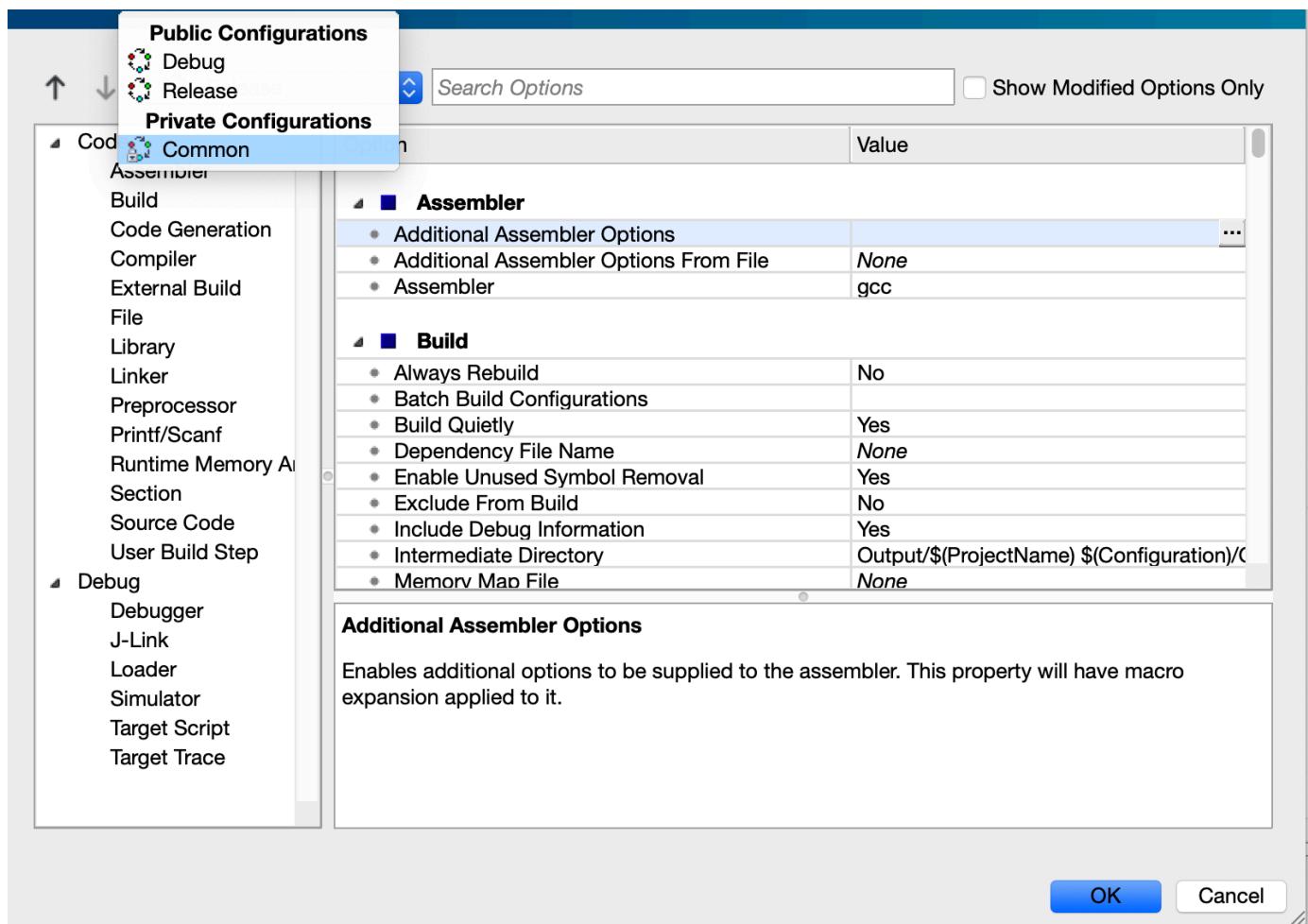
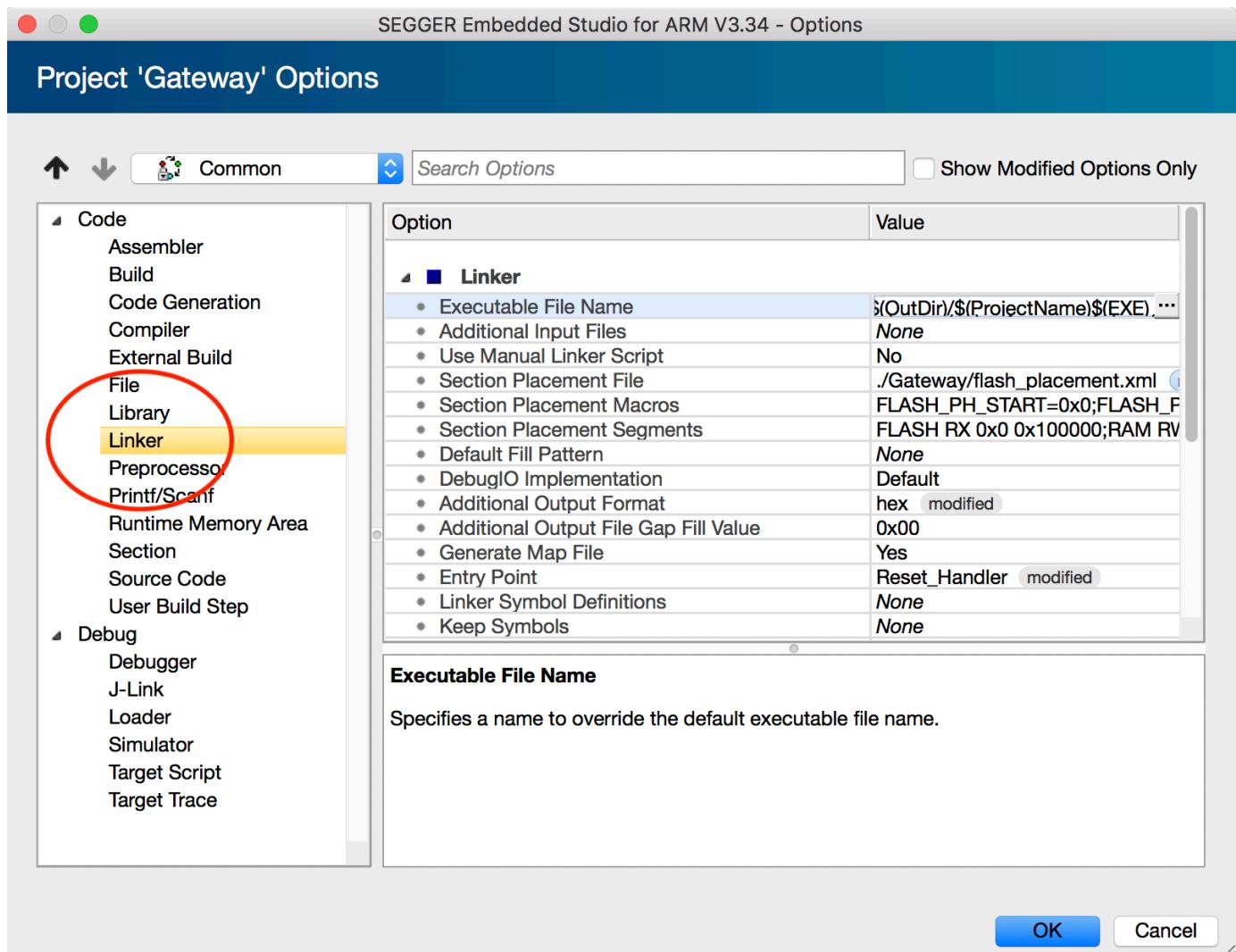


Figure 49: Project Options

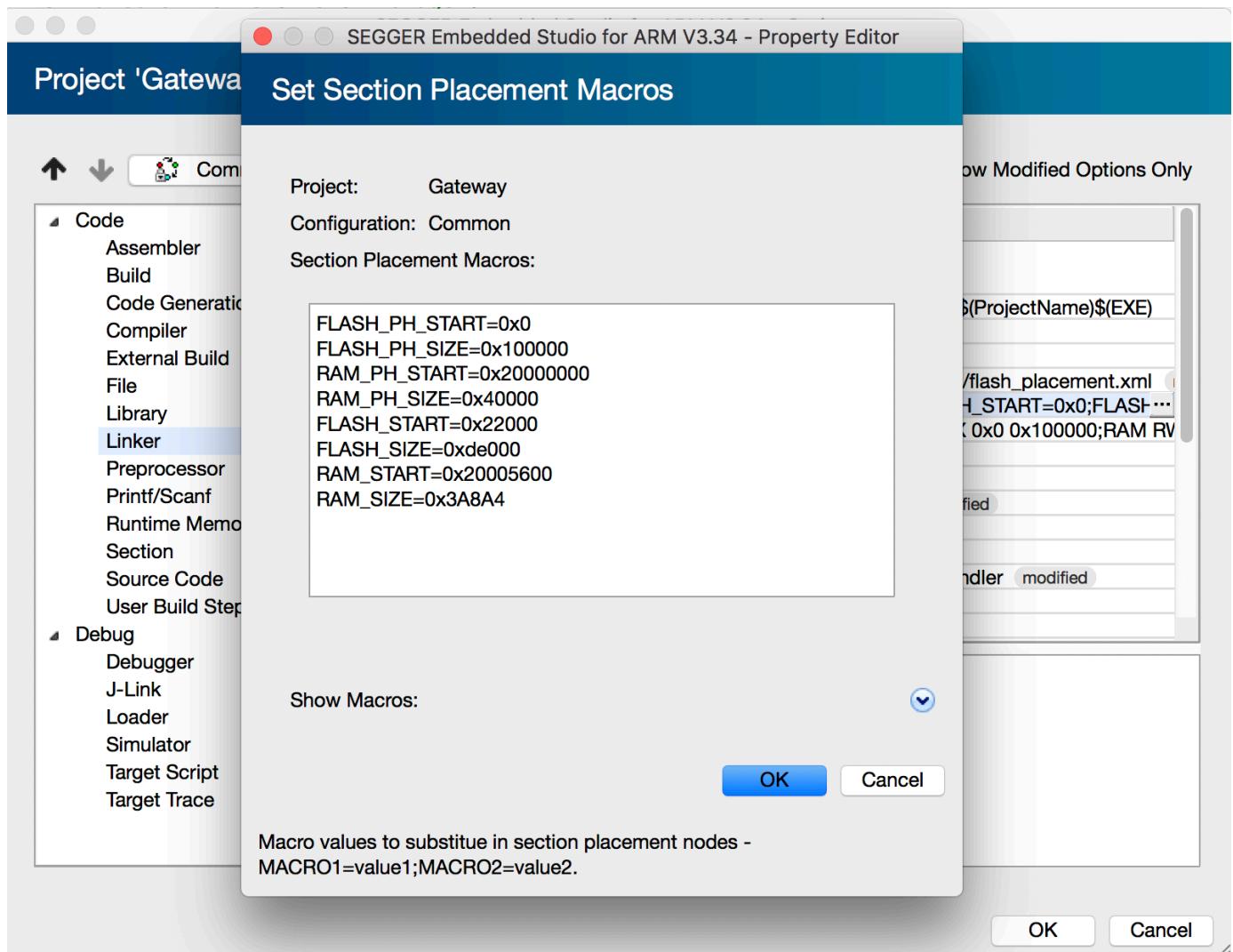
- From the Configurations drop-down menu, choose "Common":

*Figure 50: Common Configuration Options*

- Select "Linker":

*Figure 51: Linker Options*

- Double click on "Section Placement Macros", then modify the RAM start and RAM size to match the values provided by the SoftDevice in the Debug Terminal output:

*Figure 52: Section Placement Macros*

Here are a couple of useful YouTube videos by Nordic Semiconductor explaining the Build Options for a project:

- [Build Configurations video](#)
- [Build Settings video](#)

Tip #2: Message Sequence Charts

For many of the SoftDevice operations and application flow, Nordic's InfoCenter provides Message Sequence Charts that can be very helpful in knowing the correct sequence of API calls to achieve a certain task. For example, for setting Advertisement data and starting Advertisements, the [following sequence chart](#) shows the correct sequence of API calls to achieve the task:

GAP Advertisement

Message Sequence Charts



Figure 53: Example of a Message Sequence Chart

Tip #3: Custom Services & Characteristics (Vendor-Specific UUIDs)

When adding custom services and characteristics, the SoftDevice requires special handling of these compared to Bluetooth SIG-adopted services and characteristics. Before adding any custom attributes, make sure you modify the `NRF_SDH_BLE_VS_UUID_COUNT` macro to reflect the number of custom attributes your application is adding or wants to be aware of. To do so, you can use the CMSIS Configuration Wizard (as described in the Development Environment Setup section).

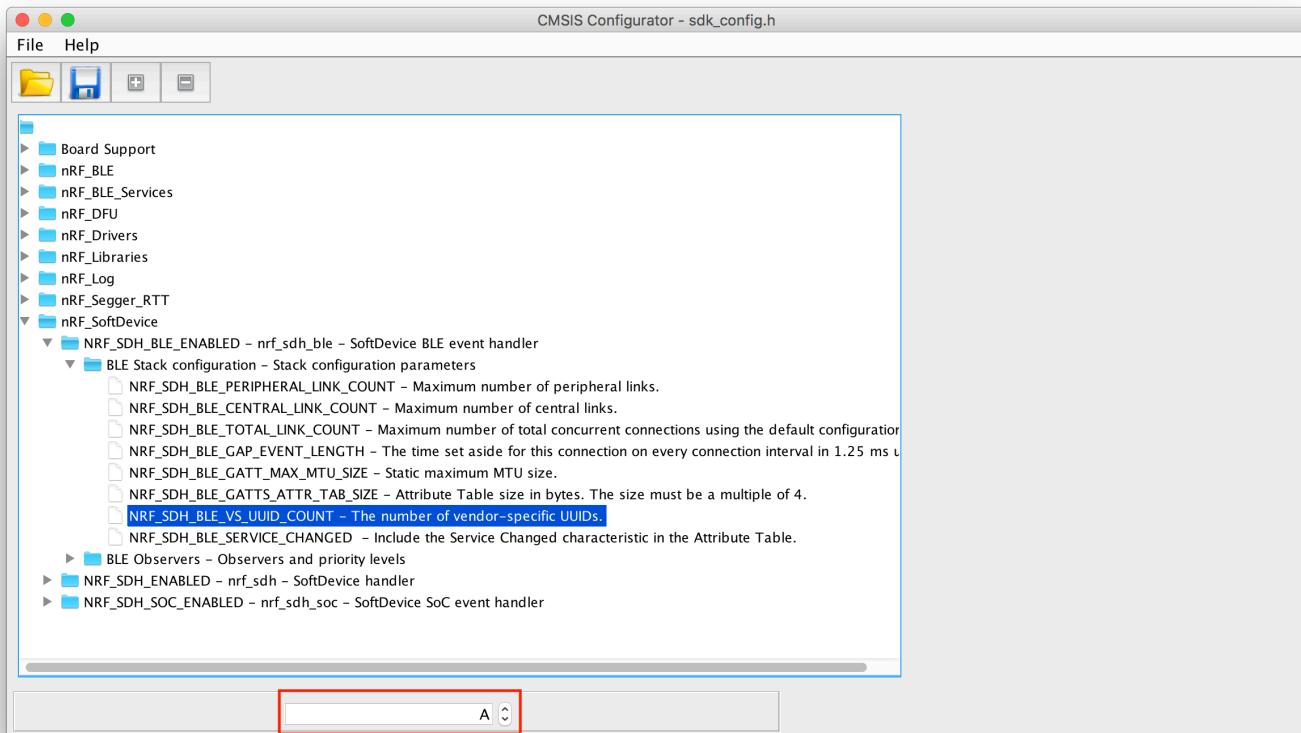


Figure 54: Vendor-Specific UUID Count

The value shown here `A` is the hexadecimal value for the macro. This enables the SoftDevice to reserve the number of custom services and characteristics in the stack.

Tip #4: SoftDevice Errors

Error codes reported by the SoftDevice can be a bit cryptic. To quickly understand the error you are seeing, refer to the API reference for the SoftDevice you are using: [SoftDevice S140 v6.1.0](#). Keep in mind that you may have to refer to the API returning the error to determine what type of error it is before looking up the matching value of the error code.

Tip #5: nRF5 SDK Examples

Another good reference is the examples folder provided within the SDK. The SDK contains examples for BLE central applications, BLE peripheral applications, hardware peripherals, and others. You can find them located under the folder `nRF5_SDK_current/examples`.

The "Hello World" Example

In this chapter, we'll go over a simple implementation of a BLE Peripheral. Think of it as the "Hello World" example for BLE. The application will implement the following:

- A Peripheral application which "advertises" its presence as well as allow a connection from a Central device.
- A simple GATT structure that exposes one Service which includes two characteristics:
 - A Characteristic to indicate whether Button 1 of the development kit is in the pressed state (value = 0x01) or in the released state (value = 0x00).
 - Another Characteristic to store a one-byte value that the Central can write to and read. Let's dive into the code!

Note: The source code along with the SES project file is included in the GitHub repository that accompanies the book under the folder named "**Hello World**". The repository also includes a "**Hello World**" example for the nRF52832 chipset.

Basic Application Structure

The application code is structured as following:

- `main.c` : implements the main function of the application. The function initializes all the necessary parts for the BLE Peripheral such as the SoftDevice, timers, buttons, LEDs, and others.
- `simple_service.h` : the header file for the service that implements the Service and Characteristics mentioned above.
- `simple_service.c` : the implementation code for initializing and adding the Service which includes the two Characteristics mentioned: the Button 1 pressed/released state Characteristic, and the "store value" Characteristic.

Source Code

main.c

```
int main(void)
{
    // Initialize.
    timers_init();
    log_init();
    buttons_leds_init();
```

```

ble_stack_init();
gap_params_init();
gatt_init();
advertising_init();
services_init();
conn_params_init();
peer_manager_init();

// Start execution.
NRF_LOG_INFO("Novel Bits Hello World application started.");

advertising_start();

// Enter main loop.
for (;;)
{
    if (NRF_LOG_PROCESS() == false)
    {
        power_manage();
    }
}
}

```

Note: Most of the following function names will be found in any application built around the SoftDevice from Nordic Semiconductor:

- `timers_init()` : initializes the timer library needed specifically for the LEDs and the detection of the button press.
- `log_init()` : initializes the Logger module that allows the developer to output debug messages.
- `buttons_leds_init()` : initializes the Board Support Package library responsible for detecting button presses as well as controlling the LEDs.
- `ble_stack_init()` : initializes the BLE stack.
- `gap_params_init()` : initializes the parameters related to the GAP layer including the device name, the Device Appearance, and the Peripheral Preferred Connection Parameters (PPCP) such as the minimum/maximum Connection Interval, Slave Latency, and Supervision Timeout.
- `gatt_init()` : initializes the parameters related to the GATT layer.
- `advertising_init()` : initializes the Advertising module including the setting of the Advertising parameters such as the Advertising Interval, Timeout, and the Advertising data.
- `services_init()` : initializes the “simple service” along with its Characteristics.
- `conn_params_init()` : initializes parameters such as the Connection Parameters Update delay and the event handler for the Connection Parameters Update event.
- `peer_manager_init()` : initializes the Peer Manager module which includes setting the security parameters as well as setting the Peer Manager event handler.
- `advertising_start()` : starts sending Advertising packets for other devices to discover.
- `power_manage()` : calls `sd_app_evt_wait()` , which puts the device to sleep until it an event occurs that needs to wake up the application.

simple_service.h

First, we add a conditional include for the header file, which makes sure the header file is only included once during compilation.

```
#ifndef SIMPLE_SERVICE_H
#define SIMPLE_SERVICE_H
```

We need to include the header file `ble.h` and `ble_srv_common.h` which are needed for BLE related data structures.

```
#include <stdint.h>
#include "ble.h"
#include "ble_srv_common.h"
```

Next, we define our Service. The UUIDs for the Service and Characteristics can be randomly chosen (*refer to the GATT chapter for more details*).

```
// Simple service: E54B0001-67F5-479E-8711-B3B99198CE6C
//   Button 1 press characteristic: E54B0002-67F5-479E-8711-B3B99198CE6C
//   Store value characteristic: E54B0003-67F5-479E-8711-B3B99198CE6C

// The bytes are stored in little-endian format, meaning the
// Least Significant Byte is stored first
// The order used in the "human-readable" comment above is big-endian
// (you'll see them reversed and byte-separated in as the code below to make the little-endian)
```

Using the Nordic SoftDevice, we need to separate the UUIDs into two parts: the **Base**, and the 2 bytes that occupy the **3rd and 4th Most Significant Bytes (MSB)**.

The nRF SDK utilizes this methodology in order to make it simpler for developers to interact with both Bluetooth SIG-adopted UUIDs and Vendor-Specific (custom) UUIDs in the same manner.

```
// Base UUID: E54B0000-67F5-479E-8711-B3B99198CE6C
#define BLE_UUID_SIMPLE_SERVICE_BASE_UUID {0x6C, 0xCE, 0x98, 0x91, 0xB9, 0xB3, 0x11, 0x87, 0x9E, 0x47,
0xF5, 0x67, 0x00, 0x00, 0x4B, 0xE5}

// Service & characteristics UUIDs
#define BLE_UUID_SIMPLE_SERVICE_UUID      0x0001
#define BLE_UUID_BUTTON_1_PRESS_CHAR_UUID 0x0002
#define BLE_UUID_STORE_VALUE_CHAR_UUID    0x0003
```

Next, we “forward-declare” the main **simple service** data structure `ble_simple_service_t`. This is so we could use it in the preceding data structures and function type definitions (such as `ble_simple_evt_handler_t`).

```
typedef struct ble_simple_service_s ble_simple_service_t;
```

The following enumeration defines two enum values that are passed to the application (in `main.c`) to be notified of two events: enabling the Notification of the Characteristic by the Central device, as well as disabling the Notification.

```
typedef enum
{
    BLE_BUTTON_1_PRESS_EVT_NOTIFICATION_ENABLED,
    BLE_BUTTON_1_PRESS_EVT_NOTIFICATION_DISABLED,
} ble_simple_evt_type_t;
```

The following data structure stores the simple service event.

```
typedef struct
{
    ble_simple_evt_type_t evt_type;
} ble_simple_evt_t;
```

Next, we define the event handler function pointer that is used by the application level (in `main.c`).

```
typedef void (*ble_simple_evt_handler_t) (ble_simple_service_t * p_simple_service, ble_simple_evt_t * p_evt);
```

Following is the main simple service data structure that stores all the data related to the service including: the connection handle, the service handle, the vendor-specific UUID type value, the event handler function pointer, as well as the handles for the two Characteristics.

```
typedef struct ble_simple_service_s
{
    uint16_t                      conn_handle;
    uint16_t                      service_handle;
    uint8_t                        uuid_type;
    ble_simple_evt_handler_t      evt_handler;
    ble_gatts_char_handles_t     button_1_press_char_handles;
    ble_gatts_char_handles_t     store_value_char_handles;
} ble_simple_service_t;
```

Next, we declare the Service initialization function `ble_simple_service_init()`.

```
uint32_t ble_simple_service_init(ble_simple_service_t * p_simple_service, ble_simple_evt_handler_t app_evt_handler);
```

Here, we declare the Service event handler function that needs to be called from the application level (`main.c`). It is used to pass on any BLE events that need to be handled locally by the Service.

```
void ble_simple_service_on_ble_evt(ble_simple_service_t * p_simple_service, ble_evt_t const * p_ble_evt);
```

The last function declaration is for updating the button press Characteristic, which also sends a Notification to the Central device in case they are enabled.

```
void button_1_characteristic_update(ble_simple_service_t * p_simple_service, uint8_t *button_action);
```

Finally, we end the conditional `#include` for the header file. `#endif /* SIMPLE_SERVICE_H */`

simple_service.c

Include files needed for the application:

```
#include <string.h>
#include "nrf_log.h"
#include "boards.h"
#include "simple_service.h"
```

Here, we define strings that are used in the descriptors for the Characteristics:

```
static const uint8_t Button1CharName[] = "Button 1 press";
static const uint8_t StoreValueCharName[] = "Store Value";
```

The following are functions that set the connection handle upon a successful connection, and reset it upon disconnection.

```
static void on_connect(ble_simple_service_t * p_simple_service, ble_evt_t const * p_ble_evt)
{
    p_simple_service->conn_handle = p_ble_evt->evt.gap_evt.conn_handle;
}
```

```
static void on_disconnect(ble_simple_service_t * p_simple_service, ble_evt_t const * p_ble_evt)
{
    UNUSED_PARAMETER(p_ble_evt);
    p_simple_service->conn_handle = BLE_CONN_HANDLE_INVALID;
}
```

The next function handles two write events: one for writing to the “store value” Characteristic, and another for enabling/disabling Notifications on the button press Characteristic. In the case Notifications are enabled on the button press, the application level event handler is called.

```
static void on_write(ble_simple_service_t * p_simple_service, ble_evt_t const * p_ble_evt)
{
    ble_gatts_evt_write_t const * p_evt_write = &p_ble_evt->evt.gatts_evt.params.write;

    // Custom Value Characteristic Written to.
    if (p_evt_write->handle == p_simple_service->store_value_char_handles.value_handle)
    {
        NRF_LOG_INFO("Store Value written: %d", *p_evt_write->data);
    }

    if ((p_evt_write->handle == p_simple_service->button_1_press_char_handles.cccd_handle)
        && (p_evt_write->len == 2))
    {
        ble_simple_evt_t evt;

        if (ble_srv_is_notification_enabled(p_evt_write->data))
        {
            NRF_LOG_INFO("Notification enabled for button 1 press");
            evt.evt_type = BLE_BUTTON_1_PRESS_EVT_NOTIFICATION_ENABLED;
        }
        else
        {
            NRF_LOG_INFO("Notification disabled for button 1 press");
            evt.evt_type = BLE_BUTTON_1_PRESS_EVT_NOTIFICATION_DISABLED;
        }

        if (p_simple_service->evt_handler != NULL)
        {
            // CCCD written, call application event handler.
            p_simple_service->evt_handler(p_simple_service, &evt);
        }
    }
}
```

Next, we define a function that adds the button 1 press Characteristic:

```
static uint32_t button_1_press_char_add(ble_simple_service_t * p_simple_service)
{
```

```

uint32_t err_code;
ble_gatts_char_md_t char_md;
ble_gatts_attr_md_t cccd_md;
ble_gatts_attr_t attr_char_value;
ble_uuid_t ble_uuid;
ble_gatts_attr_md_t attr_md;
uint8_t init_value = 0;

memset(&char_md, 0, sizeof(char_md));
memset(&cccd_md, 0, sizeof(cccd_md));
memset(&attr_md, 0, sizeof(attr_md));
memset(&attr_char_value, 0, sizeof(attr_char_value));

```

We want the connecting Client to be able to read and write to the CCCD (which allows enabling/disabling Notifications). We also want the value to be read-only (Writes not allowed).

```

// Set permissions on the CCCD and Characteristic value
BLE_GAP_CONN_SEC_MODE_SET_OPEN(&cccd_md.read_perm);
BLE_GAP_CONN_SEC_MODE_SET_OPEN(&cccd_md.write_perm);
BLE_GAP_CONN_SEC_MODE_SET_NO_ACCESS(&attr_md.write_perm);
BLE_GAP_CONN_SEC_MODE_SET_OPEN(&attr_md.read_perm);

```

Here, we let the SoftDevice handle storing the CCCD value. Another option would be to have it stored and handled by the user (application).

```

// CCCD settings (needed for notifications and/or indications)
cccd_md.vloc = BLE_GATTS_VLOC_STACK;

// Characteristic Metadata
char_md.char_props.read      = 1;
char_md.char_props.notify    = 1;
char_md.p_char_user_desc     = Button1CharName;
char_md.char_user_desc_size  = sizeof(Button1CharName);
char_md.char_user_desc_max_size = sizeof(Button1CharName);
char_md.p_char_pf            = NULL;
char_md.p_user_desc_md       = NULL;
char_md.p_cccd_md            = &cccd_md;
char_md.p_sccd_md            = NULL;

// Define the Button ON press Characteristic UUID
ble_uuid.type = p_simple_service->uuid_type;
ble_uuid.uuid = BLE_UUID_BUTTON_1_PRESS_CHAR_UUID;

// Attribute Metadata settings
attr_md.vloc      = BLE_GATTS_VLOC_STACK;
attr_md.rd_auth   = 0;
attr_md.wr_auth   = 0;
attr_md.vlen      = 0;

```

```

// Attribute Value settings
attr_char_value.p_uuid      = &ble_uuid;
attr_char_value.p_attr_md    = &attr_md;
attr_char_value.init_len     = sizeof(uint8_t);
attr_char_value.init_offs   = 0;
attr_char_value.max_len     = sizeof(uint8_t);
attr_char_value.p_value     = &init_value;

return sd_ble_gatts_characteristic_add(p_simple_service->service_handle, &char_md,
                                       &attr_char_value,
                                       &p_simple_service->button_1_press_char_handles);
}

```

Next is the function that handles creating and adding the “store value” Characteristic.

```

static uint32_t store_value_char_add(ble_simple_service_t * p_simple_service)
{
    uint8_t local_variable_initial_value = 0;
    ble_gatts_char_md_t char_md;
    ble_gatts_attr_md_t cccd_md;
    ble_gatts_attr_t    attr_char_value;
    ble_uuid_t          ble_uuid;
    ble_gatts_attr_md_t attr_md;

    memset(&char_md, 0, sizeof(char_md));
    memset(&cccd_md, 0, sizeof(cccd_md));
    memset(&attr_md, 0, sizeof(attr_md));
    memset(&attr_char_value, 0, sizeof(attr_char_value));

```

For this Characteristic, we do not want Notifications to be allowed, so we restrict access to the CCCD.

```

// Set permissions on the CCCD and Characteristic value
BLE_GAP_CONN_SEC_MODE_SET_NO_ACCESS(&cccd_md.read_perm);
BLE_GAP_CONN_SEC_MODE_SET_NO_ACCESS(&cccd_md.write_perm);

```

However, we want Writes and Reads to be enabled:

```

BLE_GAP_CONN_SEC_MODE_SET_OPEN(&attr_md.write_perm);
BLE_GAP_CONN_SEC_MODE_SET_OPEN(&attr_md.read_perm);

// CCCD settings (needed for notifications and/or indications)
cccd_md.vloc = BLE_GATTS_VLOC_STACK;

char_md.char_props.read      = 1;
char_md.char_props.write     = 1;
char_md.p_char_user_desc    = StoreValueCharName;
char_md.char_user_desc_size = sizeof(StoreValueCharName);

```

```

char_md.char_user_desc_max_size = sizeof(StoreValueCharName);
char_md.p_char_pf = NULL;
char_md.p_user_desc_md = NULL;
char_md.p_cccd_md = NULL;
char_md.p_sccd_md = NULL;

// Define the Button OFF press Characteristic UUID
ble_uuid.type = p_simple_service->uuid_type;
ble_uuid.uuid = BLE_UUID_STORE_VALUE_CHAR_UUID;

// Attribute Metadata settings
//attr_md.vloc = BLE_GATTS_VLOC_USER;
attr_md.vloc = BLE_GATTS_VLOC_STACK;
attr_md.rd_auth = 0;
attr_md.wr_auth = 0;
attr_md.vlen = 0;

// Attribute Value settings
attr_char_value.p_uuid = &ble_uuid;
attr_char_value.p_attr_md = &attr_md;
attr_char_value.init_len = sizeof(uint8_t);
attr_char_value.init_offs = 0;
attr_char_value.max_len = sizeof(uint8_t);
attr_char_value.p_value = &local_variable_initial_value;

return sd_ble_gatts_characteristic_add(p_simple_service->service_handle, &char_md,
                                       &attr_char_value,
                                       &p_simple_service->store_value_char_handles);
}

```

Here, we define the function that initializes the Service.

```

uint32_t ble_simple_service_init(ble_simple_service_t * p_simple_service, ble_simple_evt_handler_t
app_evt_handler)
{
    uint32_t err_code;
    ble_uuid_t ble_uuid;

    // Initialize service structure
    p_simple_service->conn_handle = BLE_CONN_HANDLE_INVALID;
    // The application does not need to assign an event handler, so we need to check it first
    if (app_evt_handler != NULL)
    {
        p_simple_service->evt_handler = app_evt_handler;
    }
}

```

We need to add the Service's Base UUID (as a vendor-specific UUID):

```

// Add service UUID
ble_uuid128_t base_uuid = {BLE_UUID_SIMPLE_SERVICE_BASE_UUID};

```

```

err_code = sd_ble_uuid_vs_add(&base_uuid, &p_simple_service->uuid_type);
if (err_code != NRF_SUCCESS)
{
    return err_code;
}

```

Then, we define the part of the UUID that gets added to the base (the 3rd and 4th Most Significant Bytes of the UUID):

```

// Set up the short UUID for the service (base + service-specific)
ble_uuid.type = p_simple_service->uuid_type;
ble_uuid.uuid = BLE_UUID_SIMPLE_SERVICE_UUID;

```

Before adding any Characteristics, we need to add the Service to the stack:

```

// Set up and add the service
err_code = sd_ble_gatts_service_add(BLE_GATTS_SRVC_TYPE_PRIMARY, &ble_uuid, &p_simple_service-
>service_handle);
if (err_code != NRF_SUCCESS)
{
    return err_code;
}

```

Now, we can add each of the Characteristics:

```

// Add the different characteristics in the service:
//  Button 1 press characteristic: E54B0002-67F5-479E-8711-B3B99198CE6C
//  Store Value characteristic: E54B0003-67F5-479E-8711-B3B99198CE6C
err_code = button_1_press_char_add(p_simple_service);
if (err_code != NRF_SUCCESS)
{
    return err_code;
}

err_code = store_value_char_add(p_simple_service);
if (err_code != NRF_SUCCESS)
{
    return err_code;
}

return NRF_SUCCESS;
}

```

Following is the function that handles the relevant BLE events:

```

void ble_simple_service_on_ble_evt(ble_simple_service_t * p_simple_service, ble_evt_t const * p_ble_evt)
{
    if (p_simple_service == NULL || p_ble_evt == NULL)
    {
        return;
    }

    switch (p_ble_evt->header.evt_id)
    {
        case BLE_GAP_EVT_CONNECTED:
            on_connect(p_simple_service, p_ble_evt);
            break;

        case BLE_GAP_EVT_DISCONNECTED:
            on_disconnect(p_simple_service, p_ble_evt);
            break;

        case BLE_GATTS_EVT_WRITE:
            on_write(p_simple_service, p_ble_evt);
            break;
        default:
            // No implementation needed.
            break;
    }
}

```

Lastly, we define the function that gets called by the application level (`main.c`) when the button is pressed. It handles two different aspects: updating the value of the Characteristic in the SoftDevice database, and sending the updated value to any Client that enabled Notifications for the Characteristic.

```

void button_1_characteristic_update(ble_simple_service_t * p_simple_service, uint8_t *button_action)
{
    uint32_t err_code = NRF_SUCCESS;

    ble_gatts_value_t gatts_value;

    // Initialize value struct.
    memset(&gatts_value, 0, sizeof(gatts_value));

    gatts_value.len      = sizeof(uint8_t);
    gatts_value.offset   = 0;
    gatts_value.p_value = button_action;

    // Update database.
    err_code = sd_ble_gatts_value_set(p_simple_service->conn_handle,
                                      p_simple_service->button_1_press_char_handles.value_handle,
                                      &gatts_value);

    APP_ERROR_CHECK(err_code);

    if (p_simple_service->conn_handle != BLE_CONN_HANDLE_INVALID)
    {

```

```
NRF_LOG_INFO("Sending notification for button 1 press/release");
    uint16_t           len = sizeof (uint8_t);
    ble_gatts_hvx_params_t hvx_params;
    memset(&hvx_params, 0, sizeof(hvx_params));

    hvx_params.handle = p_simple_service->button_1_press_char_handles.value_handle;

    hvx_params.type    = BLE_GATT_HVX_NOTIFICATION;
    hvx_params.offset  = 0;
    hvx_params.p_len   = &len;
    hvx_params.p_data  = (uint8_t*)button_action;

    err_code = sd_ble_gatts_hvx(p_simple_service->conn_handle, &hvx_params);
    APP_ERROR_CHECK(err_code);
}
}
```

Bluetooth 5

Bluetooth 5 focuses on broadening the range of Internet of Things (IoT) applications that can utilize BLE. It brings us twice the speed, four times the range, and eight times the advertising capacity.

Let's go over the list of new features introduced by Bluetooth 5 (from the Bluetooth specification document):

New features added in version 5:

- CSA 5 features (Higher Output Power)
- Slot Availability Mask (SAM)
- 2 Msym/s PHY for LE
- LE Long Range
- High Duty Cycle Non-Connectable Advertising
- LE Advertising Extensions
- LE Channel Selection Algorithm #2

In this chapter, we'll focus on the most important of these changes:

- 2 Msym/s PHY for LE (2x the speed)
- LE Long Range (4x the range)
- LE Advertising Extensions (8x the Advertising capacity)

To learn more about the other new features introduced in Bluetooth 5 and not covered in this chapter, refer to the Bluetooth 5 specification document.

Note: Msym/s (Megasymbols per second) is used here instead of Mbps because it refers to the actual radio transmission capability. In some cases (as we will see for the Coded PHY), multiple symbols will be used to represent a single bit, therefore reducing the Mbps rate. In the remainder of this chapter, we will be using "M" for short in place of "Msym/s".

Twice the Speed, Four Times the Range

Recall that a PHY refers to the physical radio. The Bluetooth specifications before Bluetooth 5 allowed a single PHY, operating at 1 MSym/sec.

As we've mentioned previously, Bluetooth 5 introduced two new (optional) PHYs:

2M PHY

Used to achieve twice the speed of earlier versions of Bluetooth. It offers a couple of extra benefits as well:

- Reduced power consumption, since the same amount of data is transmitted in less time, thus reducing radio-on time.
- Improvement of wireless coexistence because of the decreased radio-on time.

One downside to using the 2M PHY is that it has the potential of reducing the range, as the higher speed results in a decrease in radio sensitivity on the receiving end. Another is that the use of the 2M PHY is restricted to the secondary advertisement and data channels. It's important to note that this new PHY represents a hardware change, so older chipsets and modules may not support it.

Coded PHY

Used to achieve four times the range of earlier versions of Bluetooth. The obvious benefit of using the **coded PHY** is increased range, with two trade-offs:

- **Higher power consumption:** due to the fact that we are transmitting multiple symbols to represent one bit of data, resulting in longer radio-on time to transmit the same amount of data.
- **Reduced speeds:** due to the fact that more bits are needed to transmit the same amount of data (125 kbps or 500 kbps, depending on the **coding scheme** used — *explained below*).

Ranges as far as 800 meters line-of-sight have been recorded while testing with the coded PHY ([see this YouTube video](#)). This makes it possible to use BLE in applications such as ones that require communication with a device hundreds of meters away.

The data rates we discussed above define the rate at which the radio transmits raw data. When it comes to the application data rate — in terms of how much bandwidth your application can utilize — these numbers are reduced.

This is due to mandatory (time) gaps in between packets (150 microseconds, per the Bluetooth specification), packet overhead, as well as some other requirements defined by the specification (for specific use cases such as responses and confirmation packets).

As an example, in the case of the 2M PHY, one can achieve a maximum application data rate of around 1.4 Mbps ([based on data transfer experiments run between two nRF52840 boards](#)).

2M PHY

At the application level, you do not need to know much about the low-level details of this PHY, other than setting it when you want to achieve higher speeds. But keep in mind that using this PHY potentially reduces the range. Another restriction that was mentioned above is that the **2M PHY** is not allowed in primary advertisements. There are two ways to utilize this mode:

- Secondary advertisements (**extended advertising mode**) are used and sent on the 2M PHY, which allow a connection on that PHY from the central device.
- Advertising on the primary or secondary channels using the 1M or the coded PHY. A connection is then established, and either side can request a PHY update to use the 2M PHY during the connection.

One important thing to note is that the link between a peripheral and a central can be asymmetric, meaning that the packets from the peripheral can be sent using the 1M PHY, while packets from the central can be sent using the 2M PHY.

Coded PHY

As mentioned earlier, Bluetooth 5 achieves the longer range compared to earlier versions of Bluetooth by introducing the new coded PHY. So, what does **coding** mean? And how does it help achieve a longer range of communication?

It achieves this by utilizing a telecommunications technique called **Forward Error Correction (FEC)**. FEC allows the receiver to recover the data from errors that occur due to noise and interference. It accomplishes this by introducing redundancy in the data being transmitted, using a specific algorithm. So, instead of requiring retransmission of data when an error occurs, the receiver can recover the originally transmitted data by utilizing the redundancy in the data.

There are two coding schemes used by the coded PHY:

- **S = 2**, where 2 symbols represent 1 bit therefore supporting a bit rate of **500 kbps**.
- **S = 8**, where 8 symbols represent 1 bit therefore supporting a bit rate of **125 kbps**.

Eight Times the Advertising Capacity

Extended Advertisements

Bluetooth 5 introduced the concept of **secondary advertising channels** which allow the device to offload data to advertise more data than what's allowed on the **primary advertisement channels**. Advertisements that are transmitted only on the primary advertisement channels are called **legacy advertisements**, whereas advertisements that start by transmission on the primary channels and then continue on the secondary channels are called **extended advertisements**.

In the case of extended advertisements, the advertisement packets sent on the primary advertisement channels provide the information necessary to discover the offloaded advertisements that are sent on the **secondary advertisement channels**. These are utilized for sending significantly more data (8x) than legacy advertisements allow (up to 255 bytes vs. 31 bytes). They are also useful in reducing congestion on the three primary advertising channels.

Advertisement packets sent on the secondary advertisement channels can use any of the three PHYs (1M PHY, 2M PHY, or coded PHY), whereas the primary advertisement channels can only use the coded PHY or the original 1M PHY. This means that a central must use the 1M PHY or coded PHY when initially searching for peripherals that are sending out advertising packets.

Periodic Advertisements

Think of the following use case: we have multiple temperature sensors distributed in a building. The temperature readings from these sensors change over time, and they need to be distributed along with other data (location, time of reading, etc.) to multiple devices that pass this data up to the cloud.

We could potentially use extended advertisements since we may have more data than would fit into the legacy advertisement packet (31 bytes), but that means the centrals will have to be looking for advertisements all the time, potentially consuming a lot of power (especially if the advertisement data does not change often).

Instead, we could utilize a new feature that was introduced in Bluetooth 5: **periodic advertisements**. Periodic advertisements are a special case of **extended advertisements** and allow a central to “synchronize” to a peripheral that is sending these extended advertisements at a fixed interval. This helps reduce power consumption when the advertisements are sent periodically at longer intervals, while allowing multiple centrals to be synchronized to the same peripheral.

The way periodic advertisements work is by transmitting advertising packets on the primary advertisement channels, which hold information (e.g., time offset, PHY, etc.) to help locate the extended advertisement packet. That packet, in turn, contains fields that define the data needed to synchronize to the periodic advertisement packets — similar to how connections are synchronized using a channel map, hop increment, the selected PHY, etc.

More on Extended Advertisements

Extended advertisements utilize the **Secondary Advertisement Channels**, which are the same channels used by data packets transmitted during a connection between two devices. Extended advertisements are not considered part of the advertisement events we talked about previously (also called **legacy advertisements**), which occur on the primary advertising channels (37, 38, and 39).

Extended advertisements are used to “offload” data that would otherwise exist on the Primary Advertising Channels — also called **auxiliary packets**. Offloading is accomplished by first advertising on the primary channel data values that point to an auxiliary packet on the secondary channel. The advertisement packets sent on the primary channels contain the PHY channel and the offset to the start time of the extended advertisement packet.

Another important aspect is that extended advertisements can use any of the three PHYs (1M PHY, 2M PHY, or the coded PHY), whereas primary advertisement packets can only be sent using the 1M PHY or the coded PHY.

Since **non-Bluetooth 5** devices are not able to discover extended advertisements, it is recommended that peripherals also use an advertising set (additional advertisements) with legacy advertising PDUs for older central devices to be able to discover these peripherals. Here’s a diagram showing an example of extended advertising:

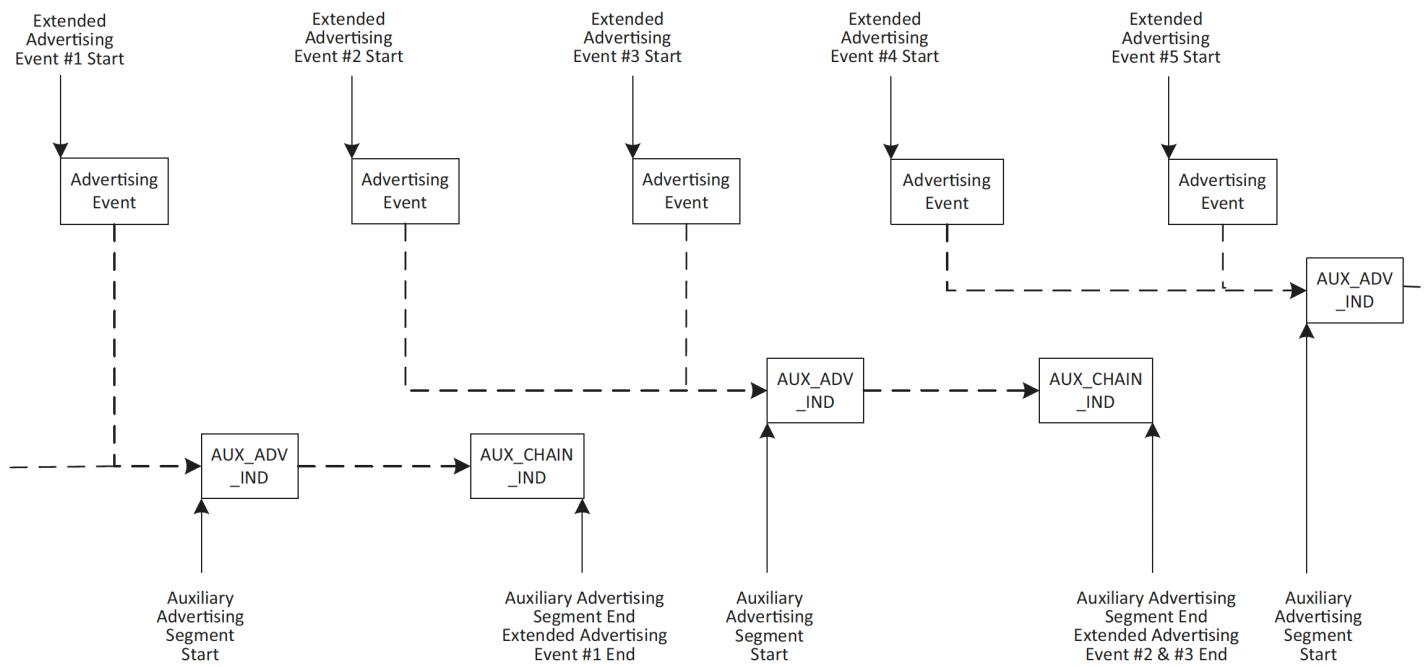


Figure 55: Extended advertising
(Source: Bluetooth 5 specification document)

The periodic advertising mode allows two or more devices to communicate in a connection-less manner. The peripheral device sends out synchronization information along with the other extended advertisement data allowing another device to become **synchronized** with this peripheral. This synchronization allows devices to receive the peripheral device's extended advertisements at regular, deterministic intervals. Here's a diagram showing an example of periodic advertising:

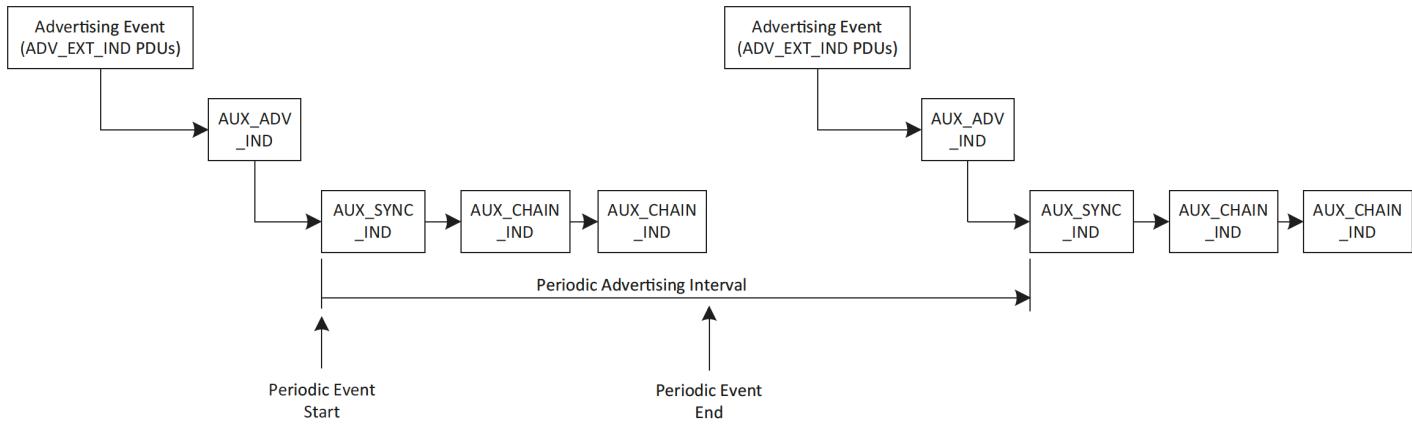


Figure 56: Periodic advertising
(Source: Bluetooth 5 specification document)

Bluetooth 5 Example: High-Speed and Long-Range Modes

For this example, we'll be utilizing the `ble_app_att_mtu_throughput` example located under the `nRF5_SDK_current/examples/ble_central_and_peripheral/experimental` folder in the GitHub repository accompanied with the book (refer to the **GitHub Repository** chapter). This example demonstrates utilizing two of the three major new features in Bluetooth 5:

- **2M PHY for 2x speed**
- **Coded PHY for 4x range**

You will need **two nRF52840 development kits** to run the example and be able to demonstrate both PHYs. It works as follows:

- You have the two development boards flashed with the same firmware.
- You will set one of them to be the Central, and the other as Peripheral.
- For the Peripheral board, this is done by pressing **Button 4** on the development board.
- For the Central board, you need to press **Button 3** on the development board.
- You need to have the Central board connected to a computer and Terminal program to interact, configure, and start the test.
- There are many Serial Terminal programs, however the one I recommend, and have found to work best with this example, is a program called **ZOC7**, available for download [here](#) (runs on **Windows** and **macOS**).
- The application sends **1024 Kbytes** by default.
- The application uses the 2M PHY as the default for the PHY Update Request message. That is, if you do not configure the PHY explicitly, the 2M PHY will be used for the test.
- The data transfer is done by the central exposing a service called the "AMT" or "Application MTU Throughput" service. The Peripheral subscribes to notifications from the Central's AMT Characteristic. Data transfer is then performed by the Central sending a Notification to the Peripheral including the data to be transferred. The advantage of using Notifications is that they do not require "acknowledgement" packets (Confirmations), which reduces the overhead and delays in transferring the next packet from the Central — in order to achieve the highest transfer speed.
- **NOTE:** This is the first time we've used a Central device in the Server Role. While most uses have a Peripheral act in the Server Role and the Central as the Client, either the Central or Peripheral device can be a Client or a Server.

Important Note:

In BLE, you can have a connection configured asymmetrically with different PHYs for transfers from each end. For example, the connection could be configured to use the 2M PHY for transfers from the Central and Coded PHY for transfers sent from the Peripheral.

In this example, the transfer from the Peripheral is always set to use the 2M PHY. (This does not affect the speed of transfer since all the data is sent from the Central to the Peripheral)

We'll briefly go over how to run the example, and then walk through some of the code that relates to Bluetooth 5 features.

Here's a link to the blog post by Nordic Semiconductor that describes the example:

[Throughput and long range demo blog post](#)

Running the Example

To get started:

- Install ZOC
- Launch Segger Embedded Studio (SES)
- Navigate to File → Open Solution:

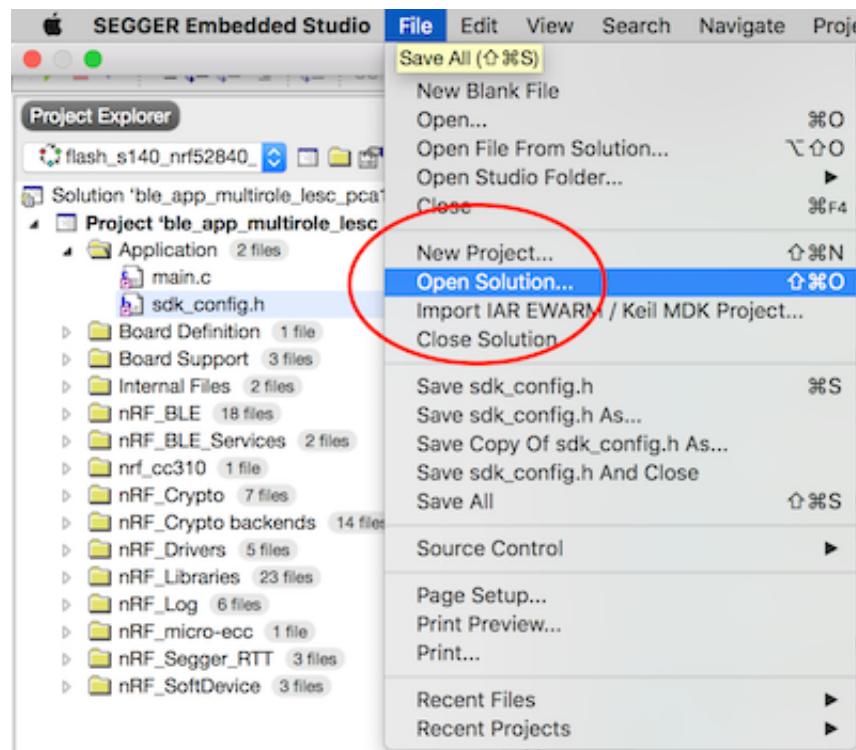
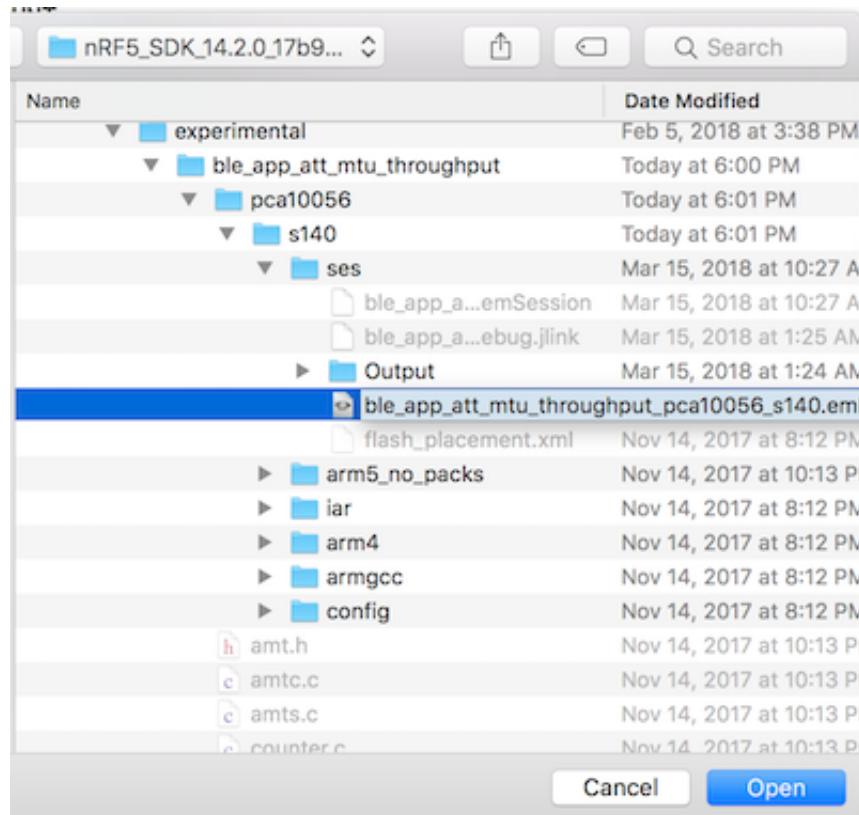


Figure 57: Open Solution (SES)

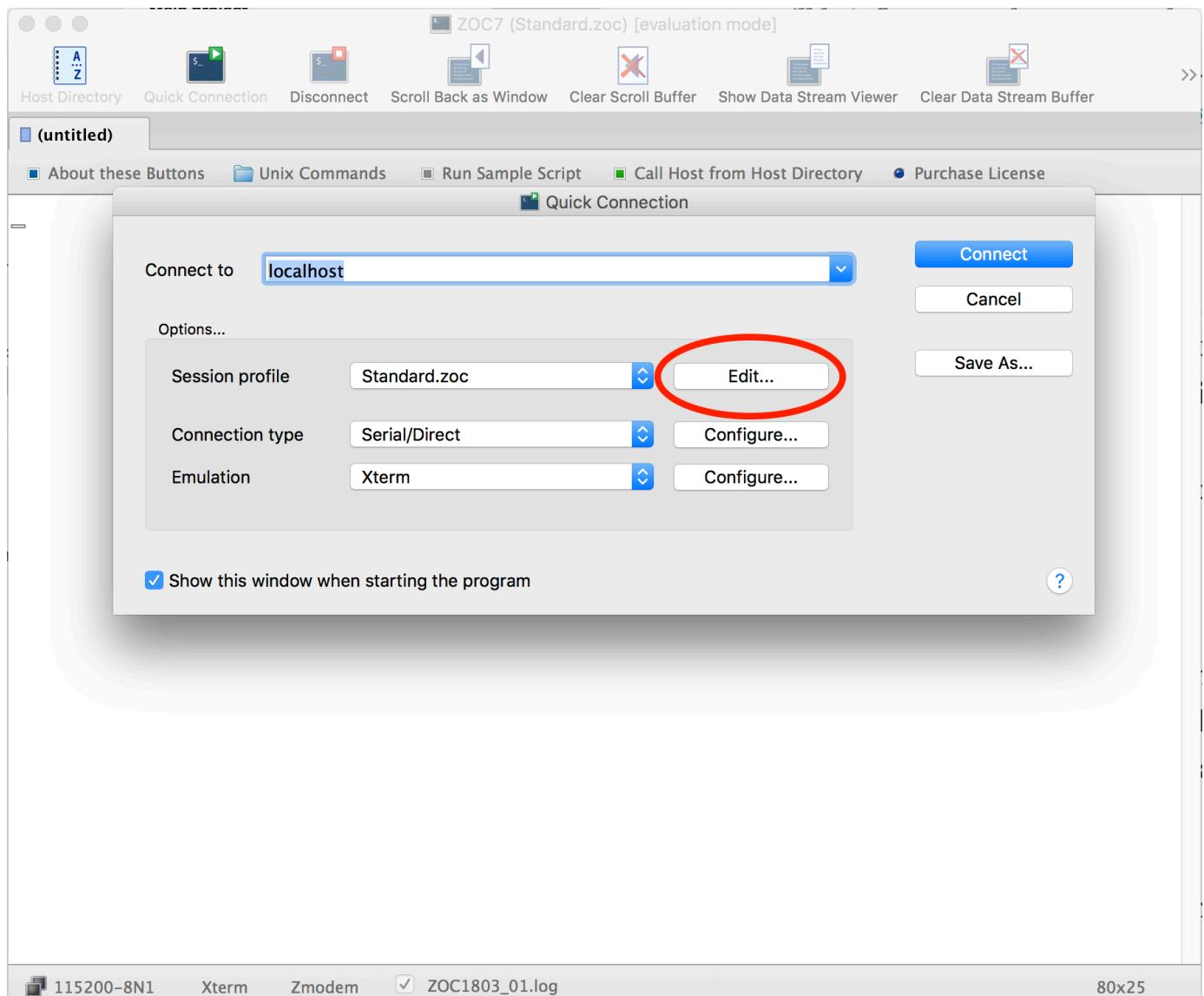
- Locate the project file for the example:

*Figure 58: Locate SES Project File*

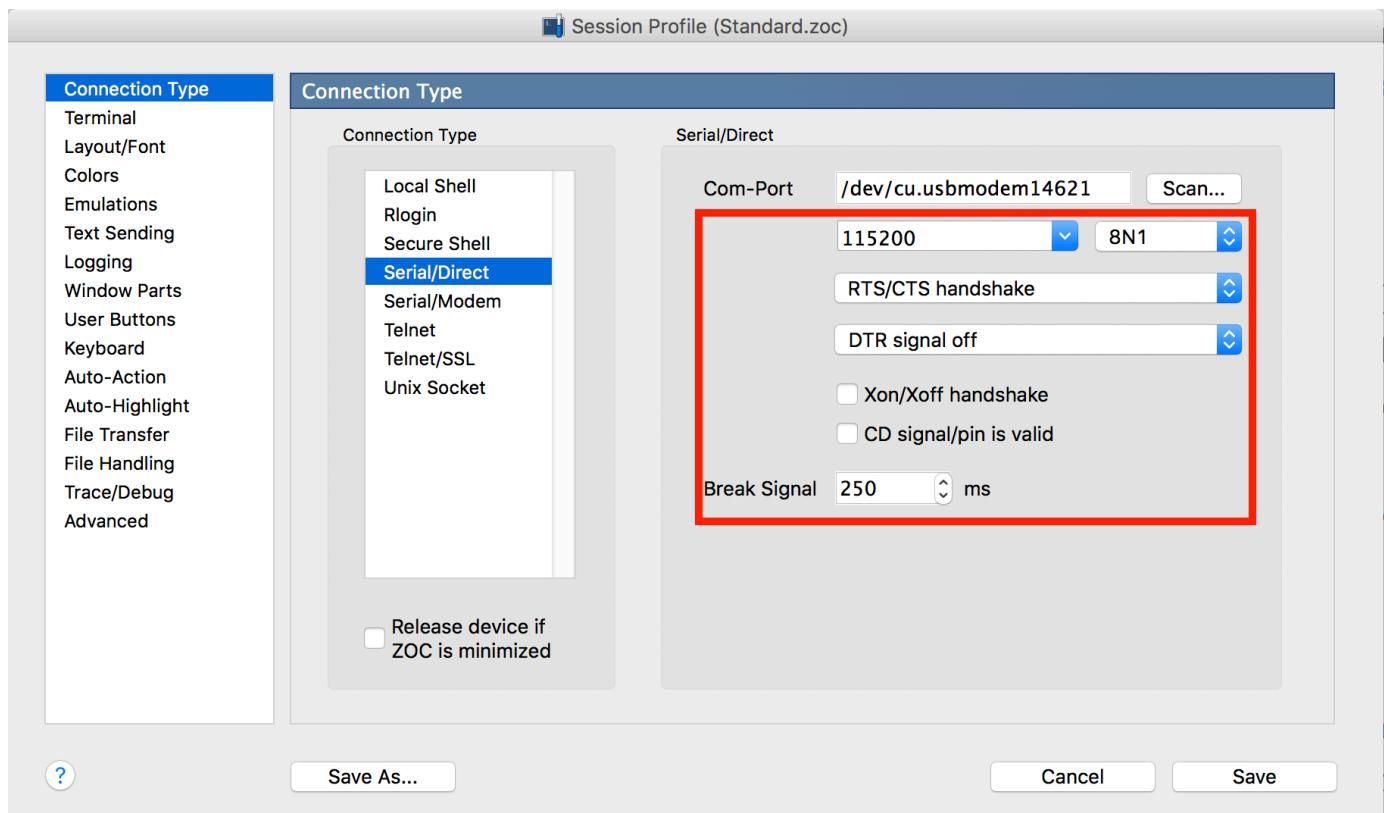
- Click Open
- Build the project
- Connect only one board at a time (to make it easier). And repeat the next step for each of the boards.
- Flash **each** board by pressing the **Start Without Debugging** button (labeled !) in the top toolbar:

*Figure 59: Start without Debug Button*

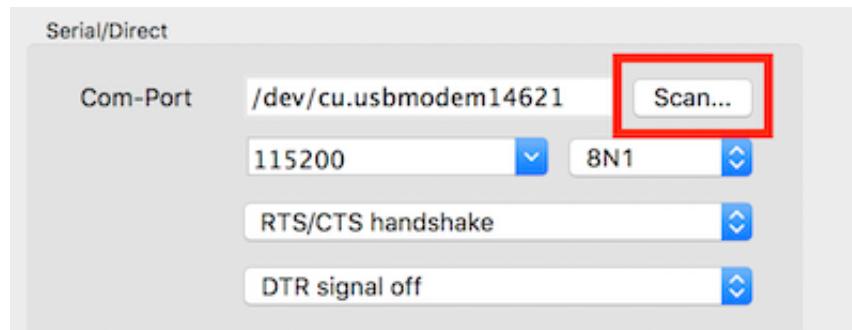
- Once the program is flashed to the development boards, connect the one you'd like to use as a central to your computer (via the USB connection).
- Start the ZOC program. You will be presented with the following screen. Hit the "Edit" button:

*Figure 60: ZOC Edit Session Profile*

- Once you hit Edit, you will be presented with the connection settings. Make sure you have settings that match those in the screenshot except the **Com-Port** which may differ:

*Figure 61: ZOC Serial Configuration*

- Hit the Scan button:

*Figure 62: ZOC Scan Option*

- If you have the board connected to your computer correctly, you should see a serial port connection in the list:

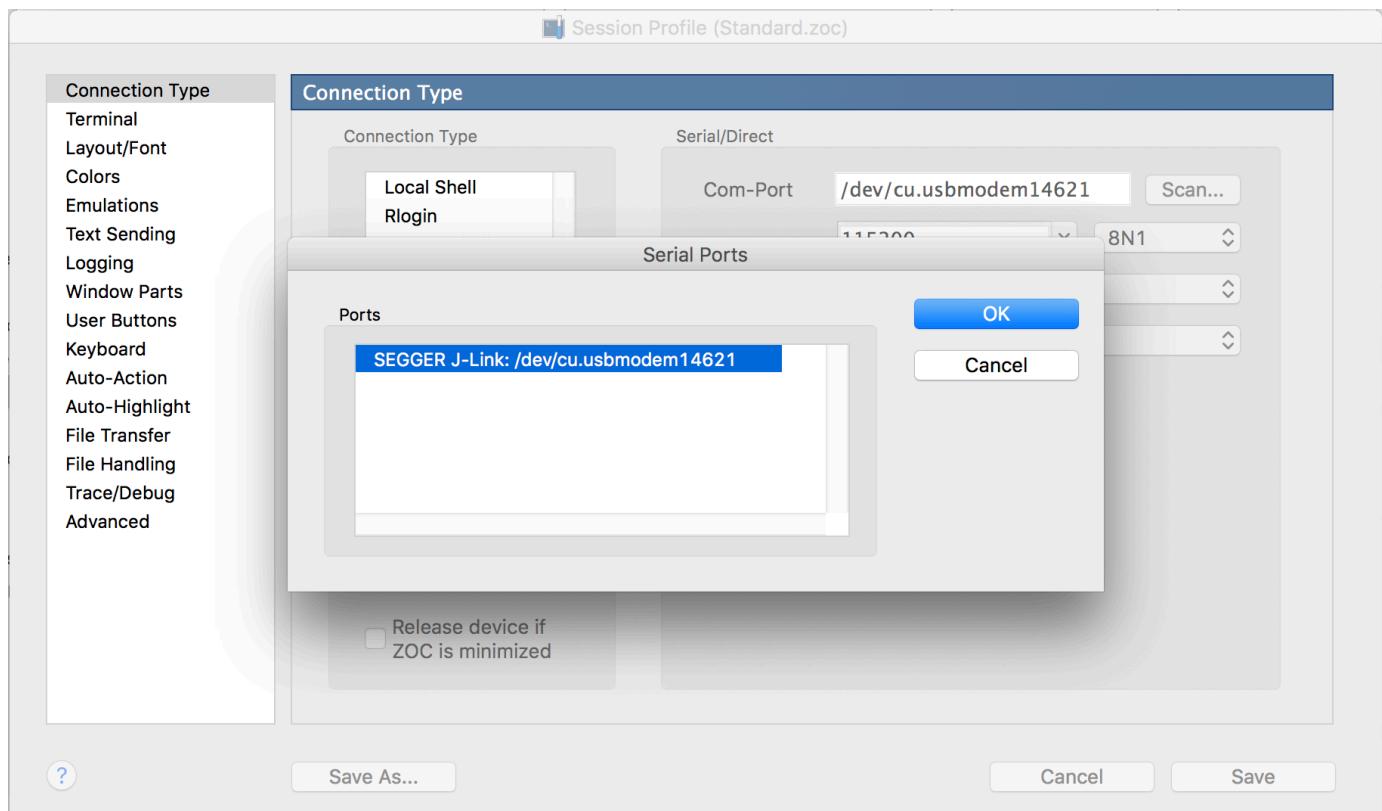


Figure 63: ZOC Serial Ports List

- Click on the port and press OK
- You may be presented with the following message. Click "No":



Figure 64: RTS/CTS Handshake Dialog Popup

- The board may need to be restarted to display the startup command interface. Hit the **BOOT/RESET** button on the development kit:

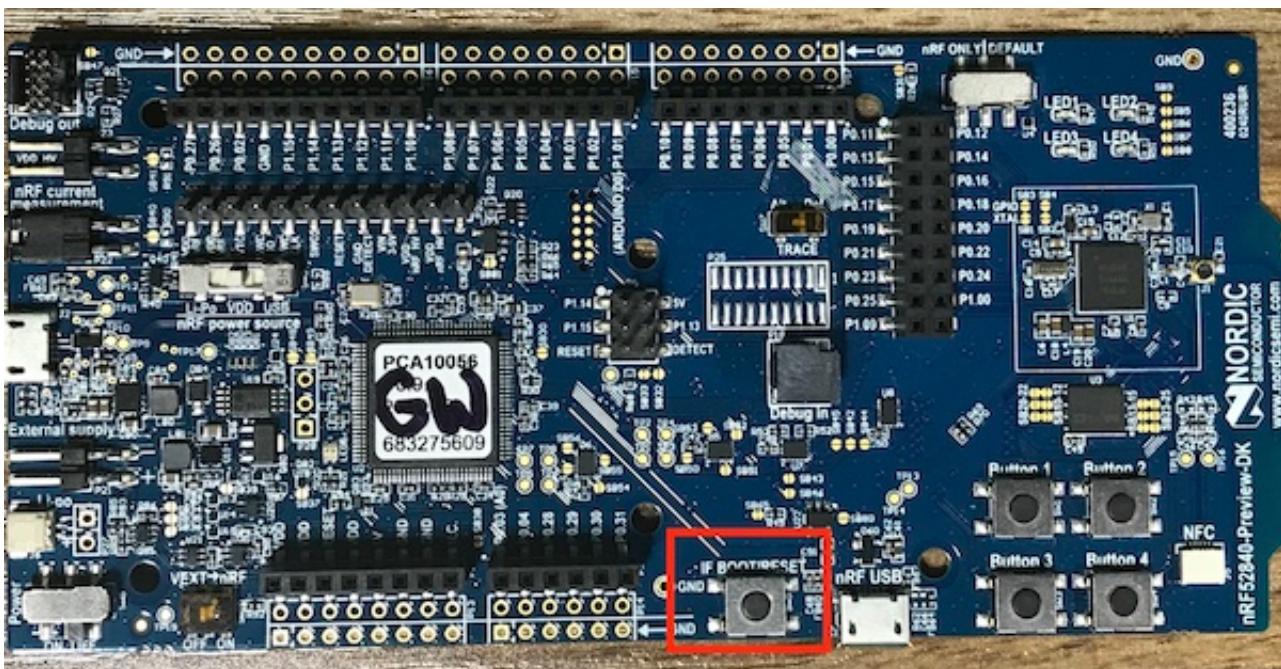


Figure 65: nRF52-DK Reset Button

- Now, you should see the startup terminal command line interface in ZOC:

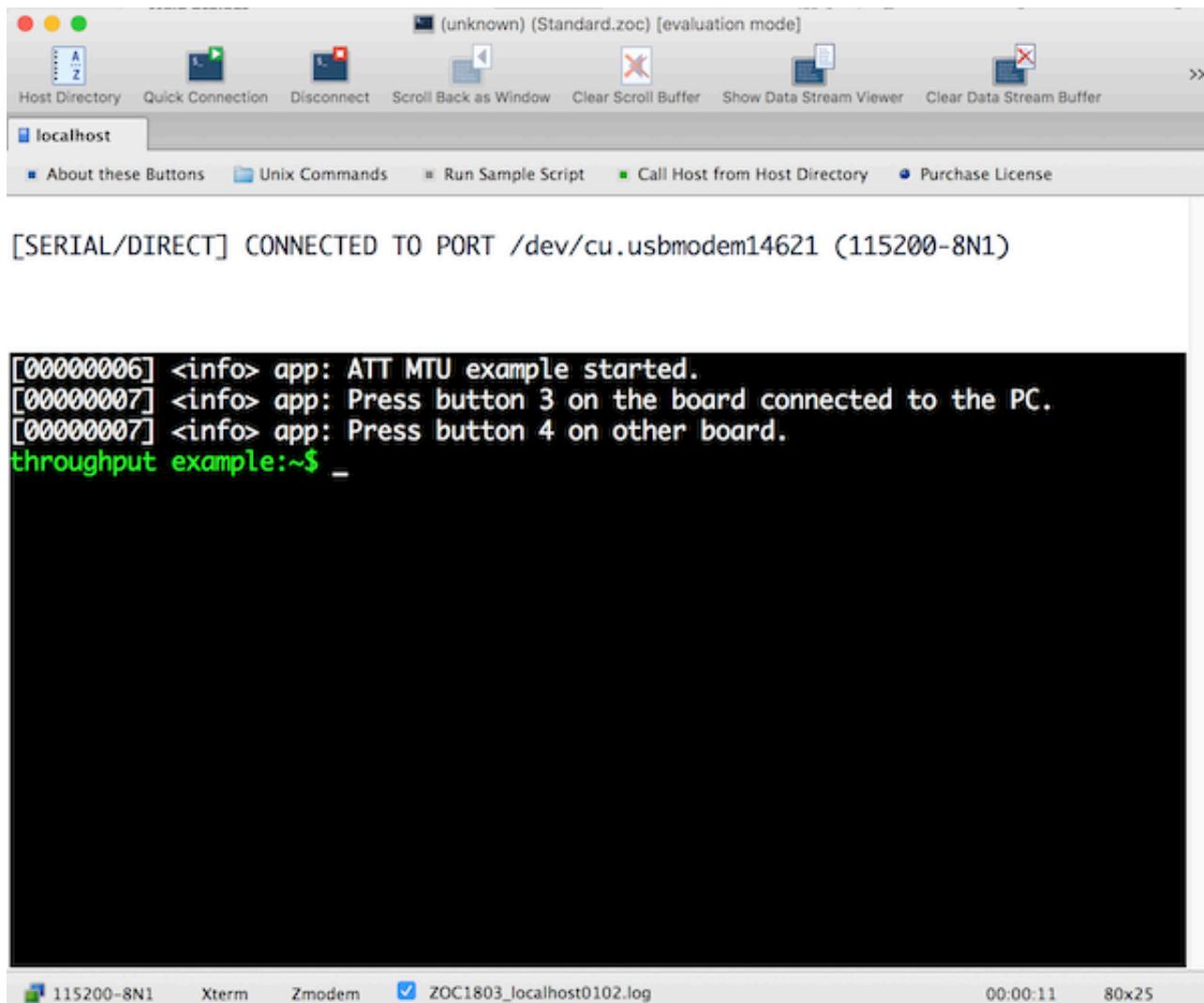


Figure 66: ZOC Serial Connected Successfully

- Hit Button 3 on the Central mode development kit (the one you connected to your computer):

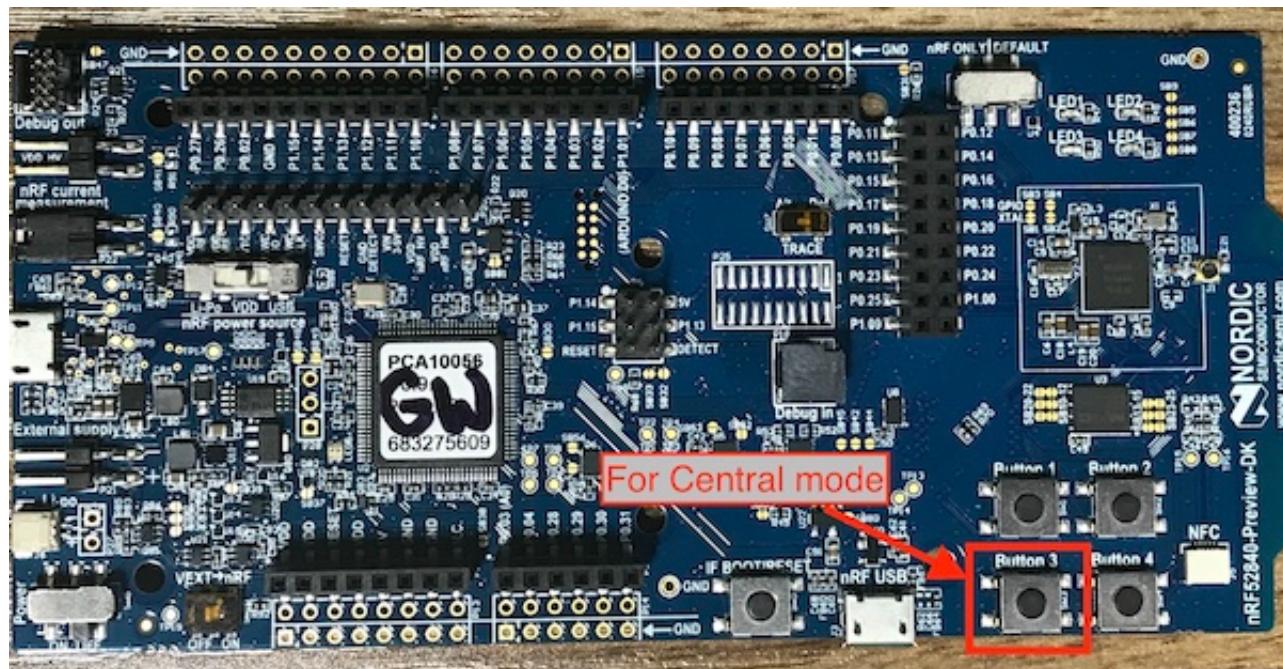


Figure 67: nRF52-DK Button 3

- Hit Button 4 on the Peripheral mode development kit (no need to connect this one to your computer):

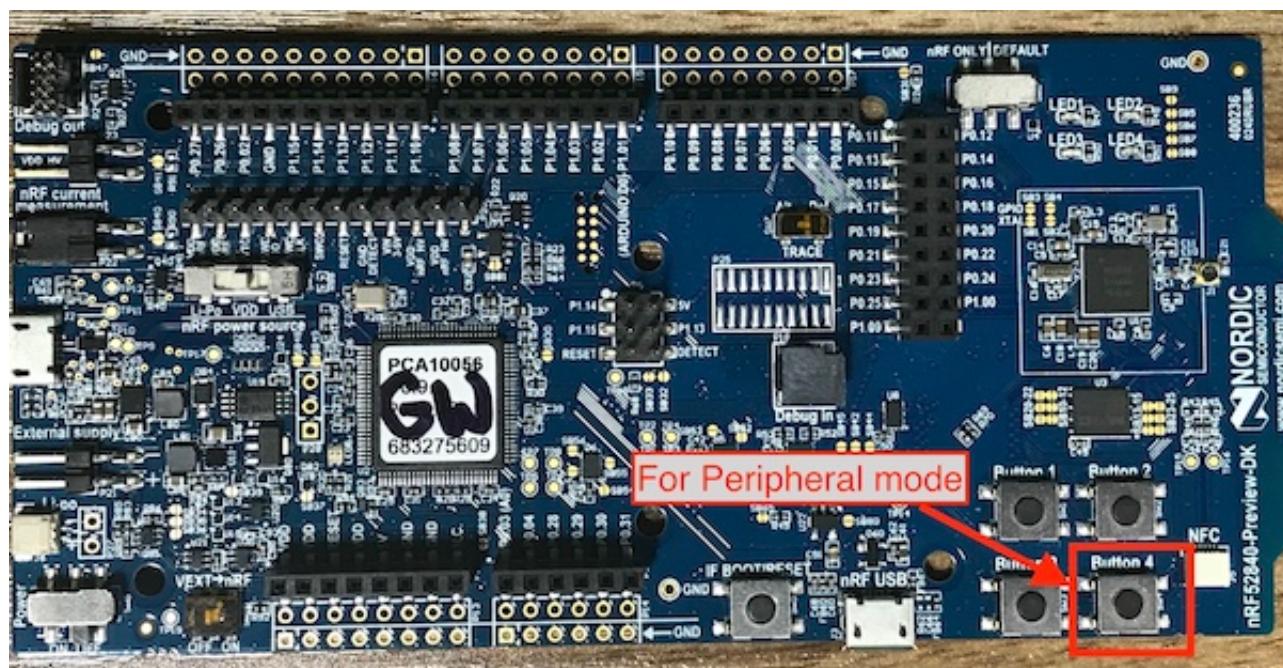


Figure 68: nRF52-DK Button 4

- You should now see LED #1 on the peripheral board light up:

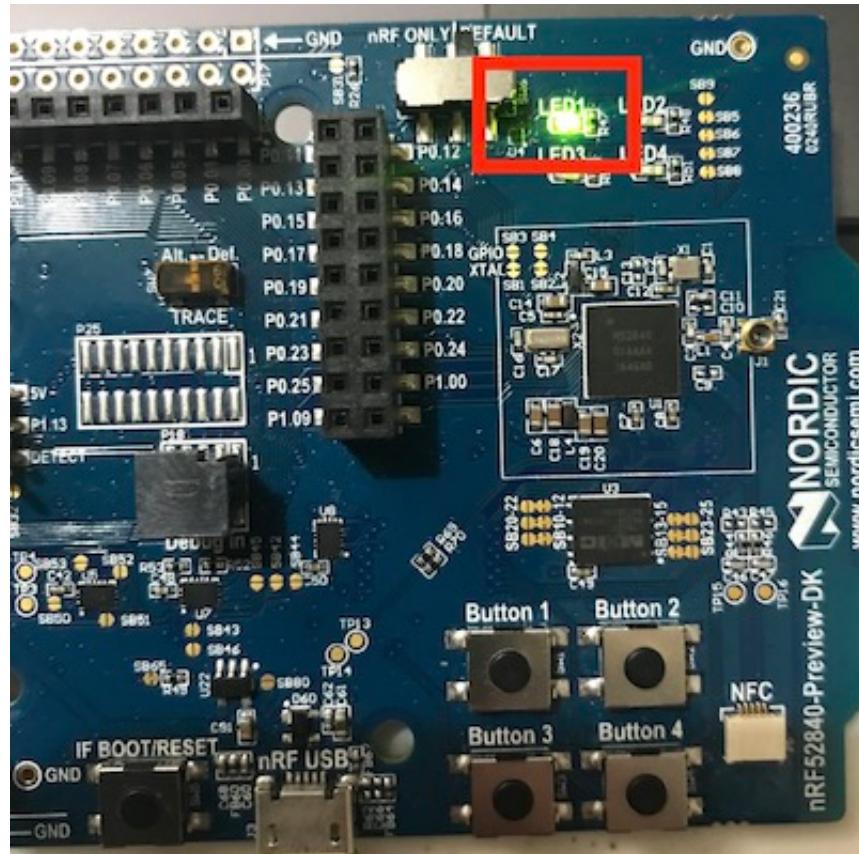
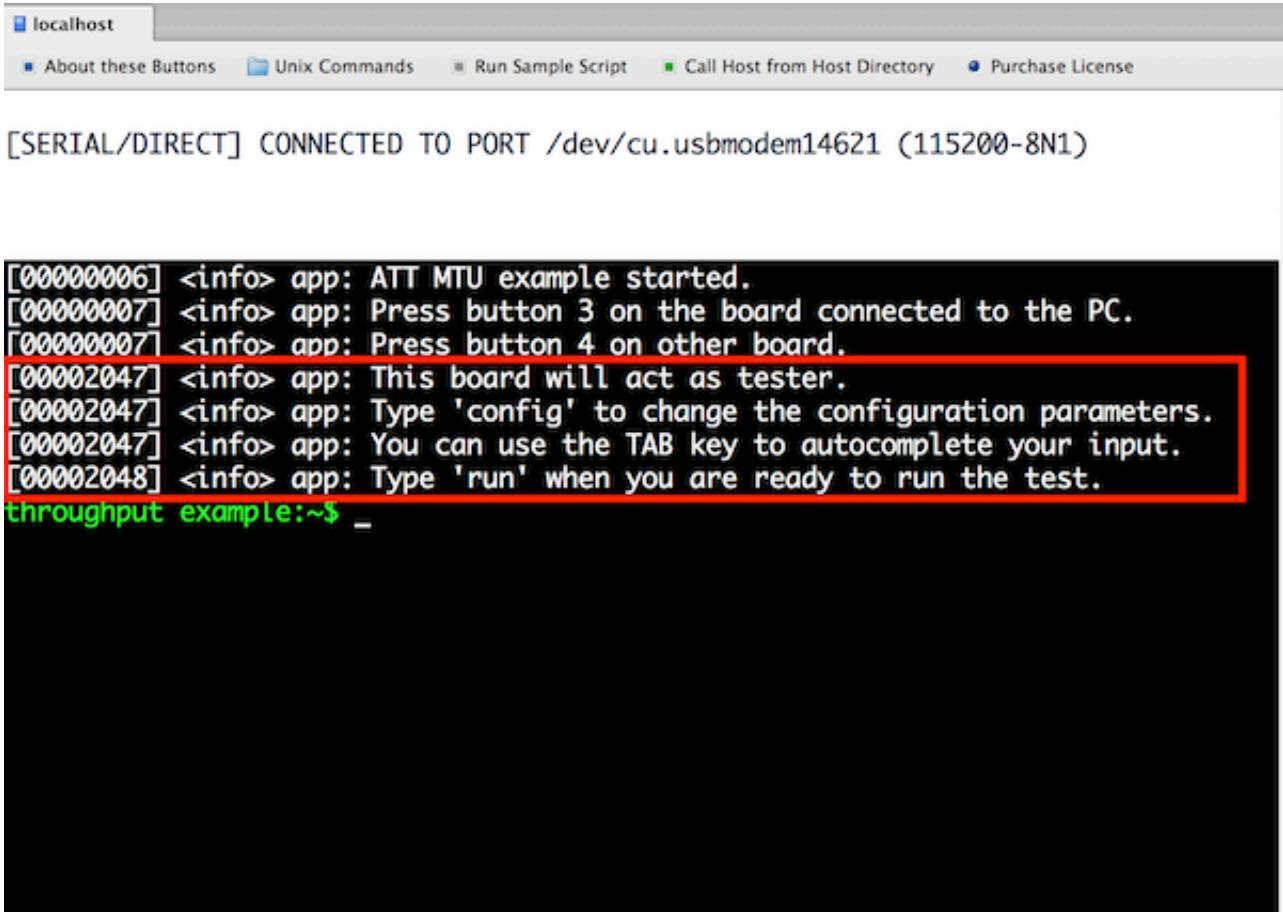


Figure 69: nRF52-DK LED #1

- Now, in ZOC, you should see the following message show up indicating that the connected board is acting as the Central device:



The screenshot shows a software interface with a toolbar at the top containing buttons for 'localhost', 'About these Buttons', 'Unix Commands', 'Run Sample Script', 'Call Host from Host Directory', and 'Purchase License'. Below the toolbar, a message '[SERIAL/DIRECT] CONNECTED TO PORT /dev/cu.usbmodem14621 (115200-8N1)' is displayed. A red box highlights several debug messages from the application:

```
[00000006] <info> app: ATT MTU example started.  
[00000007] <info> app: Press button 3 on the board connected to the PC.  
[00000007] <info> app: Press button 4 on other board.  
[00002047] <info> app: This board will act as tester.  
[00002047] <info> app: Type 'config' to change the configuration parameters.  
[00002047] <info> app: You can use the TAB key to autocomplete your input.  
[00002048] <info> app: Type 'run' when you are ready to run the test.
```

At the bottom of the terminal window, the prompt 'throughput example:~\$' is visible.

Figure 70: Board Debug message

- Next, type `config` to see the different options for configuration of the test:

```
throughput example:~$ config
config - Configure the example
Options:
-h, --help :Show command help.
Subcommands:
att_mtu      :Configure ATT MTU size
conn_interval :Configure GAP connection interval
dle           :Enable or disable Data Length Extension
print         :Print current configuration
phy           :Configure preferred PHY
throughput example:~$ _
```

Figure 71: Test Config Options

- These commands are case-sensitive to make sure you type them exactly as they are displayed.
- You can press the TAB button your keyboard to autocomplete commands and display the available ones as well.
Test case #1: 2M PHY - high speed test
- For this test case, type `config phy 2M`:

```
throughput example:~$ config phy 2M
Preferred PHY set to 2 Mbps.
throughput example:~$ _
```

Figure 72: 2M PHY Configuration

- Next, type `run`. You should see LEDs 2 & 3 lit up:

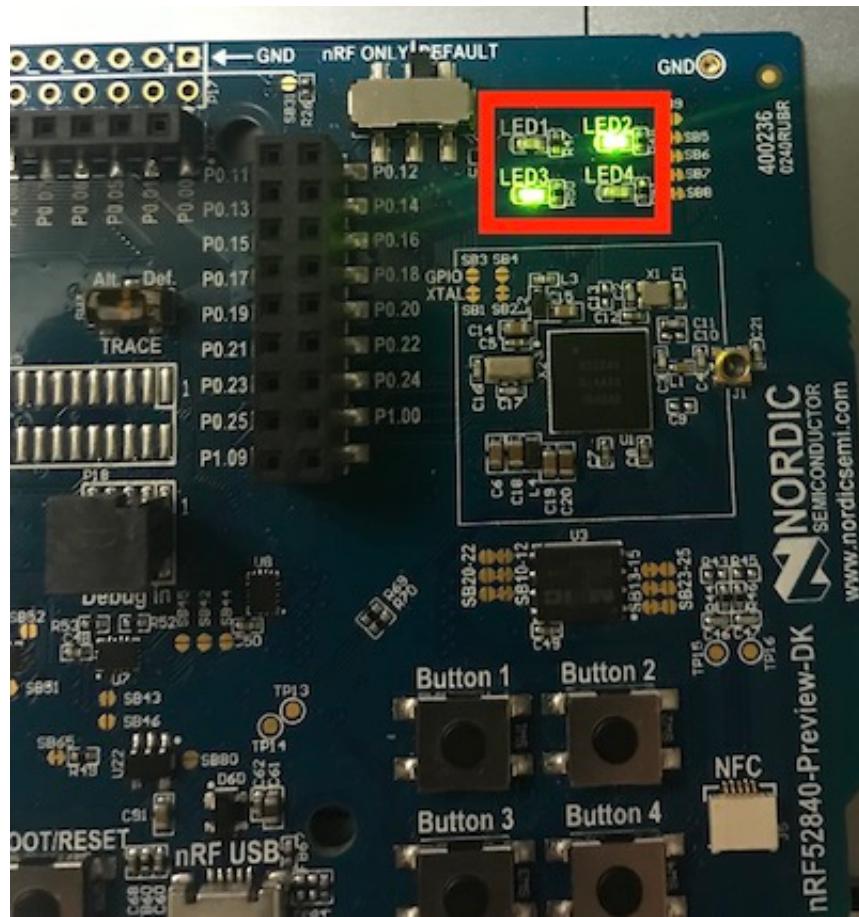


Figure 73: LED 2 & 3 On

- You should also see the transfer start with the number of bytes displayed in the Terminal program, and finally the transfer will complete with a final summary of the speed shown:

```

[00001987] <info> app: Sent 1010 KBytes
[00002233] <info> app: Sent 1011 KBytes
[00002524] <info> app: Sent 1012 KBytes
[00002770] <info> app: Sent 1013 KBytes
[00003016] <info> app: Sent 1014 KBytes
[00003262] <info> app: Sent 1015 KBytes
[00000457] <info> app: Sent 1016 KBytes
[00000749] <info> app: Sent 1017 KBytes
[00000994] <info> app: Sent 1018 KBytes
[00001240] <info> app: Sent 1019 KBytes
[00001486] <info> app: Sent 1020 KBytes
[00001732] <info> app: Sent 1021 KBytes
[00002025] <info> app: Sent 1022 KBytes
[00002271] <info> app: Sent 1023 KBytes
[00002516] <info> app: Sent 1024 KBytes
[00002622] <info> app: Done.
[00002622] <info> app: =====
[00002622] <info> app: Time: 8.755 seconds elapsed.
[00002623] <info> app: Throughput: 958.27 Kbps.
[00002623] <info> app: =====
[00002624] <info> app: Sent 1048712 bytes of ATT payload.
[00002624] <info> app: Retrieving amount of bytes received from peer...
[00003082] <info> app: Peer received 1048712 bytes of ATT payload.
[00003082] <info> app: Disconnecting...
throughput example:~$
```

115200-8N1 Xterm Zmodem ZOC1803_localhost01.log 00:03:55 80x25

Figure 74: 2M PHY Test Summary

Test case #2: Coded PHY - long range test

- Reset the Central board.
- Type `config phy coded` :

```

throughput example:~$ config phy coded
Preferred PHY set to Coded.
throughput example:~$
```

Figure 75: Coded PHY Configuration

- Type `run`, and the transfer should start. When it completes, it will also display a summary. Notice that the rate is significantly lower using the Coded PHY:

```
[00002492] <info> app: Sent 1021 KBytes
[00002089] <info> app: Sent 1022 KBytes
[00002279] <info> app: Sent 1023 KBytes
[00002220] <info> app: Sent 1024 KBytes
[00001381] <info> app: Done.
[00001382] <info> app: =====
[00001382] <info> app: Time: 326.629 seconds elapsed.
[00001383] <info> app: Throughput: 25.68 Kbps.
[00001383] <info> app: =====
[00001383] <info> app: Sent 1048712 bytes of ATT payload.
[00001384] <info> app: Retrieving amount of bytes received from peer...
[00002656] <info> app: Peer received 1048712 bytes of ATT payload.
[00002657] <info> app: Disconnecting...
throughput example:~$
```

Figure 76: Coded PHY Test Summary

- This test would mostly be useful for testing with a long distance between the two boards.

Source Code Walkthrough

Let's go through some of the source code and explain how the PHYs are setup. We'll be referring to only the parts of the code that relate to the PHY setup.

From `main.c`:

- We are using the nRF52840 development kit, so the SoftDevice used is the S140, so we'll only be looking at the code that applies to that SoftDevice: `#if defined(S140)`
- The following configuration defines the preferred PHYs that are to be used. It is set to allow connections using any of the three PHYs:

```
static test_params_t m_test_params =
{
...
#if defined(S140)
    .phys.tx_phys      = BLE_GAP_PHY_2MBPS | BLE_GAP_PHY_1MBPS | BLE_GAP_PHY_CODED,
    .phys.rx_phys      = BLE_GAP_PHY_2MBPS | BLE_GAP_PHY_1MBPS | BLE_GAP_PHY_CODED,
#else
...
};
```

- Here, in the function `on_ble_gap_evt_connected(ble_gap_evt_t const * p_gap_evt)`, we have a request to update the preferred PHYs when we are in the Peripheral role: `if (m_gap_role == BLE_GAP_ROLE_PERIPH) { #if defined(S140) err_code = sd_ble_gap_phy_request(p_gap_evt->conn_handle, &m_test_params.phys); APP_ERROR_CHECK(err_code); #else }`
- Later in the code, we see a function that sets the preferred PHYs:

```
void preferred_phy_set(ble_gap_phys_t * p_phy)
{
#if defined(S140)
    ble_opt_t opts;
    memset(&opts, 0x00, sizeof(ble_opt_t));
    memcpy(&opts.gap_opt.preferred_phys, p_phy, sizeof(ble_gap_phys_t));
    ret_code_t err_code = sd_ble_opt_set(BLE_GAP_OPT_PREFERRED_PHYS_SET, &opts);
    APP_ERROR_CHECK(err_code);
#endif
    memcpy(&m_test_params.phys, p_phy, sizeof(ble_gap_phys_t));
}
```

- In `test_begin()`, we have a default set for the Peripheral's **transfer** PHY. In this case, it has the 2M PHY set as the default PHY for transfers (from the peripheral device). This does not affect the transfer — as we mentioned previously — since the data transfer is all done from the central device.
- The preferred PHYs set for receiving data at the Peripheral end are left at the default setting in the `m_test_params` data structure (which sets it to all three PHYs).

```
switch (m_gap_role)
{
...
    case BLE_GAP_ROLE_PERIPH:
        advertising_start();
        m_test_params.phys.tx_phys = BLE_GAP_PHY_2MBPS;
        break;
...
}
```

- The Central has all PHYs enabled (via the `m_test_params` data structure) for both data **transfer** and **receive**.

From `example_cmds.c`:

```
static void cmd_phy_1m_set(nrf_cli_t const * p_cli, size_t argc, char ** argv)
{
    phy_set(p_cli, BLE_GAP_PHY_1MBPS);
}
```

```

static void cmd_phy_2m_set(nrf_cli_t const * p_cli, size_t argc, char ** argv)
{
    phy_set(p_cli, BLE_GAP_PHY_2MBPS);
}

#ifndef NRF52840_XXAA
static void cmd_phy_coded_set(nrf_cli_t const * p_cli, size_t argc, char ** argv)
{
    phy_set(p_cli, BLE_GAP_PHY_CODED);
}
#endif

NRF_CLI_CREATE_STATIC_SUBCMD_SET(m_prephy_cmds)
{
    NRF_CLI_CMD(1M,     NULL, "Set preferred PHY to 1Mbps", cmd_phy_1m_set),
    NRF_CLI_CMD(2M,     NULL, "Set preferred PHY to 2Mbps", cmd_phy_2m_set),
#ifndef NRF52840_XXAA
    NRF_CLI_CMD(coded, NULL, "Set preferred PHY to Coded", cmd_phy_coded_set),
#endif
    NRF_CLI_SUBCMD_SET_END
};

```

- The above code sets the commands needed for the different settings for the preferred PHY.
- Once the user configures the preferred PHY via the `config phy` command, that PHY is set as the preferred PHY for both transfers and receives from the Central:

```

static void phy_set(nrf_cli_t const * p_cli, uint8_t value)
{
    ble_gap_phys_t phy =
    {
        .rx_phys = value,
        .tx_phys = value,
    };

    preferred_phy_set(&phy);
    nrf_cli_fprintf(p_cli, NRF_CLI_NORMAL, "Preferred PHY set to %s.\r\n", phy_str(phy));
}

```

- When the test is run, the preferred PHY set by the user will be used for the connection.

Example 2: Coded PHY (Long-Range) Example

For this example, we'll be referring to the code provided by Nordic at their GitHub here:

[Coded PHY UART \(long range\) demo repository](#)

We'll be going through this demo and the source code in a video tutorial provided with the e-book. The video is titled "BONUS Video #1 - Bluetooth 5 long range (Coded PHY)". You can also watch the video here:

[BONUS Video #1: Bluetooth 5 long range \(Coded PHY\)](#)

Further Examples and References

There's more to cover in terms of examples that utilize Bluetooth 5.0 features. For more examples, refer to the Novel Bits blog at www.novelbits.io/blog where I post regularly.

Some of the blog posts that you may find useful include:

- [Bluetooth 5 speed: How to achieve maximum throughput for your BLE application](#)
- [Bluetooth 5 Advertisements: Everything you need to know](#)

Power Consumption

Power is the rate, per unit time, at which electrical energy is transferred. It's measured in watts or Kilo watts — depending on the application. Measuring the average current draw is one of the simplest way to get a good idea of your device's power consumption.

In radio-related devices and applications on microcontrollers and in embedded systems, the key method to reduce power consumption it to use the radio as little as possible. In BLE terms, we minimize radio transmission, and maximize "sleep cycles". Choosing the right hardware components (chipset/module, battery, peripheral components, ...etc.) also play a big role in optimizing power consumption.

It is not the purpose of this book to cover embedded systems in general, so we will be solely focused on power consumption as it relates to BLE, how the different BLE parameters affect power consumption, and finally how to optimize these parameters to achieve longer battery life.

How BLE Achieves Low Power Consumption

Bluetooth Low Energy was designed from the ground up with low power consumption as a main concern — specifically on the peripheral side. Two of the most important aspects that help the technology achieve this goal are:

- **BLE is asymmetric by design**

In BLE, the central device is responsible for much of the processing and heavy-lifting in terms of managing connections and their different timings. This helps simplify the tasks needed to run on the peripheral side allowing it to turn off the radio, run fewer processing tasks, and sleeping more often — leading to lower power consumption.

One example is the *Slave Latency* parameter which allows the slave device to skip a number of consecutive connection intervals without sacrificing the connection — essentially allowing it to stay asleep longer and consumer less power.

- **BLE is designed for low-bandwidth applications**

The Generic Attribute Profile (GATT) provides a flexible and simple design that reduces overhead in operations involved with accessing and modifying the data exposed by a BLE device. Lower overhead in packets means lower bandwidth usage, which leads to shorter radio-on time hence lower power consumption.

Another flexibility provided by GATT is the option to use **commands** instead of **requests**. Commands do not require acknowledgments packets to confirm receipt — which also leads to more efficient use of the radio and hence lower power consumption.

So, what can you expect in terms of power consumption and actual battery life? Unfortunately, there is no simple answer to this question, but rather it depends on multiple factors, some of which are:

- The chipset/radio used.
- The BLE stack and version of Bluetooth used.
- Tuning the different BLE parameters.
- Efficiency of the firmware and code running on your device.

In this chapter, we will focus on the tuning of the different BLE parameters.

Optimizing BLE Parameters for Lower Power Consumption

Author's Note: *Put the User Experience First*

Before we get into optimizing the different BLE parameters to achieve lower power consumption, I believe there's one important aspect that all engineers and developers should keep in mind when designing a product: "Always consider the user experience aspect and keep it a high priority in the design of your product."

Now, that may have sounded a bit philosophical, but I believe engineers — *myself included* — sometimes ignore the user experience aspect of the product. We get very passionate about the product itself and the design and implementation process.

In the end, though, a finished product is not very meaningful unless it delivers a pleasant and satisfying experience to the end-user. This is especially true during the stages of optimization of the product design — where the user experience may end up being ignored or degraded. The main point here is to keep the user experience aspect in mind when optimizing for low power consumption — making sure it doesn't get sacrificed.

Advertising Parameters

- Increase the Advertising Interval while still allowing the device to be discovered and connected to in a reasonable time frame...That is, reasonable in accordance to your device's specification.
- Consider Advertising periodically instead of continuously (*turning it off when it makes sense, then back on later*).
- Optimize the size of Advertisement Data. The bigger the Advertisement Data size, the longer the radio will be turned on — leading to higher power consumption.
- Consider including the most important data in the Advertisement Packets (which occur often), and leaving the secondary/less important data to be transmitted upon demand — in Scan Response Packets (which occur less often and get triggered by the Central).
- For connected devices, increasing the Advertising Interval after a certain point may actually lead to higher power consumption: the current consumption will be lower while Advertising. However, the Central will take longer to find the Advertising Peripheral causing the Advertising duration to be higher — leading to higher power consumption.

Connection Parameters

- Increase the **Connection Interval** as much as possible while still satisfying the highest data transfer rate you'd like to achieve.
- Take advantage of a non-zero Slave Latency (to allow the Slave to sleep longer)
- When choosing a Connection Interval, keep in mind that it's possible to get multiple packet transfers per Connection Interval (*this will depend on the stack running on both sides — the Central and Peripheral, as well as some other parameters*).—
- One tip for increasing sleep cycles — especially for sensor-based devices — is to only measure and prepare data when a Client has subscribed to the associated Characteristic Value.

Data Transfers

- Enable Data Length Extensions (DLE) — which allow you to increase the MTU beyond the limit of 23 bytes. (*This feature was introduced in Bluetooth 4.2*)
- Utilize Writes Without Responses and Notifications when possible.
- Choose an MTU of 251 bytes — which reduces packet overhead.
- Combine small packets into fewer larger ones (you can even consider data compression at the application level to reduce packet sizes and radio-on time).

What to Measure?

It's difficult to compare the power consumption of a BLE device to another using one single metric. Sometimes a device gets rated by its "Peak Current". While the Peak Current plays a part in the total power consumption, a device running the BLE stack will only be consuming current at the peak level while it is transmitting.

Even in high throughput systems, a BLE device is transmitting only for a small percentage of the total time that the device is connected — in a typical application, a device running the BLE stack will spend most of the time in a sleep state between Connection Events. The primary metric that takes these other time and current measurements into account is the **Average Current** — it is this value that can be used to determine the battery life of a BLE device.

Note that a single Average Current value cannot be given for a chipset/module in its datasheet — as the Average Current is highly dependent on the connection parameters used. Any time an Average Current value is given, it is very important to understand the exact use case in which the measurement was made.

With that said, here are some other useful metrics you should keep in mind:

- The peak current is useful when comparing to what the battery vendor indicates as the recommended Maximum Peak Current. Going above this value may negatively impact the capacity and lifetime of the battery.
- How much power is used to transfer a certain amount of data.
- Sleep state current consumption.

How to Measure?

Now, let's talk about testing the power consumption of your device. The most important thing is to make sure you test in an environment as close as possible to what a user's environment would be like (real-life testing and not just in a lab setting). This will make your estimations much closer to real-life usage and ultimately make your users much happier.

- The simplest way to measure current with an oscilloscope is to use a current probe and directly monitor the current going into the system.
- If you do not have a current probe available, an easy alternative is to use a small resistor in line with the power supply input to the system. You can then use a standard oscilloscope voltage probe to measure the voltage across the resistor, and effectively measure the current by dividing the voltage by the resistance. A good resistor value to use is $10\ \Omega$, as this value is small enough that it shouldn't affect the existing circuitry, and large enough to provide a voltage that can be measured with decent precision (in addition, using a value of $10\ \Omega$ makes the calculations very simple).
- When performing measurements, it is best to use a regulated DC power supply as opposed to an actual battery. This eliminates variables that might be caused by a defective or low battery.
- Other methods of measurement include specialized hardware and software solutions provided by the vendor of the chip/module you're using. For example, Silicon Labs provides the Energy Profiler tool as part of Simplicity Studio. Nordic Semiconductor provides the Nordic Power Profiler Kit (PPK).

Practical Power Measurement Exercise

Finally, we will be taking — and comparing — current consumption measurements for multiple scenarios based on different BLE parameter values. For all these exercises, we will be using the **Nordic Power Profiler Kit (PPK)**, available [here](#).

Setup

Before attaching the Power Profiler Kit to your Nordic nRF development kit, make sure you read through the user guide — available [here](#). I **highly** recommend watching this [YouTube video](#) (from Nordic Semiconductor) as well to become more familiar with the Power Profiler Kit application through nRF Connect (*the desktop application that interfaces with the PPK hardware*).

nRF52 Power Measurement Estimator Tool

Another very useful resource for estimating power consumption is an online tool called **Online Power Profiler** also provided by Nordic Semiconductor, and available at [this link](#). Here's a screenshot of the tool:

Online Power Profiler

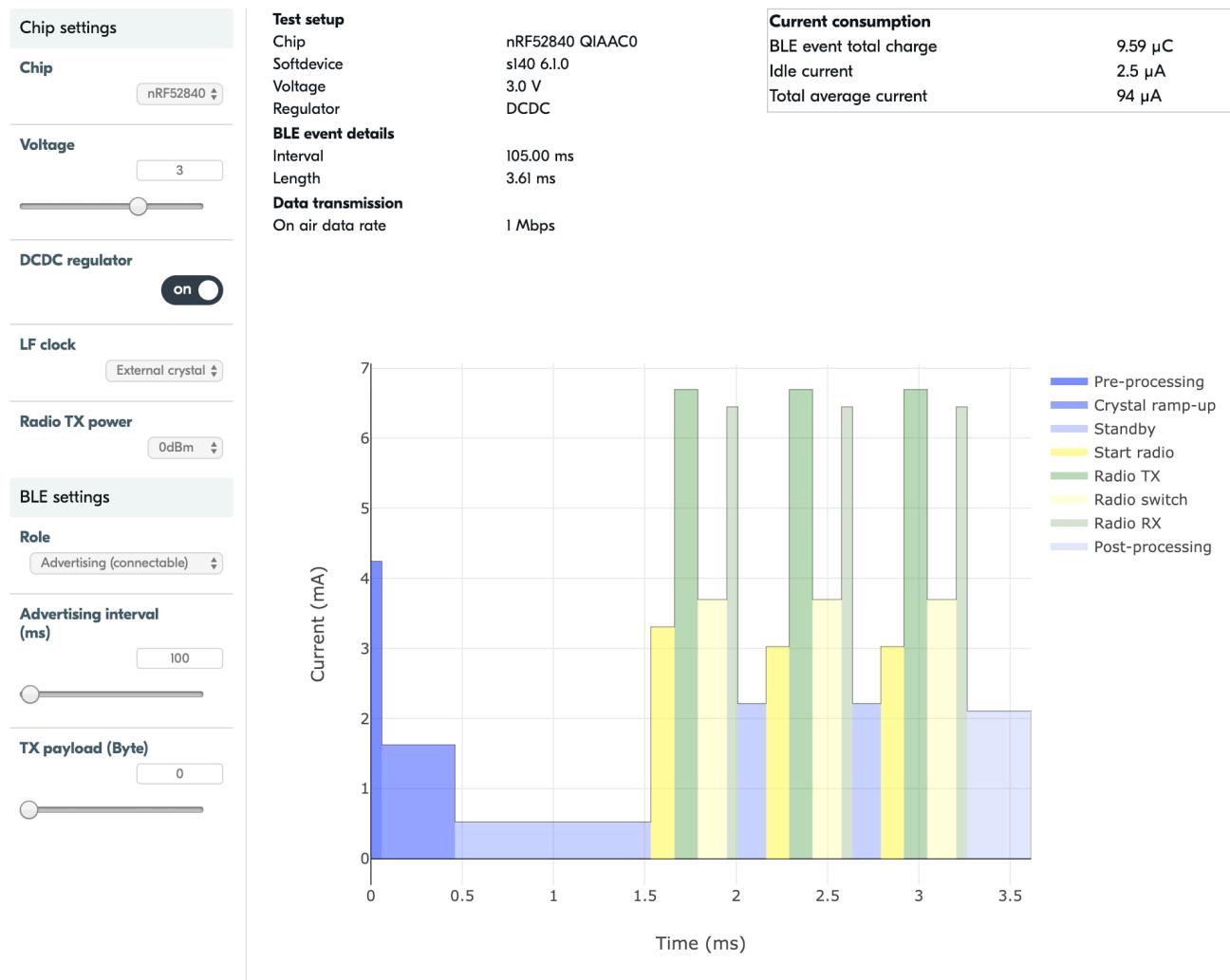


Figure 77: Nordic Online Power Profiler Tool

Exercise

For this example, we'll be running our **Remote Control** application (*included within the GitHub repository accompanying this book*). We'll take a look at the following parameters and see how they affect power consumption:

- **Advertising Interval** (default = 187.5 ms)
- **Connection Interval** (default = 15 ms)
- **Battery level measurement interval** (default = 5 seconds)

Some important notes about the test cases:

- When we modify one of these parameters, we will keep the others at their default values.
- We will run 3 test cases for each parameter (*variable*).
- For each test case, we will record the power measurements for a period of **20 seconds**.
- We will compare the Average Current Consumption values and Peak Current Consumption values for each of the test cases.

Advertising Interval

Default value: Advertising Interval = 187.5 ms

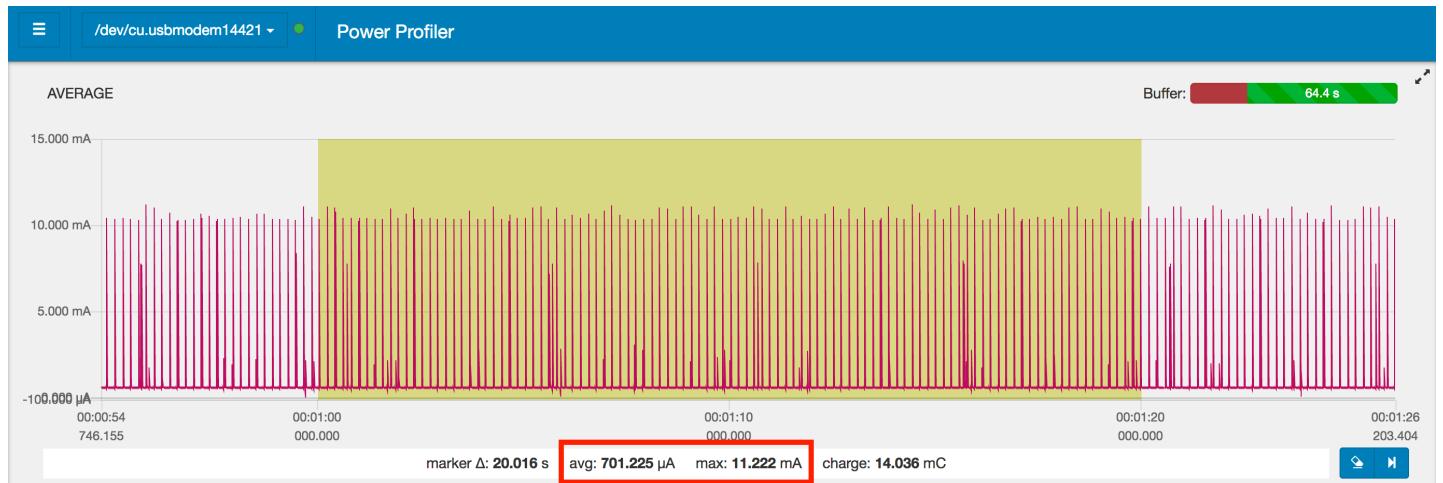


Figure 78: Advertising Interval = 187.5 ms

Test case #1: Advertising Interval = 500 ms

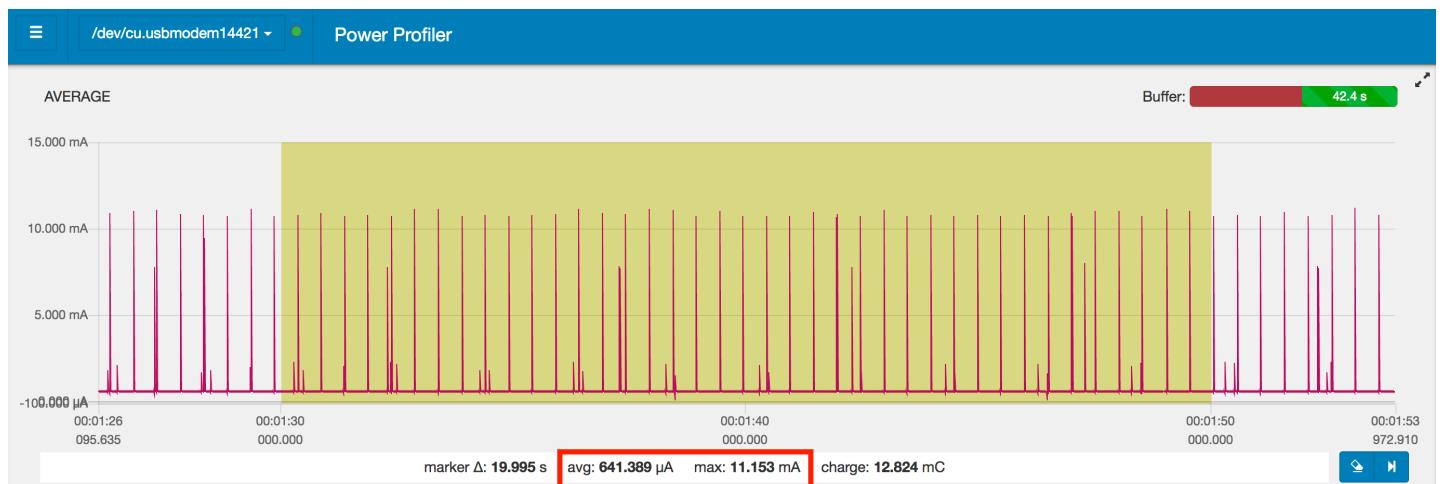
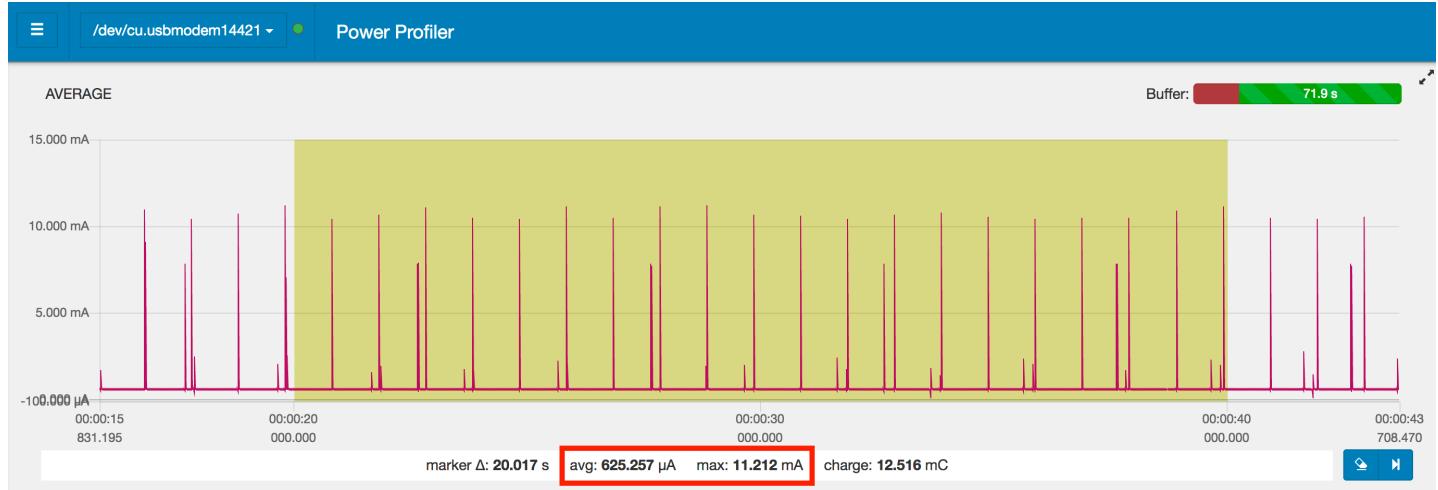
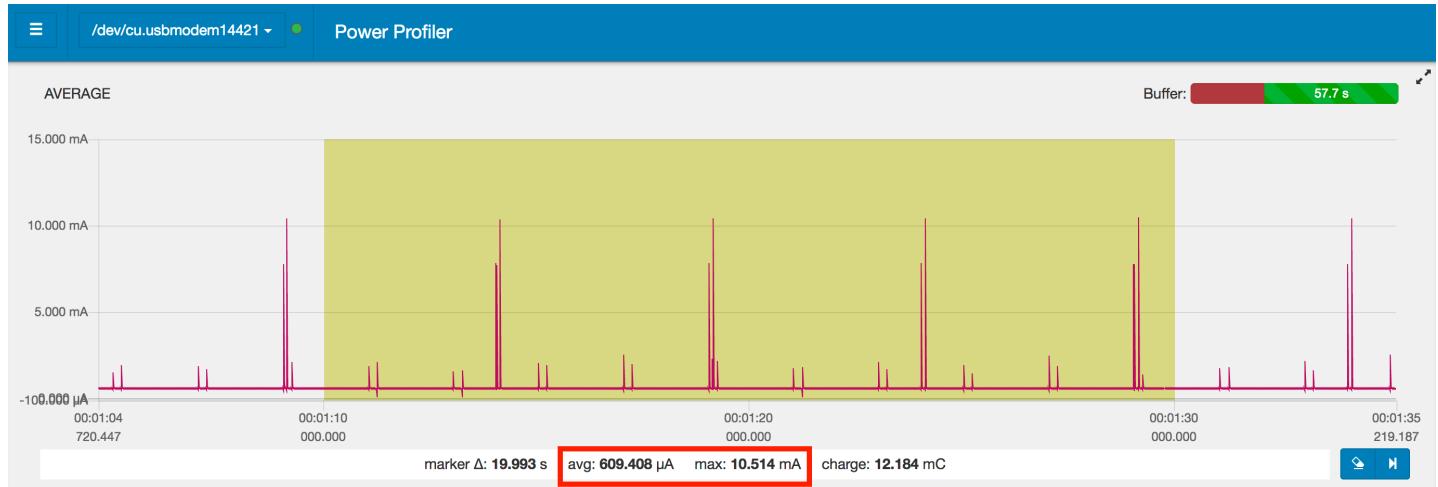


Figure 79: Advertising Interval = 500 ms

Test case #2: Advertising Interval = 1,000 ms**Figure 80: Advertising Interval = 1,000 ms****Test case #3: Advertising Interval = 5,000 ms****Figure 81: Advertising Interval = 5,000 ms****Observations:**

- The Peak Current values are very similar for all test cases.
- As expected, the Average Current Consumption for the analyzed period is highest when using a shorter Advertising Interval.
- Increasing the Advertising Interval beyond a certain point does not decrease power consumption by the same ratio — this is because the duration of high current consumption (when the radio is Advertising) is very short compared to the duration that the radio is off. So, changes to the Advertising Interval are more effective on the

faster end of the time scale.

Higher Resolution Current Measurement Using Triggers

To better understand the previous observation, we can use a very useful feature of the Power Profiler Kit application. This is called the "Trigger" feature and allows analysis of current consumption at a much higher resolution. A trigger simply tells the PPK to capture the current measurement for a much shorter time period **when a certain current consumption level is exceeded**.

In the Power Profiler Kit application, you can set the Trigger value as seen here:

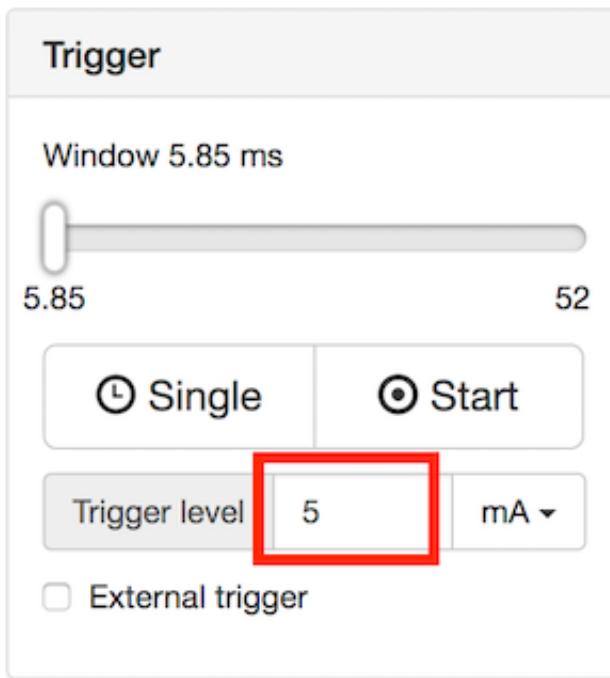


Figure 82: Trigger Value

Once you've set the Trigger level, you can then do a continuous capture or a single capture. In our case, we will be using the **Single** trigger feature:

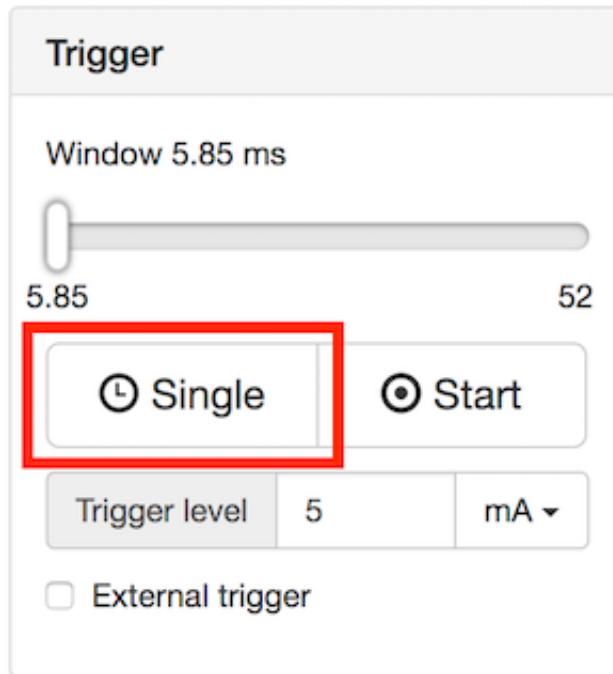


Figure 83: Single Trigger Value

Let's look at a single capture for a trigger value of **5 mA**:

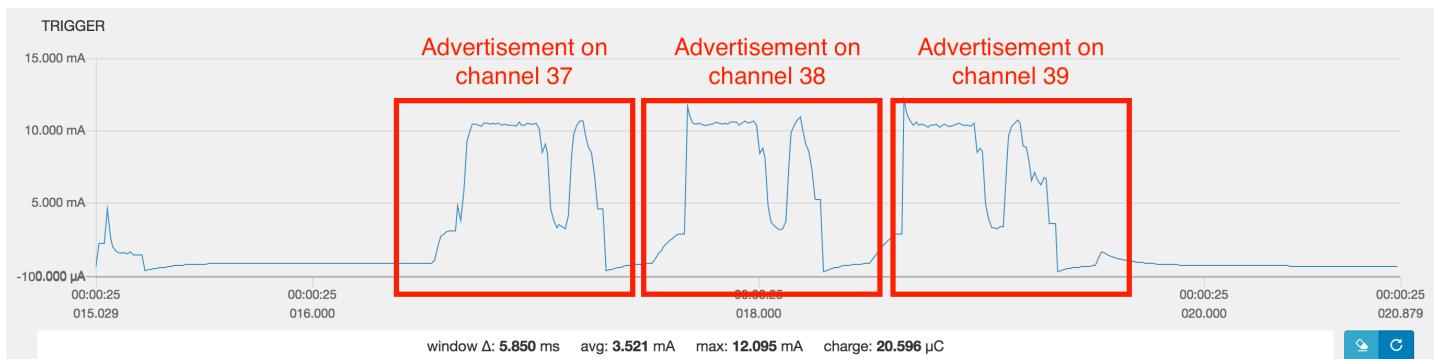


Figure 84: Single Trigger value of 5 ms

In the capture, you can see the different Advertising Packets that are sent on each of the three Primary Advertising Channels (37, 38, and 39). Now, let's look at the period of time it took to send out the three Advertising Packets:

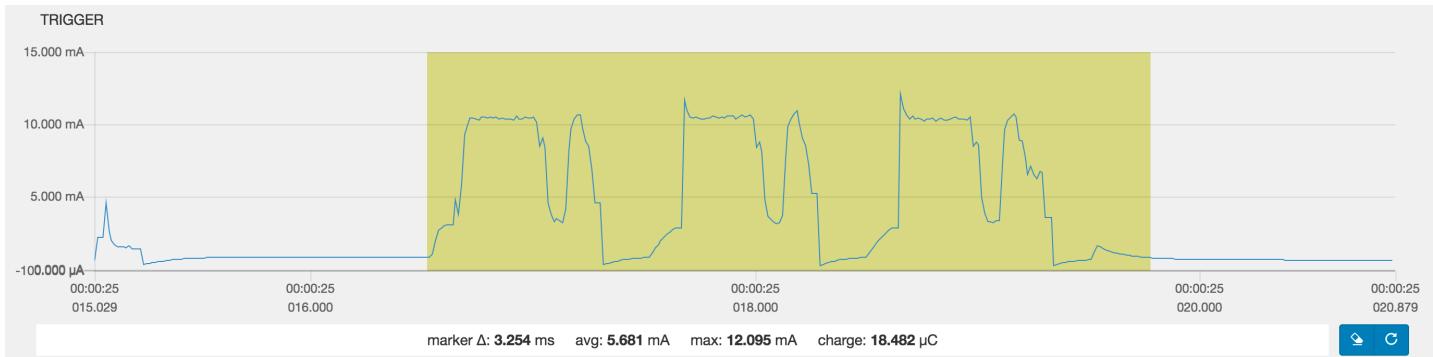


Figure 85: Advertising Interval measurement

We notice that sending all three Advertising Packets took about 3.2 ms. So, the higher the Advertising Interval, the lower the radio-on time (during transmission of advertising packets) compared to the radio-off time. This leads to the Average Current Consumption measurement being much higher when the Advertising Interval is at a low value.

Connection Interval

For Connection Interval test cases, we will connect to the Remote Control Peripheral from a mobile phone (using nRF Connect), and then wait for a few seconds after the connection to get past the Services and Characteristics discovery phase. *For consistency in measurements, we will not be performing any actions from the mobile phone during the connection.*

Default value: Connection Interval = 15 ms

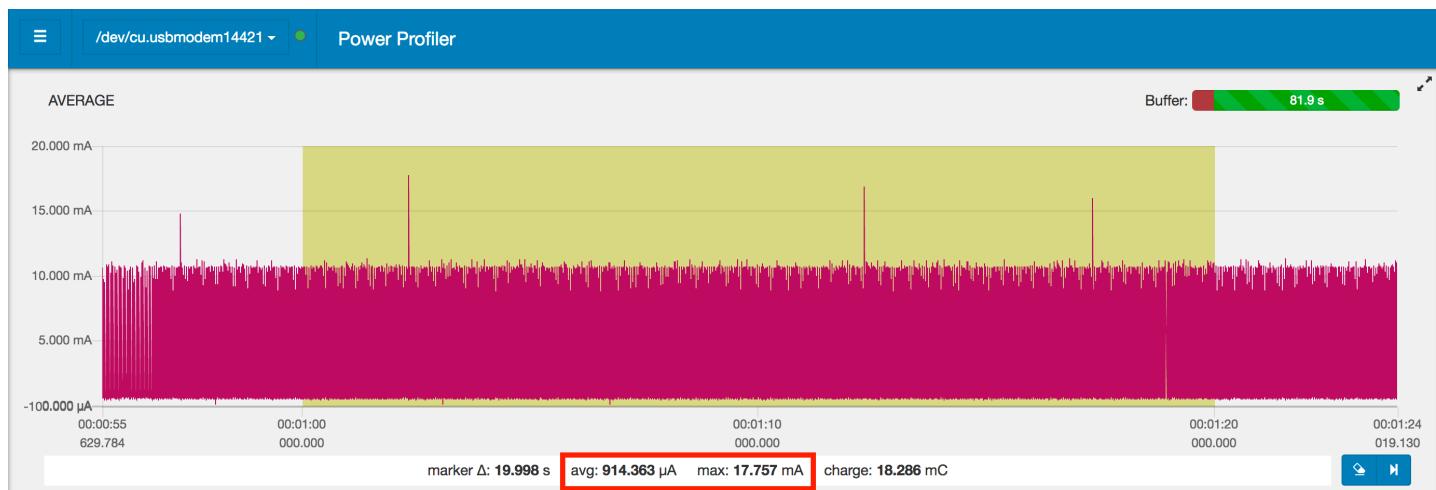
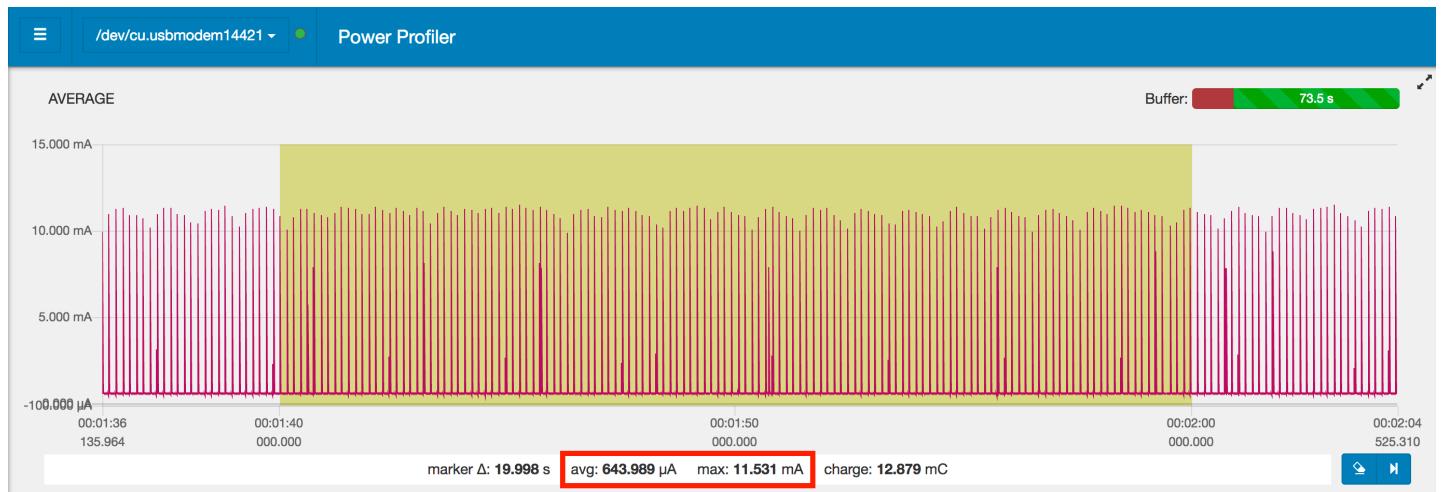
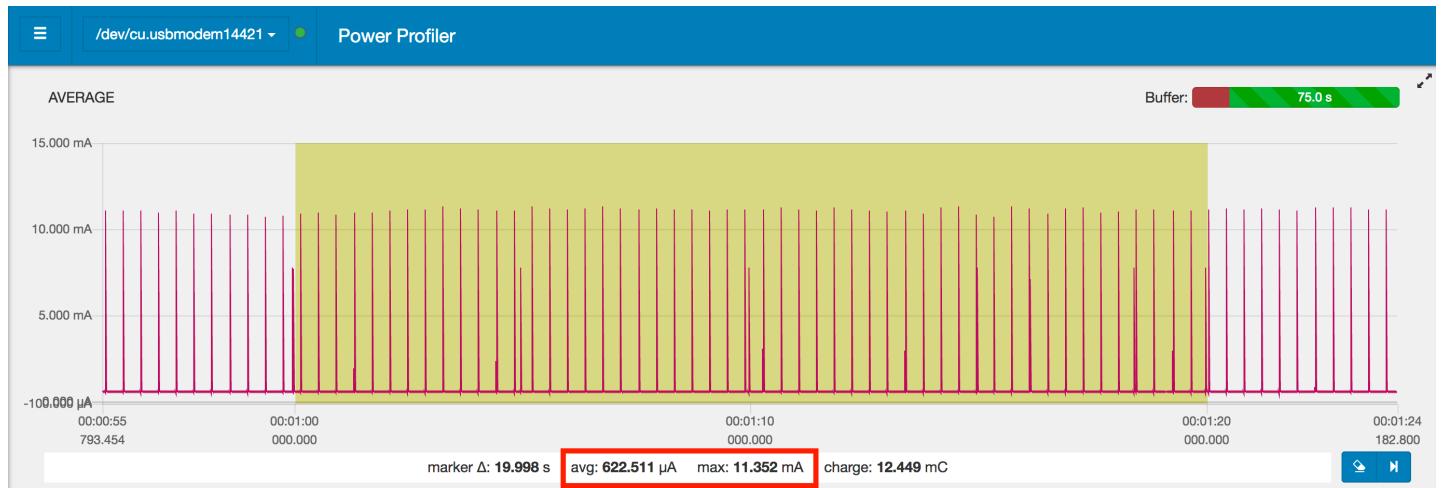


Figure 86: Connection Interval = 15 ms

Test case #1: Connection Interval = 150 ms

**Figure 87:** Connection Interval = 150 ms**Test case #2: Connection Interval = 375 ms****Figure 88:** Connection Interval = 375 ms**Test case #3: Connection Interval = 750 ms**

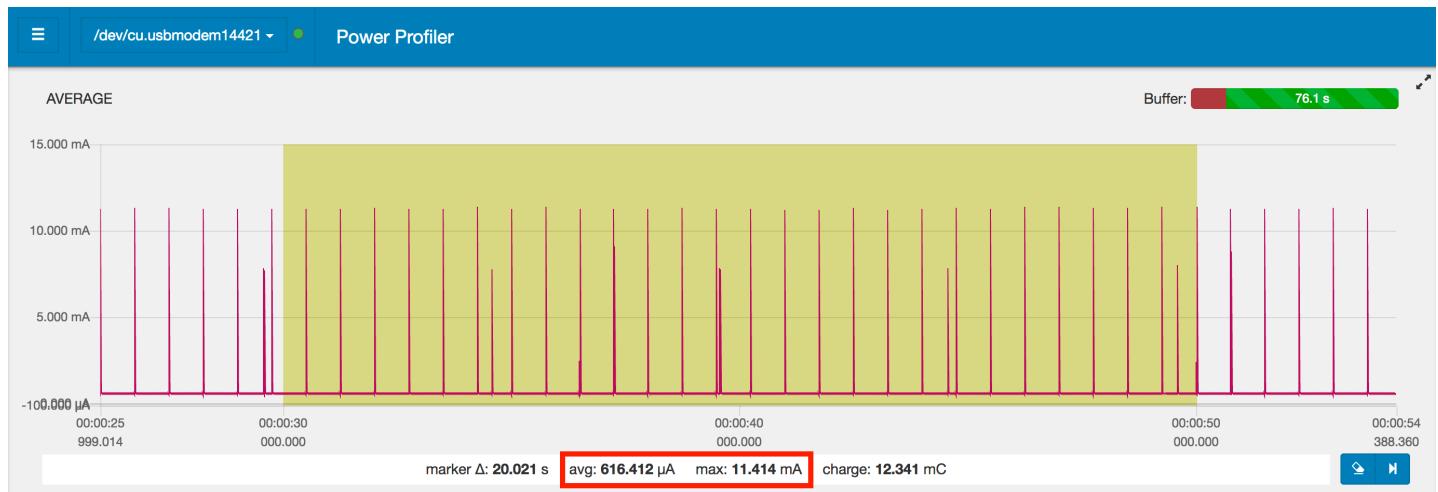


Figure 89: Connection Interval = 750 ms

Observations:

- Similar to the analysis for the advertising interval, the same applies to the Connection Interval.
- In the case of Connection Intervals, we have another way to optimize the current consumption. We can have a short connection interval and still be able to consume significantly reduced current levels. The way this can be achieved is by utilizing the **Slave Latency** parameter.

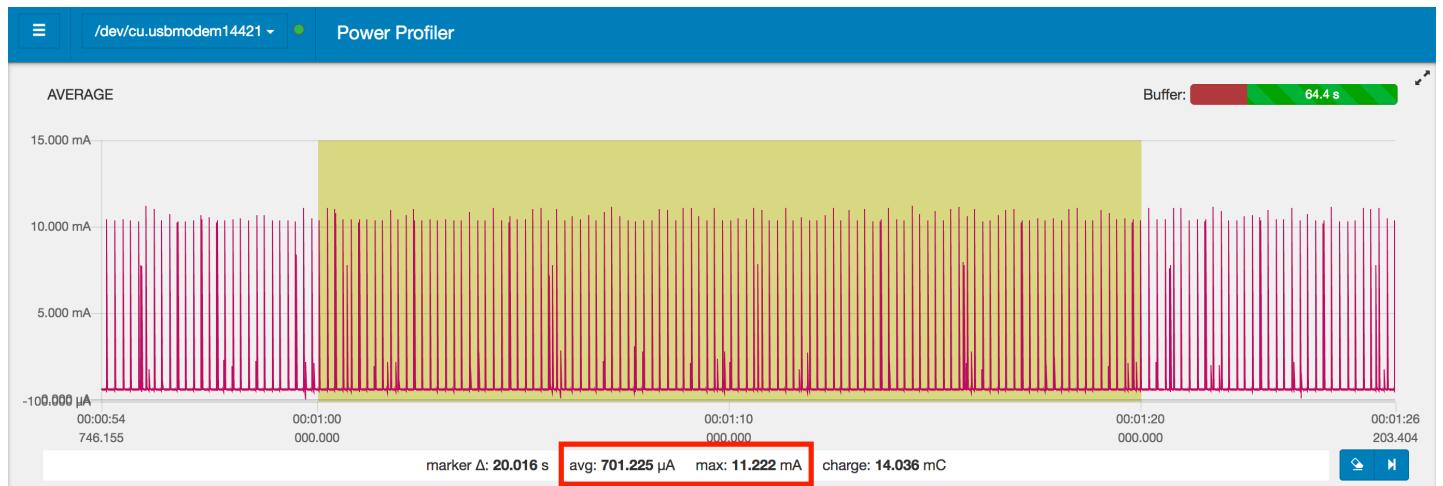
Recall that Slave Latency is a number – an integer, that specifies how many Connection Intervals can be ignored by the Peripheral without affecting the connection.

We can simply increase the Slave Latency and still be able to achieve a similar level of current consumption as if we increased the Connection Interval. This is especially true when the Peripheral device is mostly idle and has no data to send back to the central — since it can skip Connection Intervals without waking up the radio.

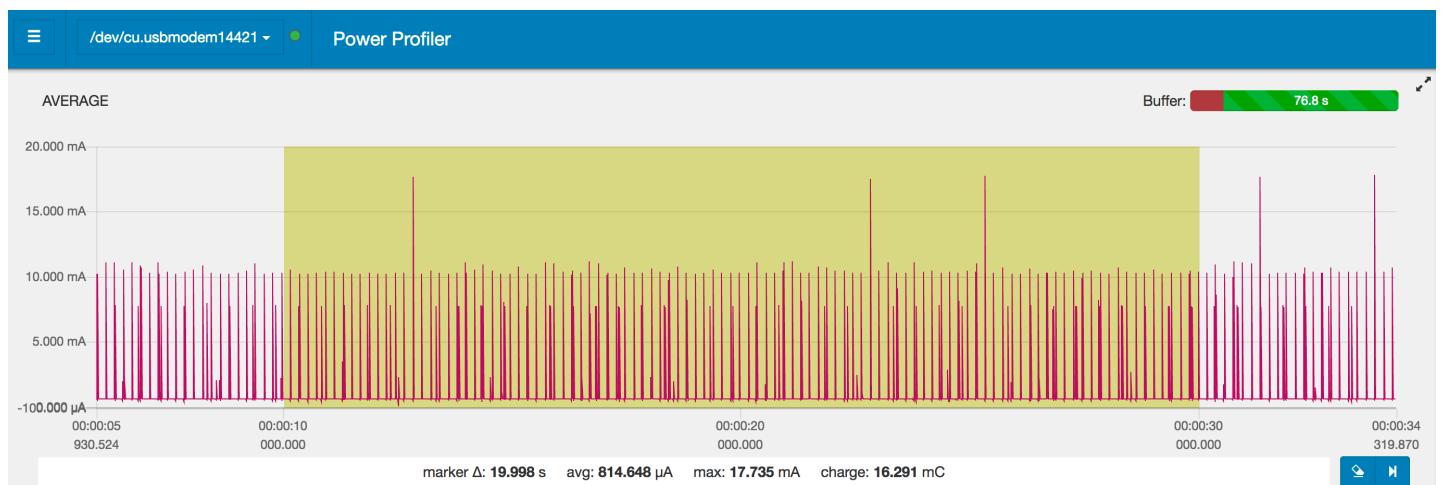
Battery Level Measurement Interval

For simplicity, for this measurement we will be running the test cases while Advertising (no Connections), but adjusting the frequency of the battery level readings and seeing how they compare.

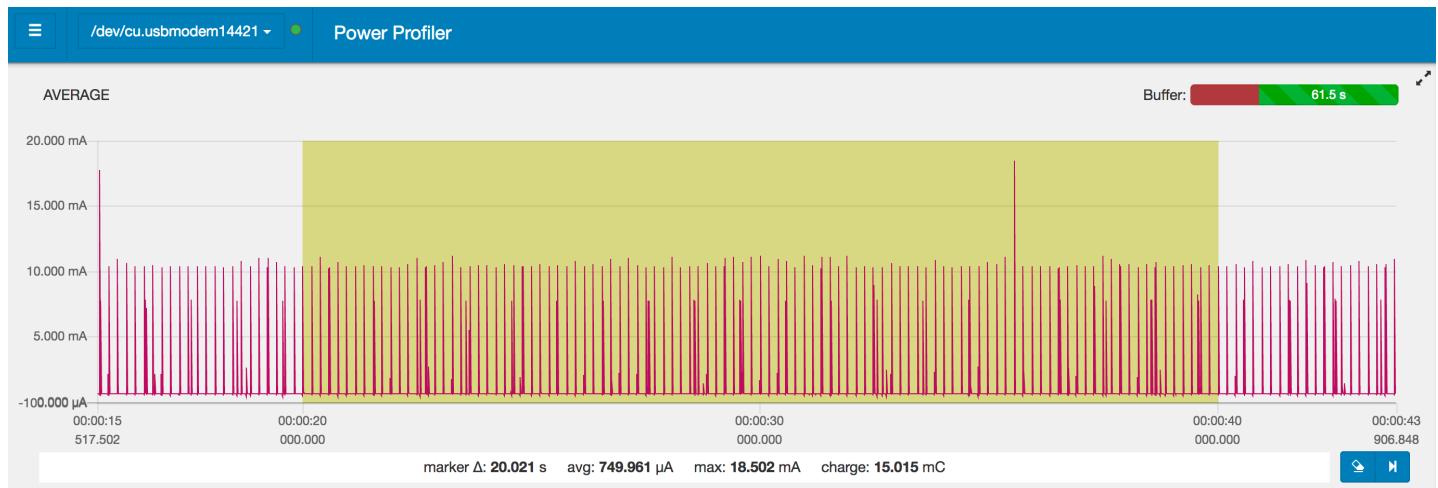
Default value: Connection Interval = 5 seconds — same as the default Advertising case

*Figure 90: Connection Interval = 5 seconds*

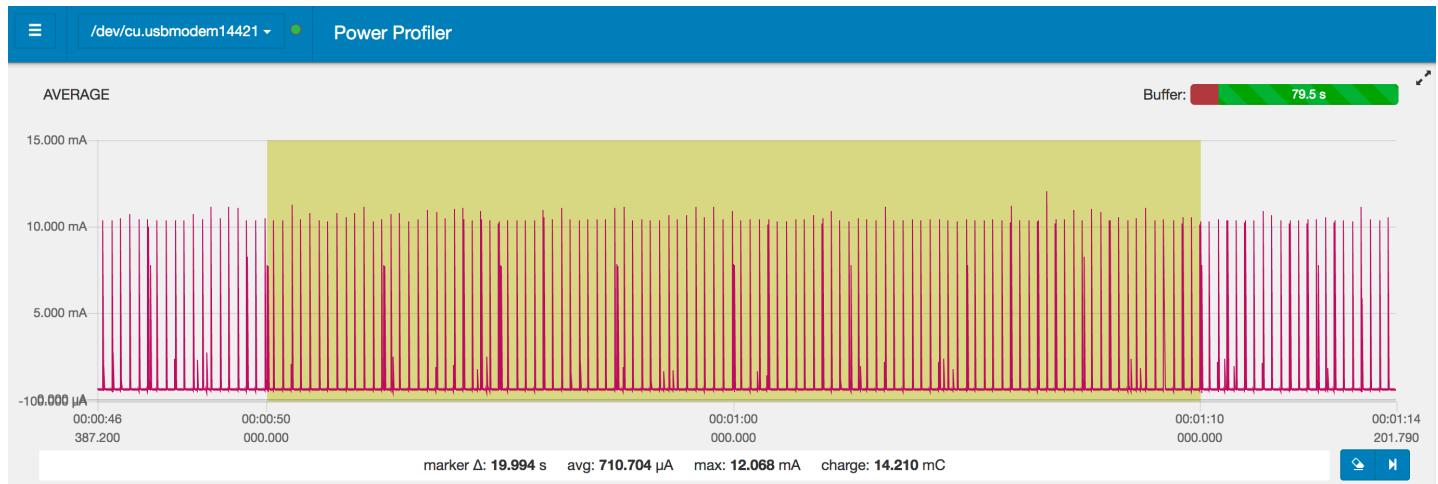
Test case #1: Measurement interval = 0.5 seconds

*Figure 91: Measurement interval = 0.5 seconds*

Test case #2: Measurement interval = 1 seconds

*Figure 92: Measurement interval = 1 seconds*

Test case #3: measurement interval = 2.5 seconds

*Figure 93: Connection Interval = 5 seconds*

Observation:

We notice here that increasing the interval at which the battery level is measured does not impact the Average Current Consumption as much as reducing the BLE parameters. This is due to the fact that Advertising was much more frequent than the battery level measurements.

Security

Security has become one of the most voiced concerns about IoT systems. With all the headline news that mention hacks and vulnerabilities discovered in many IoT products, it has become one of the major concerns for manufacturers and developers of IoT devices.

In this chapter, we will:

- Go over the different security concerns.
- Take a look at the security measures that BLE provides.
- Cover Privacy concerns.

Security Concerns

Some of the most common security concerns with any system include:

- **Authentication:** Authentication is proof that the other side is who they claim they are. So if you're connecting to a BLE device, you want to be sure that you are actually connecting to the device of interest — and not some other malicious device that's pretending to be that device.
- **Integrity:** Integrity ensures us that the data received is free from corruption and tampering by unauthorized devices.
- **Confidentiality:** Confidentiality is concerned with making sure the data is not readable by unauthorized users or devices.
- **Privacy:** Privacy is concerned with how private the communication is, and whether a third party is able to track our device — especially by its Bluetooth address.

These are some general concerns related to security that apply to any system. The importance of each one of these concerns depends on the application and use case of the product.

Types of Attacks

Based on the above mentioned concerns, there are different types of attacks that a malicious device or person may implement. Some of these include:

- **Passive Eavesdropping:** This describes when a malicious device listens in on the communication between two devices, and is able to understand the data — usually by gaining access to the encryption key in the case the data is encrypted.
- **Active Eavesdropping:** This is also known as a **Man-In-The-Middle (MITM)** attack. In this attack, the malicious device impersonates both devices (the peripheral and the central). It could then intercept the communication between them, route it so they do not realize that the attack is happening, and possibly even injecting data into the packets.

- **Privacy and Identity Tracking:** In this attack, devices and users are tracked by the Bluetooth address — possibly revealing their location and correlating it with their behavior.

Security in BLE

Security in BLE is handled by the **security manager (SM)** layer of the architecture. It is shown in the following diagram:

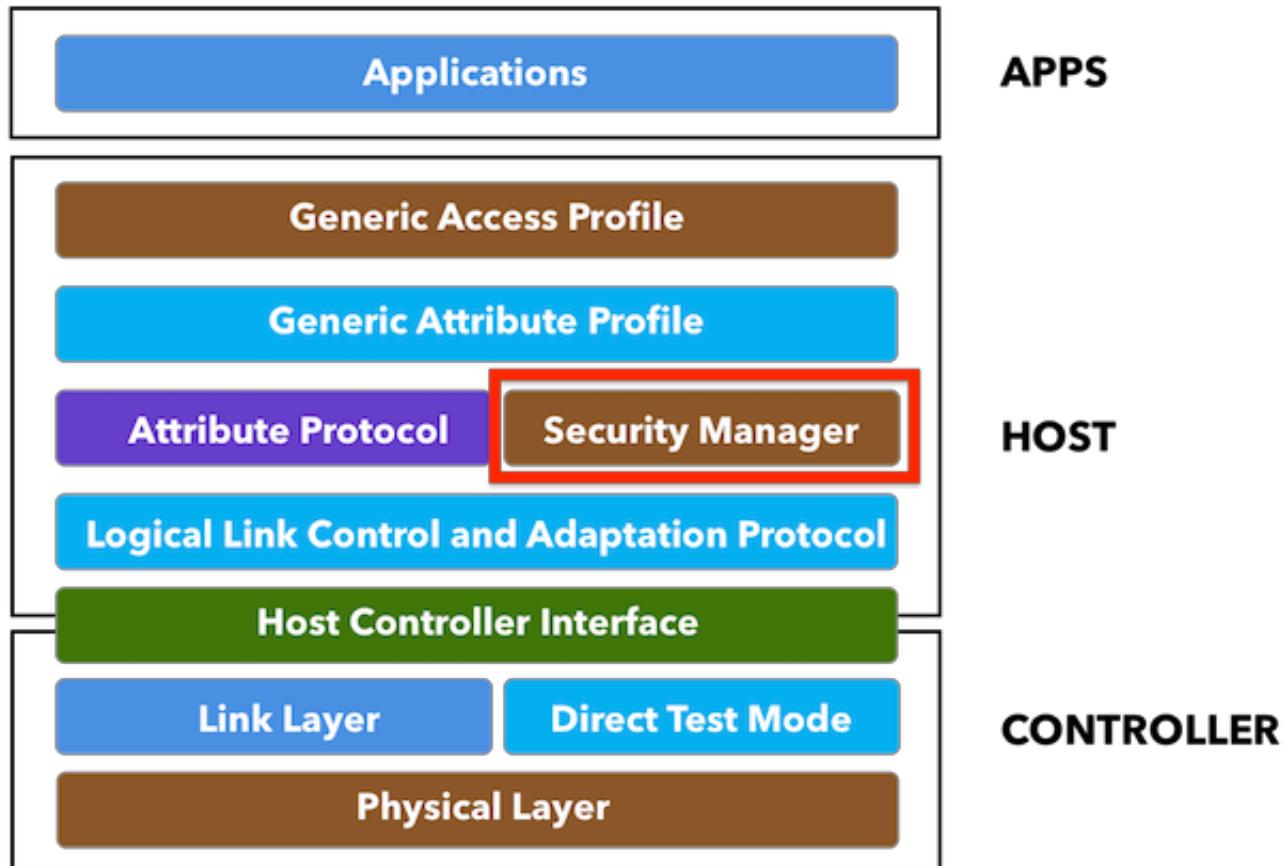


Figure 94: Security manager (SM) in the architecture of BLE

The security manager defines the protocols and algorithms for generating and exchanging keys between two devices. It involves five security features:

- **Pairing:** the process of creating shared **secret keys** between two devices.
- **Bonding:** the process of creating and storing shared **secret keys** on each side (central and peripheral) for use in subsequent connections between the devices.
- **Authentication:** the process of verifying that the two devices share the same secret keys.
- **Encryption:** the process of encrypting the data exchanged between the devices. Encryption in BLE uses the 128-bit AES Encryption standard, which is a **symmetric-key** algorithm (meaning that the same key is used to encrypt

and decrypt the data on both sides).

- **Message Integrity:** the process of **signing** the data, and **verifying** the signature at the other end. This goes beyond the simple integrity check of a calculated CRC.

The Bluetooth specification has evolved over time to provide stronger security measures. This is especially true for BLE, which introduced the concept of **LE Secure Connections (LESC)** in version 4.2. LESC utilizes the [Elliptic-curve Diffie-Hellman \(ECDH\)](#) protocol during the pairing process (covered later in this chapter), which makes the communication much more secure compared to the methods used in earlier versions of Bluetooth.

Bluetooth 4.2 also introduced the term **legacy connections**, which collectively refers to the pairing methods defined by the earlier specification versions. It's important to note, though, that legacy connections are still supported in Bluetooth 4.2 and later. We'll cover the differences between these methods in the upcoming sections.

The Security Manager addresses the different security concerns as follows:

- **Confidentiality** via encryption.
- **Authentication** via pairing & bonding.
- **Privacy** via resolvable private addresses.
- **Integrity** via digital signatures.

In BLE, the master device is the **initiator** of security procedures. The slave (**responder**) may request the start of a security procedure by sending a security request message to the master, but it is up to the master to then send the packet that officially starts the security process.

To better understand how security works in BLE, we need to understand two important concepts: **pairing** and **bonding**. But first, let's review a sequence diagram showing the security process:

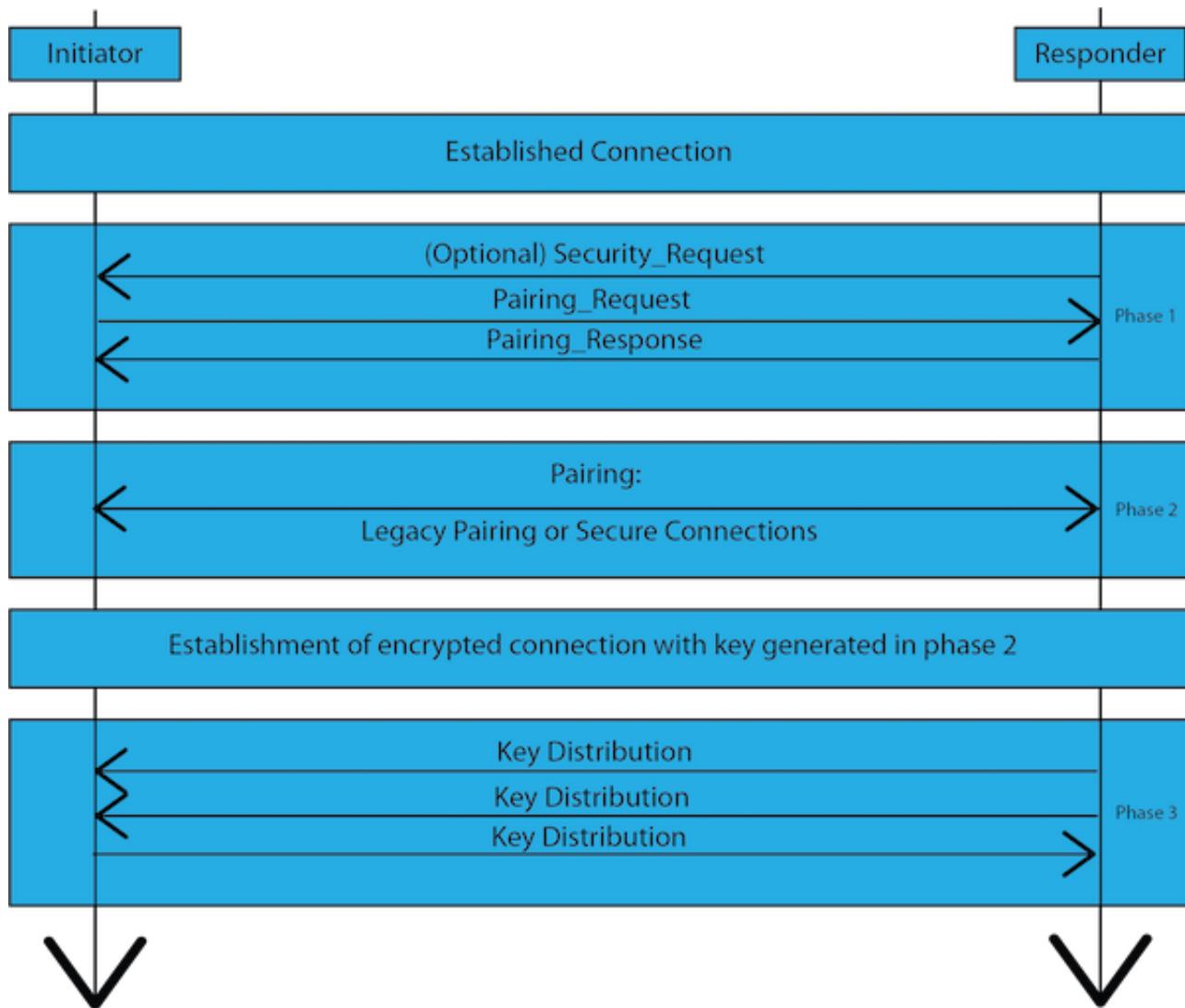


Figure 95: The different phases of the security process in BLE

Pairing is the combination of Phases 1 and 2. **Bonding** is represented by Phase 3 of the process. One important thing to note is that Phase 2 is the only phase that differs between LE Legacy Connections and LE Secure Connections.

Pairing and Bonding

Pairing is a temporary security measure that does not persist across connections. It has to be initiated and completed each time the two devices reconnect and would like to encrypt the connection between them. In order to extend the encryption across subsequent connections, bonding must occur between the two devices.

Let's go over the different phases in more detail:

Phase One

In this phase, the slave may request the start of the pairing process. The master initiates the pairing process by sending a **pairing request** message to the slave, which then responds with a **pairing response** message. The pairing request and pairing response messages represent an exchange of the features supported by each device, as well as the security requirements for each device. Each of these messages include the following:

- **Input Output (IO) capabilities:** display support, keyboard support, yes/no input support.
- **Out-Of-Band (OOB) method support.**
- **Authentication requirements:** includes MITM protection requirement, bonding requirement, secure connections support.
- **Maximum encryption key size** that the device supports.
- The different **security keys** each device is requesting to use.

The information exchanged between the two devices in this phase determines the pairing method used. Here's a table showing the different combinations of the exchanged IO capabilities (on the two pairing devices) and the resulting pairing method chosen:

		Initiator							Initiator				
Responder	DisplayOnly	Display YesNo	Keyboard Only	NoInput NoOutput	Keyboard Display	Responder	DisplayOnly	Display YesNo	Keyboard Only	NoInput NoOutput	Keyboard Display		
Display Only	Just Works Unauthenticated	Just Works Unauthenticated	Passkey Entry: responder displays, initiator inputs Authenticated	Just Works Unauthenticated	Passkey Entry: responder displays, initiator inputs Authenticated	Keyboard Only	Passkey Entry: initiator displays, responder inputs Authenticated	Passkey Entry: initiator displays, responder inputs Authenticated	Passkey Entry: initiator and responder inputs Authenticated	Just Works Unauthenticated	Passkey Entry: initiator displays, responder inputs Authenticated		
	Just Works Unauthenticated	Just Works (For LE Legacy Pairing) Unauthenticated	Passkey Entry: responder displays, initiator inputs Authenticated	Just Works Unauthenticated	Passkey Entry (For LE Legacy Pairing): responder displays, initiator inputs Authenticated		Just Works Unauthenticated	Just Works Unauthenticated	Just Works Unauthenticated	Just Works Unauthenticated	Just Works Unauthenticated		
Display YesNo	Just Works Unauthenticated	Numeric Comparison (For LE Secure Connections) Authenticated	Passkey Entry: responder displays, initiator inputs Authenticated	Just Works Unauthenticated	Numeric Comparison (For LE Secure Connections) Authenticated	NoInput NoOutput	Passkey Entry (For LE Legacy Pairing): initiator displays, responder inputs Authenticated	Passkey Entry: responder displays, initiator inputs Authenticated	Passkey Entry: responder displays, initiator inputs Authenticated	Just Works Unauthenticated	Passkey Entry (For LE Legacy Pairing): initiator displays, responder inputs Authenticated		
	Just Works Unauthenticated						Passkey Entry: initiator displays, responder inputs Authenticated	Numeric Comparison (For LE Secure Connections) Authenticated	Passkey Entry: responder displays, initiator inputs Authenticated	Just Works Unauthenticated	Numeric Comparison (For LE Secure Connections) Authenticated		

Figure 96: IO Capabilities Lookup Table
(Source: *Bluetooth 5 specification document*)

Phase Two

As mentioned previously, **phase two** differs based on which method is used: **LE secure connections** or **LE legacy connections**.

Let's explain how this phase differs between the two methods:

- **Legacy Connections:**

In **legacy connections**, there are two keys used: the **temporary key (TK)** and the **short term key (STK)**. The TK is used along with other values exchanged between the two devices to generate the STK.

- **Secure Connections:**

In **secure connections**, the pairing method does not involve exchanging keys over the air between the two devices. Rather, the devices utilize the ECDH protocol to each generate a **public/private key** pair. The devices then exchange the public keys only, and from that generate a shared secret key called the **long term key (LTK)**.

The advantage of using ECDH is that it prevents eavesdroppers from figuring out the shared secret key — even if they capture both public keys. To learn more about ECDH and how it works, refer to its Wikipedia page [here](#). I've also found that [this video](#) explains it very well.

Phase Three

Phase three represents the **bonding** process. This is an optional phase that's utilized to avoid the need to re-pair on every connection to enable a secure communication channel. The result of bonding is that each device stores a set of keys that can be used in each subsequent connection and allows the devices to skip the pairing phase. These keys are exchanged between the two devices over a link that's encrypted using the keys resulting from phase two.

Pairing Methods

Legacy Connections and Secure Connections each have different Pairing Methods. Some of the Methods share the same name, but the process and the data exchanged differs among them. The Pairing Method that gets used is determined based on the features exchanged between the two devices in Phase One.

LE Legacy Connections (All Bluetooth Versions)

As we mentioned earlier, in **legacy connections** a short term key (STK) is generated from the temporary key (TK) and two randomly generated values.

- **Just Works:**

In this method, TK is set to **0**. For obvious reasons, this method is the least secure of all methods (amongst all Bluetooth versions).

- **Out of Band (OOB):**

In this method, the TK is exchanged between the two devices over a technology other than BLE — **near field communication (NFC)** being the main one. This method can make the pairing process much more secure, especially if the non-BLE technology used provides stronger security. This is the most secure method of the legacy pairing methods.

- **Passkey:**

In this method, the TK is a six-digit number that is transferred between the devices by the end-user. For example,

it may be entered manually into one of the devices. The challenge here is that the devices need to have some minimal IO capabilities such as a display and keyboard entry methods, so its use may be limited.

LE Secure Connections (Bluetooth Version 4.2 and Later)

- **Just Works:**

In this method, the **public keys** for each device along with other generated values get exchanged between the two devices over BLE.

- **Out of Band (OOB):**

In this method, the values are exchanged over a medium other than BLE. If the used medium is secure, then this makes the connection more secure.

- **Passkey:**

In this method, an identical six-digit number is used. The six-digit number could either be entered by the user into each device, or one of the devices will generate it for the user to manually enter it into the other device.

- **Numeric Comparison:**

This method works the same as the **just works** method described above but adds an extra step at the end. This extra step allows protection from MITM attacks. This is the most secure pairing method of all methods.

Note: Security in BLE is a vast subject that cannot be fully covered in this book. You can learn more about the different pairing methods and how each of them works in more detail by reviewing message sequence diagrams provided in the Bluetooth 5 specification document (Vol 3, Part H, Appendix C. page 2364).

Privacy

Privacy is another major concern for users and it has to be taken seriously. Each Bluetooth device has an address, and if careful measures are not put in place, this address can be used to track users. Fortunately, BLE provides a privacy feature to safeguard against such vulnerabilities.

A device can use a frequently changing private address for its Bluetooth address that only trusted devices can resolve. A trusted device in this case is a bonded device. The random private address is generated using a key called the **identity resolving key (IRK)**, which is exchanged between two bonded devices during phase three. This way, the peer device has access to the IRK and can resolve the random address.

An Overview of the Different Security Keys

There are a number of keys and variables used during the different security procedures. Let's go over them one by one.

- **Temporary Key (TK):**

Generation of the temporary key (TK) depends on the pairing method chosen. The TK gets generated each time the pairing process occurs. The TK is used in legacy connections only.

- **Short Term Key (STK):**

This key is generated from the TK exchanged between the devices. The STK gets generated each time the pairing process occurs and is used to encrypt the data throughout the current connection. The STK is used in legacy connections only.

- **Long Term Key (LTK):**

This key gets generated and stored during phase three of the security process in legacy connections and during phase two in LE secure connections. It gets stored on each of the two devices that are bonded, and used in subsequent connections between the two devices.

- **Encrypted Diversifier (EDIV) and Random Number (Rand):**

These two values are used to create and identify the LTK. They also get stored during the bonding process.

- **Connection Signature Resolving Key (CSRK):**

Used to sign data and verify the signature attached to the data at the other end. This key is stored on each of the two bonded devices.

- **Identity Resolving Key (IRK):**

Used to resolve random private addresses. This key is unique per device, so the master's IRK will get stored on the slave side, and the slave's IRK will be stored on the master side.

Security Modes and Levels

There are two security modes in BLE: **Security mode 1** and **security mode 2**. Security mode 1 is concerned with encryption whereas security mode 2 is concerned with data signing.

Here are the different levels for each mode:

Security Mode 1

- **Level 1:** No security (no authentication and no encryption)
- **Level 2:** Unauthenticated pairing with encryption
- **Level 3:** Authenticated pairing with encryption
- **Level 4:** Authenticated LE secure connections pairing with encryption

Security Mode 2

- **Level 1:** Unauthenticated pairing with data signing
- **Level 2:** Authenticated pairing with data signing

A link is considered **authenticated** or **unauthenticated** based on the pairing method used. Looking back at the table listed under the section on pairing phase one we can see that for each entry, it lists whether the method is considered authenticated or unauthenticated.

A link between two devices operates in one security mode only but can operate at different levels within that mode (different characteristics may require different levels of security). For example:

- One characteristic may require Level 1 (no security) for read access.
- The same characteristic may require level 3 for write access.
- Another characteristic may require level 4 for both read and write access.

What Triggers Security on a Connection?

There are a number of operations that trigger security on a connection, some of which are:

- The master sends a pairing request, which results in the slave sending a pairing response.
- The slave sends a security request, to which the master responds with a pairing request. This results in the slave sending a pairing response.
- A client accesses a characteristic on the server which requires a specific security level, triggering a pairing — and possibly bonding the two devices.

For example, if the notification permissions on a specific characteristic are configured to require security, then when a unsecured client attempts to enable notifications, an **insufficient authentication response** message will be sent from the server indicating that a certain level and mode of security are required for the operation to be completed.

- Two previously bonded devices connecting to each other — which triggers encryption using the previously distributed Keys.

Note: iOS does not allow requiring special permissions, such as pairing, authentication, or encryption to discover services and characteristics. Instead, it requires that the server only restrict access or permissions to characteristics in order to trigger pairing. This is according to the iOS Bluetooth guidelines document that can be downloaded [here](#).

Exercise: Securing Your BLE Application

For our example, we'll be expanding on our previous "Hello World" example to enable Security. The nRF52840 board will be running the modified **secure** "Hello World" example. On the other side, the Central will be run from nRF Connect on the desktop or on a mobile device.

The Security Method we will be using is the strongest level defined within the Bluetooth specification: **LE Secure Connections with MITM Protection (Numeric Comparison) — Mode 1 Level 4**.

The source code for the example is provided in the GitHub repository that accompanies the book (in the folder named `Hello World with Security`).

We'll cover this example in a **video tutorial** titled "**BONUS Video #2 - Hello World with Security (LESC+MITM)**" — included with the book. You can also watch it [here](#):

[BONUS Video #2: Hello World with Security \(LESC+MITM\)](#)

Note: This example has not been migrated to nRF5 SDK version 15.2.0. If you are looking for example code for LE Secure Connections with MITM Protection, refer to these three examples included in nRF5 SDK version 15.2.0:

- `examples/ble_peripheral/ble_app_hrs`
- `examples/ble_central/ble_app_hrs_c`
- `examples/ble_central_and_peripheral/experimental/ble_app_multirole_lesc`

Debugging and Testing BLE Applications

Testing your BLE application can be a challenging task, especially in the early stages of developing your application.

At this stage:

- You may not have a mobile app ready to interface with your BLE Peripheral.
- Perhaps the Peripheral hardware is not ready yet and you'd still like to start testing to prove that the system design is viable.
- You have both the mobile app and hardware available, but you want to test out new features that have not been implemented on either side yet.

There are a number of ways that can help you achieve this. In this chapter, we will go over a few of these methods and tools.

Using a Central Emulator Application

A Central Emulator app is an application that runs on a device (usually a mobile device) and can be controlled via a user interface to discover and connect to BLE peripherals. The first, and obvious, requirement is that the device running the application needs to support BLE.

Two of the most common Central Emulator Applications are:

- **Nordic Semiconductor nRF Connect**, which is available for mobile devices ([iOS](#) and [Android](#)) as well as desktop computers ([Windows](#), [Linux](#) and [macOS](#)). The desktop version of this application requires nRF hardware (such as one of the nRF development kits).
- **LightBlue**, which is available for mobile devices ([iOS](#) and [Android](#)) as well as desktop computers ([macOS only](#)).

These two Central Emulator Applications are also capable of emulating a BLE Peripheral, but the focus in this chapter is on testing the Peripheral side from a Central device/Application.

We'll be focusing on how to use nRF Connect (mobile app version) in this chapter.

nRF Connect (Mobile)

First, go ahead and download the app to your smartphone. Once you've launched the application, you'll be presented with the following screen:

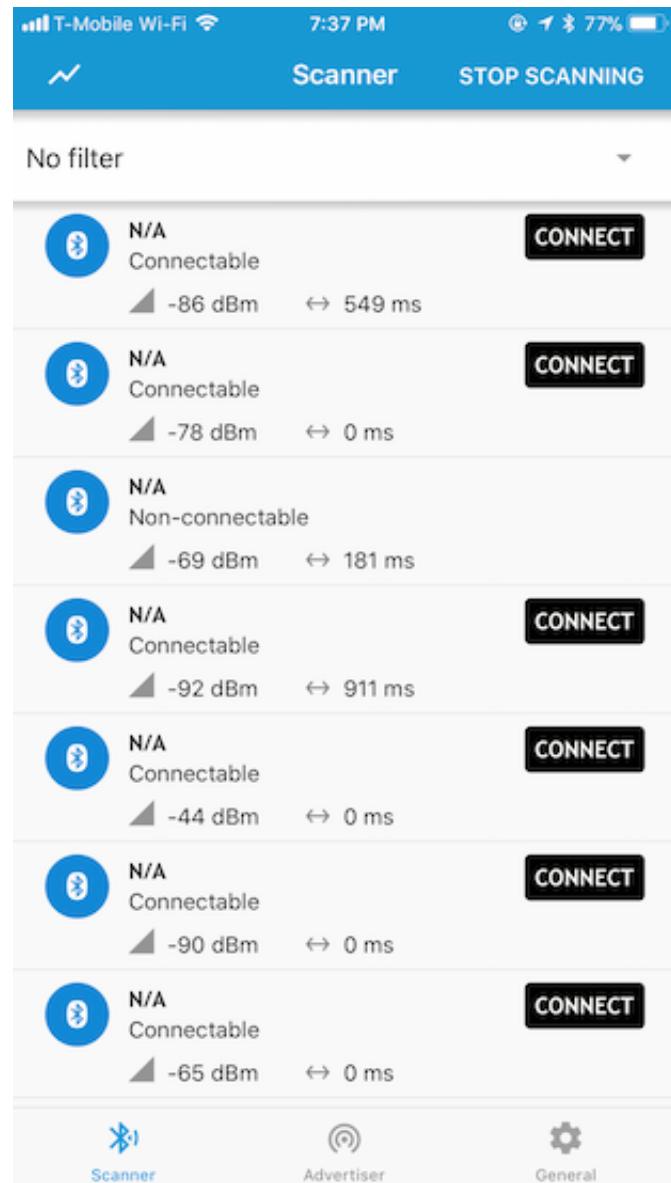


Figure 97: nRF Connect mobile application

For demo purposes, we'll be using the Thingy:52 to go through some testing scenarios and showing how to use the mobile app to interact with the Peripheral.

Advertisements and Scanning

The main screen in the application is used to discover Peripherals in the vicinity. If you're in an environment where many Peripheral devices may be present, you can use the **Filter** feature of the app.

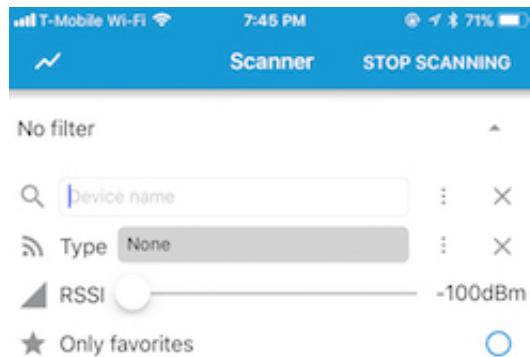


Figure 98: nRF Connect Filter Option

From this menu, you can filter by:

- **Device name:** allows you to enter text that's contained in a device's name

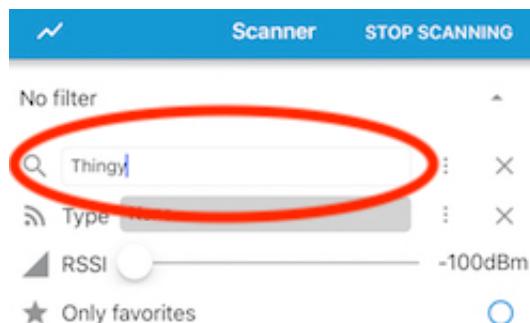


Figure 99: nRF Connect Device Name Filtering

- **Type:** the application includes options for filtering different types of devices such as: nRF Beacon, Eddystone, Physical Web, Nordic Thingy, and others.

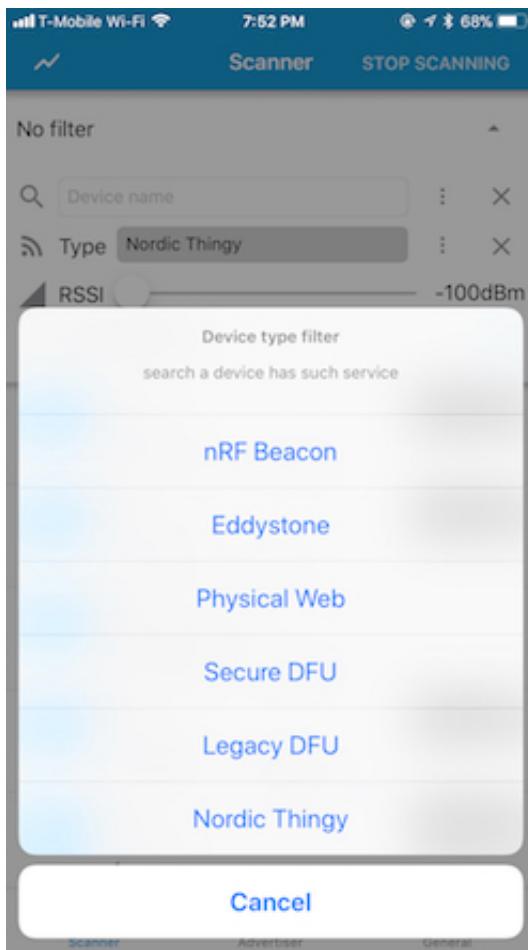


Figure 100: nRF Connect Device Type Filtering

- **RSSI:** with this option, you can filter devices by proximity to your smartphone._Tip: If the device is very close to the smartphone (within inches), setting the RSSI to -50 or so will generally be a safe bet to display the peripheral._

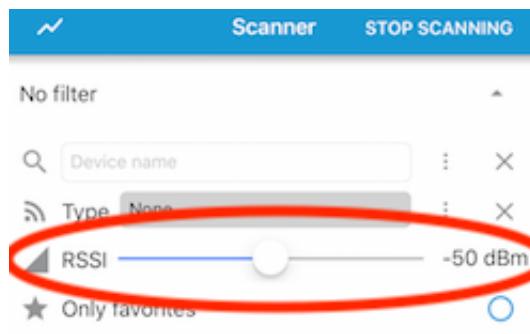


Figure 101: nRF Connect Filtering devices

- **Only Favorites:** you can “favorite” a Peripheral by tapping on the Bluetooth icon next to a device that shows up in the list (*a star will then be displayed under the icon*).

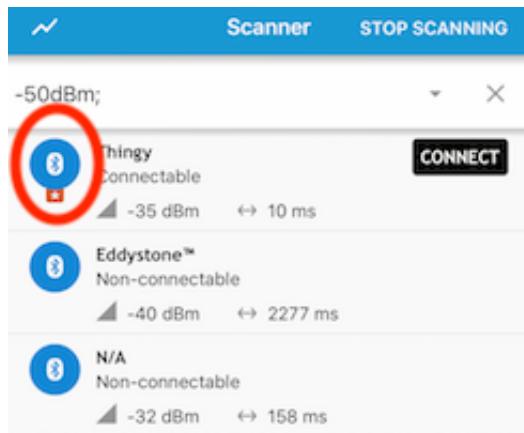


Figure 102: nRF Connect Only-Favorites Filtering

Once you've selected the “Only Favorites” filter, you will only see favorites in the list.

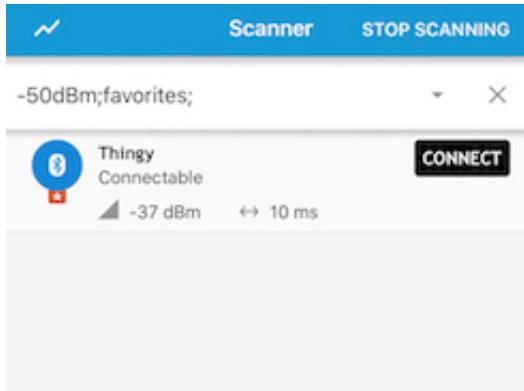


Figure 103: nRF Connect Only-Favorites Display

Once you've discovered the Thingy:52 (or the device of interest), Tap the name of the device (“Thingy” in this case) to show more information about the advertising Peripheral:

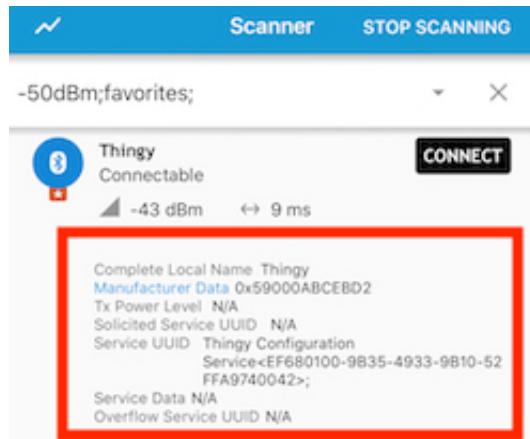


Figure 104: nRF Connect "Thingy" Selection

The information includes: Complete Local Name, Manufacturer Data, Tx Power Level, and others. The availability of this information depends on whether the device is transmitting this data in the Advertising packets or not.

Connections

Hit the **CONNECT** button next to the discovered device to connect to it. Once you've done so, the app will connect to the device and discover all its GATT's Services and Characteristics.

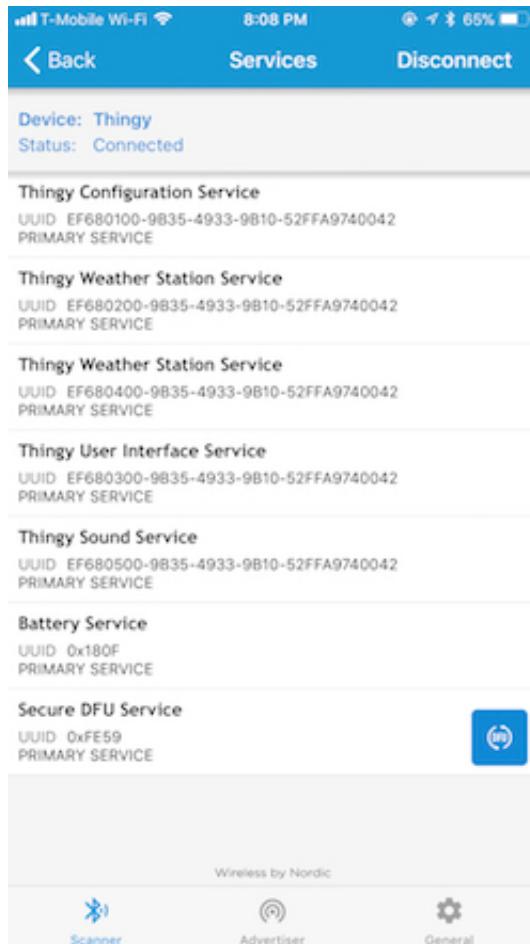


Figure 105: nRF Connect Connected to "Thingy"

Tip: Keep in mind that (usually) once a Peripheral is connected, it will stop Advertising and cannot be discovered by other Centrals. This also includes other applications running on the same smartphone. So, if you are switching between different applications such as LightBlue and nRF Connect, make sure you disconnect from the device before running the other application.

To disconnect, hit the **Disconnect** or **Back** button.

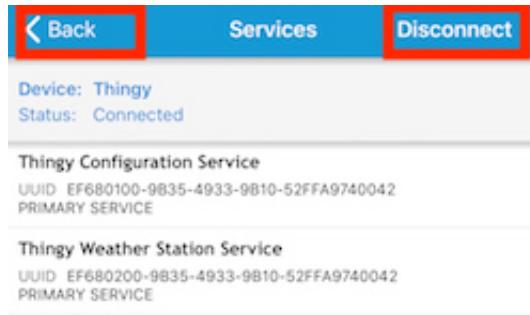


Figure 106: nRF Connect Disconnect Operation

GATT Services and Characteristics

Once you're connected to the Peripheral, you can interact with the device's Services and Characteristics. Let's explore some of the different Thingy:52 Services and Characteristics and see how we can interact with them.

Thingy Configuration Service

Click on the **Thingy Configuration Service**. You'll now be presented with a list of the Characteristics contained within the Service:

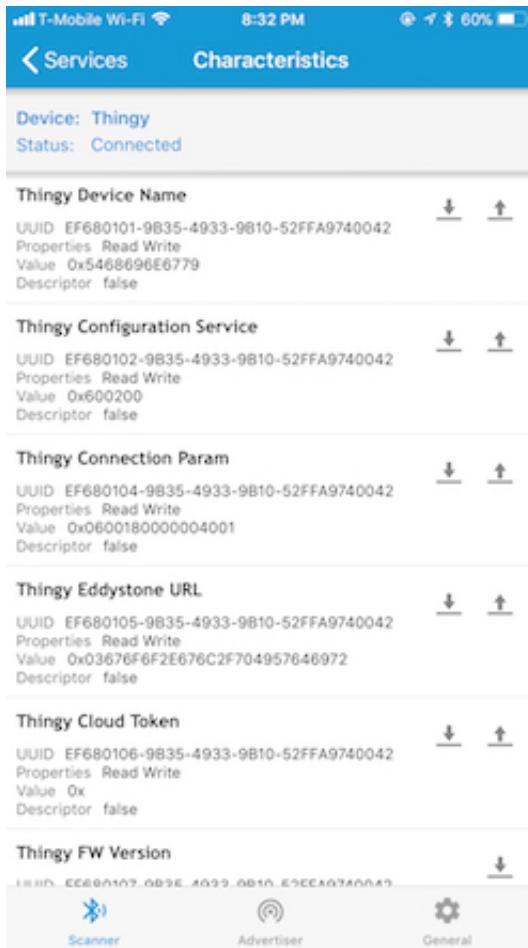


Figure 107: Thingy Configuration Service

You'll notice a few things:

- Some of the Characteristics will have different action buttons to the right side: a down arrow (for **writes** to the Characteristic), and an up arrow (for **reads** from a Characteristic).
- Under each Characteristic, you will see three fields: Properties, Value, and Descriptor. The **Properties** field tells us whether a Characteristic is "readable", "writeable", "notifiable", etc.
- Reading from the Characteristic simply updates the **Value** field with the new value.
- Writing to the Characteristic displays a popup that allows you to enter the value to be written. For a write to be successful, you would need to know the size of the value to write.
Tip: Sometimes, you can tell the size of the Characteristic by looking at the **Value** field.
- In the **Write** popup, you will see two options: Command, and Request. The difference between the two is that Commands do not require an acknowledgement from the Peripheral device making them unreliable, but requiring

less radio-on time. Requests, on the other hand, require an acknowledgment which makes them more reliable.

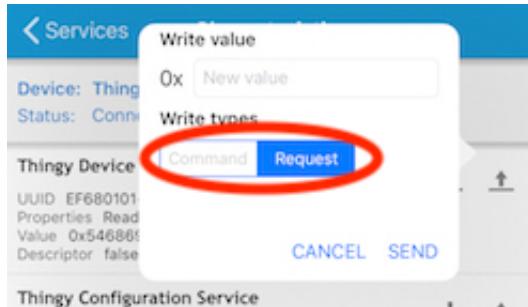


Figure 108: nRF Connect Write Operation

- One downside is that Values are displayed in HEX units. This makes it very difficult to interpret the values if they are meant to be strings. In this case, you would have to manually type the value into a HEX-to-ASCII converter to parse the data.
- One advantage for using nRF Connect with a device such as the Thingy:52 is that the application is programmed to interpret the custom Services and Characteristics and display their labels in human-readable text. In this case, we can see that all the Services and Characteristics are displayed with human-readable text instead of the 128-bit UUIDs only. This is because the nRF Connect app is aware of the Thingy's custom/vendor-specific UUIDs.
- *Tip: One advantage of using the LightBlue application over nRF Connect is that you can choose to represent values in other display units (Octal, Binary, and UTF-8 strings). For example, the Device Name Characteristic under the Thingy Configuration Service can be displayed in LightBlue as following:*

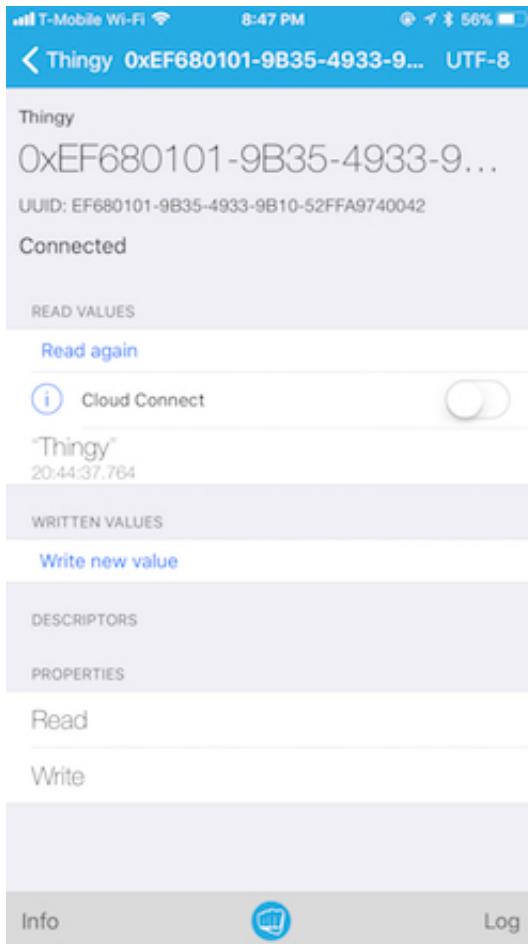
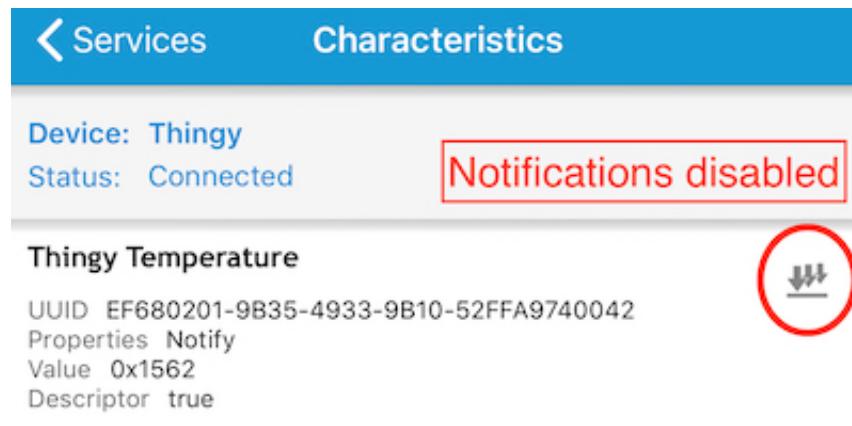
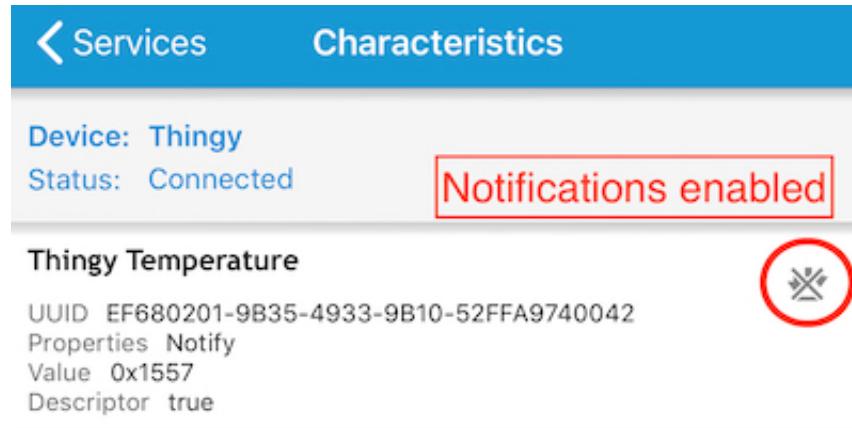


Figure 109: Device Name Display in LightBlue

Thingy Weather Station Service

Let's explore another service: the **Thingy Weather Station Service**.

Once you click on this service, you will see that most of the Characteristics have a different icon to the right side. This icon, which shows three arrows pointing down, indicates that the Characteristics has Notifications allowed. If you click on the icon, the application will now receive Notifications from the Peripheral device for whenever the value has changed. You can tell if you have Notifications enabled by noticing that the icon is now marked with an X. Clicking it again will disable Notifications for that Characteristic (the default state).

*Figure 110: Notifications Disabled**Figure 111: Notifications Enabled*

Enable Notifications for the **Thingy Temperature** Characteristic and you will notice updated values as the temperature of the Thingy's surrounding changes.

If a Characteristic has a **Descriptor**, then it will display **true** next to the **Descriptor** field. In this case, you can click on the Characteristic and see the Descriptor information for that Characteristic. Here it lists the Client Characteristic Configuration descriptor (CCCD), which gets used for enabling/disabling Notifications and Indications. (img)

Note: When using either of these applications, the connection will persist even if you exit out of the app. To disconnect from the Peripheral, you will have to manually disconnect from within the app or **hard kill** the app. In the case of LightBlue, Notifications will even show up as popups from the application (at least on iOS).

As you can see, using a Central Emulator Application can make testing small GATT changes simple and fast, especially compared to modifying a mobile app to support these changes.

Using a Bluetooth Sniffer

Using the Emulator Applications mentioned above is helpful in the case where you are debugging or testing GATT-related problems. If you already have a developed companion mobile application for your system, or you're looking to test and debug problems with an existing connection between the two devices, then these Emulator Applications may be rendered useless.

A Bluetooth sniffer is one of the most useful tools, and one that any Bluetooth developer can't do without. The difference between a Bluetooth sniffer and the Client Emulator apps mentioned previously is that sniffers can "spy" on the communication between your Central and Peripheral devices. The sniffer acts in the background without requiring you to modify any of your system's behavior, so it can help debug problems that may be hard to reproduce.

Bluetooth Sniffers come in two main varieties:

- Simple and low-cost sniffers (in the range of \$50-\$100) that are usually based on development kits.
- More advanced, higher-cost sniffers (anywhere between \$1,000-\$30,000).

The low-cost sniffers usually have limitations, such as the lack of advanced features on the desktop end as well as not being able to scan all the Advertising Channels simultaneously. Another advantage for using a commercial sniffer is that a lot of effort is usually put into the user experience, fixing any issues, as well as regularly adding new features.

Using a Bluetooth Low Energy sniffer can help tremendously in debugging problems with the connection and data transfer between the Peripheral and the Central device. Not only that, but it can also serve as a great educational tool for learning how the communication flow happens between two BLE devices.

Examples of sniffers include:

- **High-end/commercial:** [Ellisys sniffers](#), [Teledyne LeCroy sniffers](#) (*formerly Frontline*).
- **Low-cost:** [TI CC2540 USB dongle sniffer](#), [Nordic nRF51 sniffer](#), [Ubertooth One](#).

Here's a table comparing the different BLE sniffers in the market:

SNIFFER	PRICE	PROS	CONS
Adafruit Bluefruit LE Sniffer	\$30	Low cost Integrates with Wireshark (Windows only) through use of Nordic nRFSniffer software (command line utility) Linux and Mac OS X support provided through python scripts	Can listen on only one advertising channel at a time A bit of setup required Drops packets occasionally
TI BLE Sniffer (CC2540EMK-USB dongle)	\$50	Relatively easy-to-use Reasonable cost Minimal setup required	Can listen on only one advertising channel at a time Uses proprietary analysis application Difficult to export captured data Drops packets occasionally Crashes occasionally Dated software with 4.1 support only
Nordic nRF Sniffer(nRF51 PCA10031 USB dongle)	\$50	Reasonable price Integrates with Wireshark (Windows only)	Can listen on only one advertising channel at a time A bit of setup required Drops packets occasionally
Ubertooth One	\$120	Open-source software and hardware	Can listen on only one advertising channel at a time Difficult to get set up on Mac OS X or Windows (much simpler on Linux)
Teledyne LeCroy (formerly Frontline) ComProbe BPA low energy	\$1,000	Can listen to all 3 advertising channels simultaneously Compact design Powerful PC software	Relatively pricey Windows only Cumbersome UI (too many bells and whistles) Minimal tutorials available
Ellisys Bluetooth Tracker	\$10,000+	Very compact and portable Supports Bluetooth 5 Lower cost than other sniffers with comparable features	May not be affordable for some
Teledyne LeCroy Sodera LE Wideband Bluetooth® Protocol Analyzer	\$20,000	Uses software-defined radio (SDR): can be updated to support any future version of BLE (including Bluetooth 5)	Very expensive Cumbersome UI Windows only

Ellisys Bluetooth Explorer 400-STD-LE	\$30,000	Uses software-defined radio (SDR): can be updated to support any future version of BLE (including Bluetooth 5)	Very expensive
www.novelbits.io			

Figure 112: Comparison of Bluetooth Sniffers

Some of the scenarios where a BLE sniffer can prove useful include:

- Debugging issues with communication between two BLE devices such as random disconnections, dropped packets, and timing-related problems.
- Debugging GATT-related operations such as discovery of Services, discovery of Characteristics, Reads, Writes, Notifications and Indications.
- Debugging the different stages of security operations such as Pairing, Distribution of Keys, and Bonding between two BLE devices.
- Monitoring and decrypting encrypted data exchanged between two BLE devices (providing the link keys by the user will be a requirement in this use case).

We'll be going over how to use two of the sniffers: the [nRF-based sniffer] and the [Ellisys Bluetooth Tracker]. Most sniffers are accompanied by a desktop application that interfaces with the sniffer hardware and provides a user-friendly interface to capture and examine packets transmitted by BLE devices.

The nRF Sniffer

Setup

In order to run the nRF sniffer utility, you will need an nRF-based development kit or USB dongle (including nRF51822 DK, nRF51822 USB Dongle, nRF52832 DK, nRF52840 DK, nRF52840 USB Dongle). We'll be using an nRF51 USB dongle development kit to run the sniffer firmware required for the desktop sniffer software. The latest version of the nRF sniffer provides a plugin that runs within Wireshark rather than its own application.

In order to get this up and running, go to the [following link](#) and download the software as well as the sniffer user guide. Make sure you download version 2 (or higher) of the sniffer software and the user guide. Once you've downloaded the guide, follow the instructions listed there for installation and setup.

Next, start Wireshark and choose the **nRF Sniffer** as the Interface:

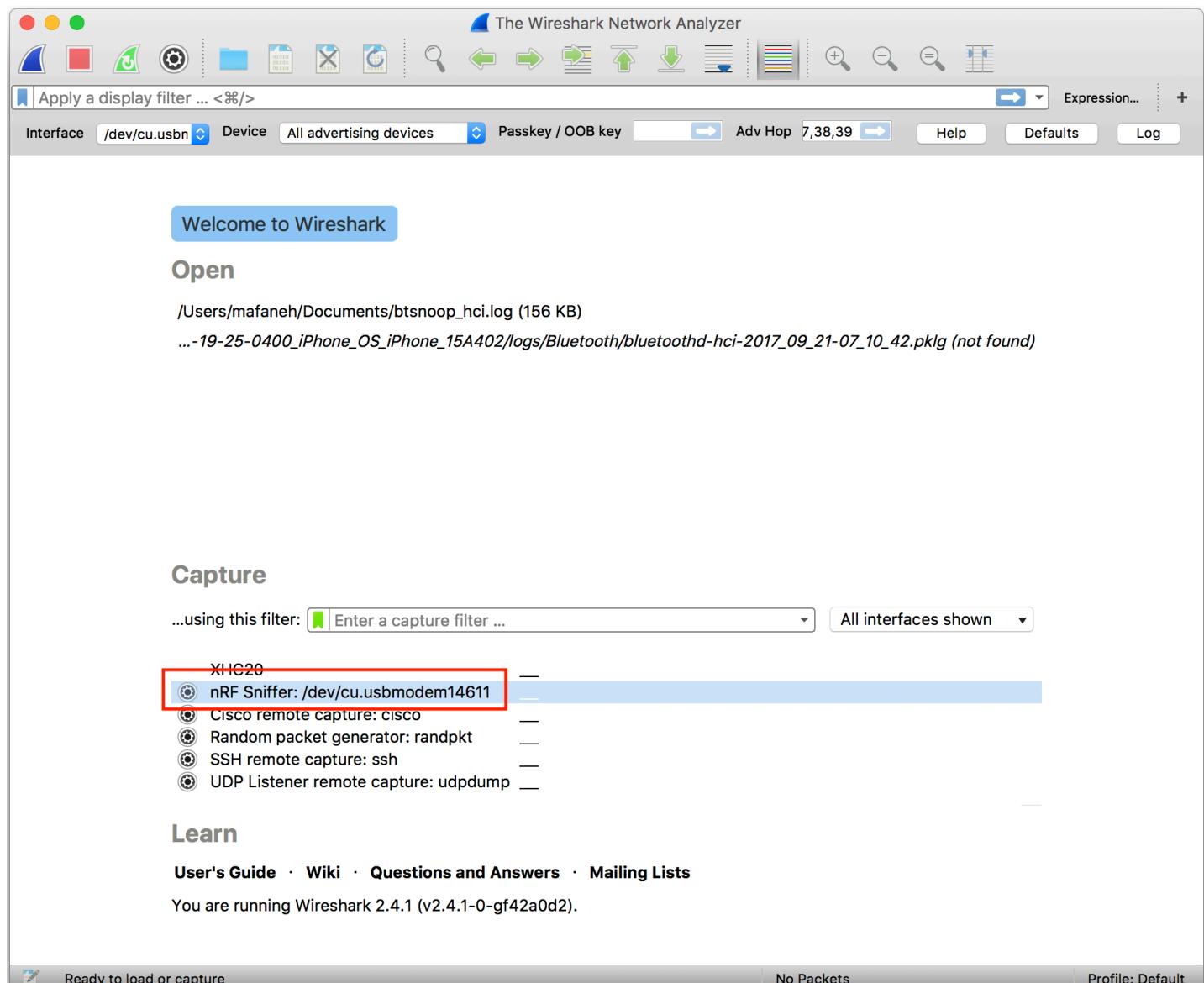


Figure 113: Wireshark nRF Sniffer Interface

The nRF Wireshark plugin adds its own toolbar, but it is not enabled by default. To enable it, navigate to **View --> Interface Toolbars** and make sure that **nRF Sniffer** is enabled:

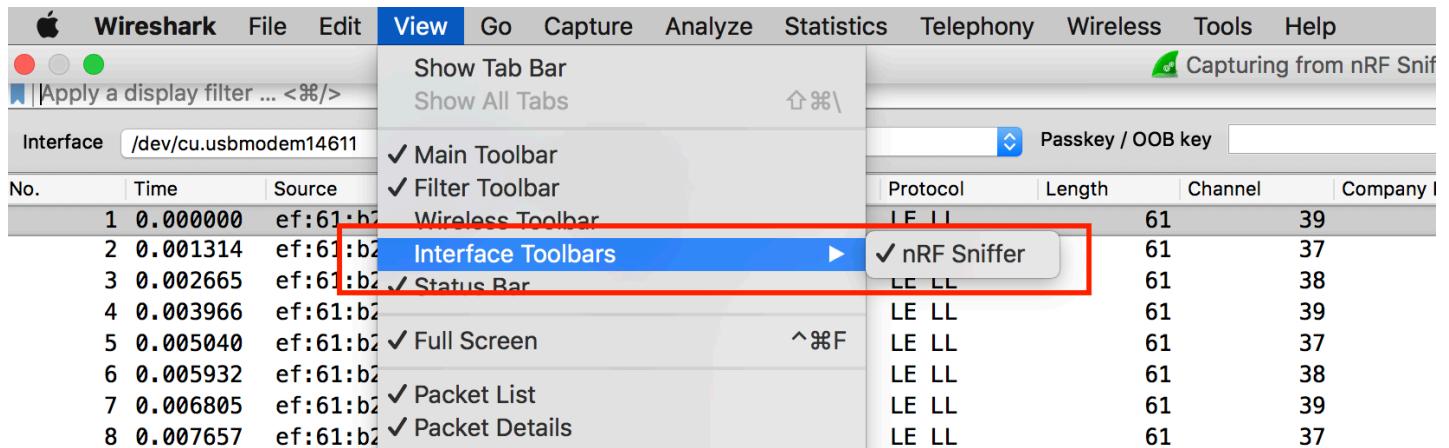


Figure 114: Wireshark Interface Toolbars

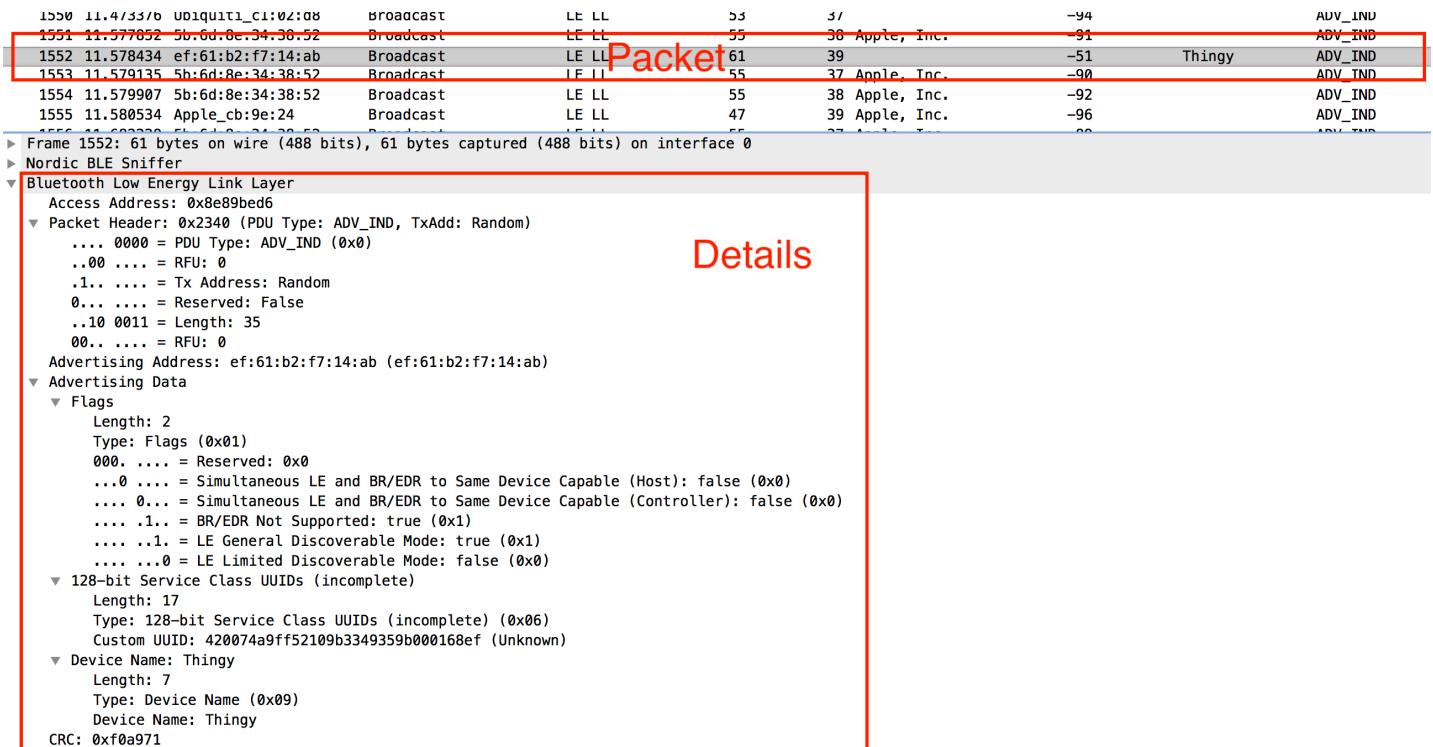
Next, enable automatic scrolling to see the packets as they arrive in real-time:



Figure 115: Wireshark Automatic Scrolling

Sniffing Advertisements

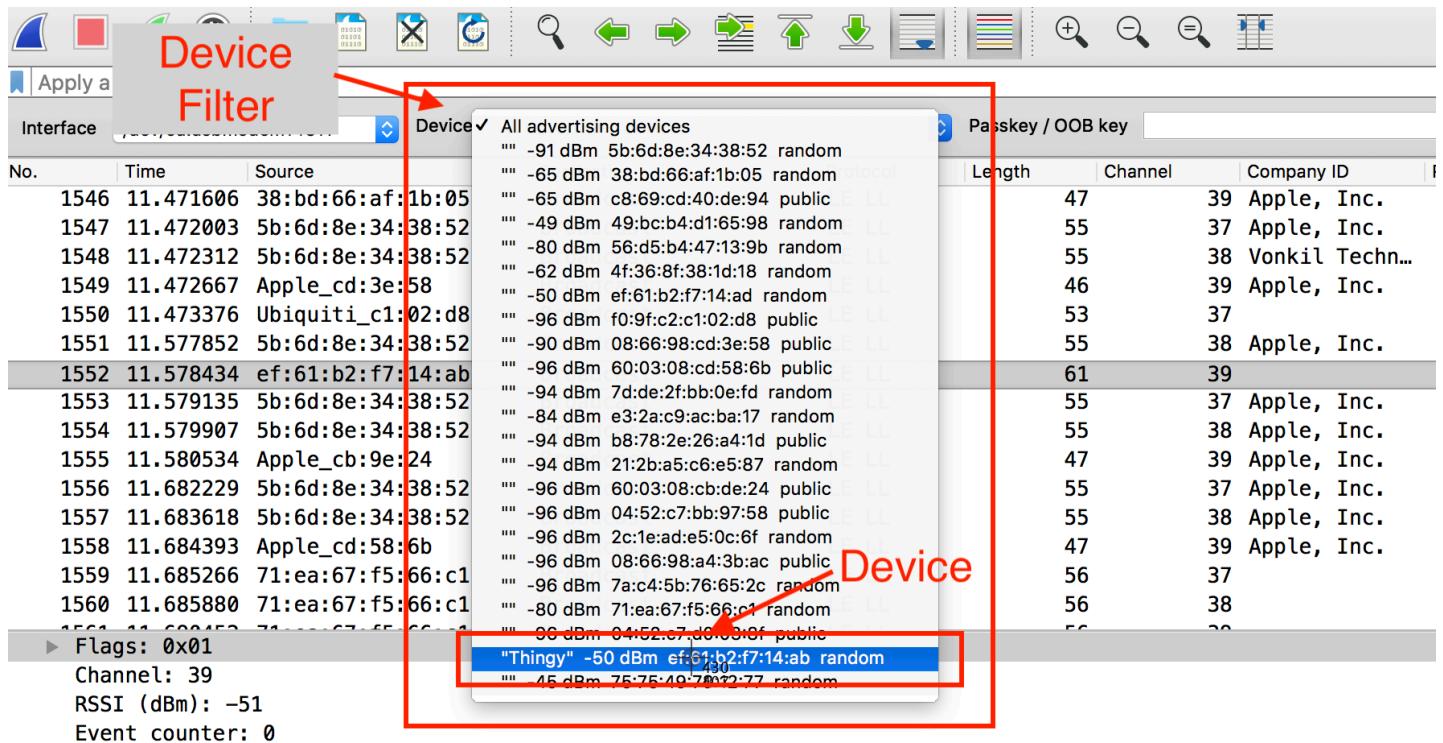
Once you've started the capture, the list of Advertisements will start showing up in the list. If you click on a specific advertisement, you can see more detailed information relating to that Advertisement Packet:

*Figure 116: Wireshark Advertisement Packet Details*

Some of the information shown includes:

- Nordic Sniffer information: RSSI, Channel, and more.
- The packet header information: PDU Type, Advertising Address, and more.
- The Advertisement Data: Advertisement Flags, UUIDs, device name, and more.
- CRC: the computed CRC transmitted by the sender.

To reduce the number of Advertisement packets shown in the capture, you can filter by a specific device of interest by choosing the device from the **Device** drop-down menu from within the nRF Sniffer toolbar:

**Figure 117: Wireshark Device Filters**

Another useful trick is to add any of the Advertisement data fields shown to the columns displayed in the capture window. You can do so by right-clicking on any field, and selecting **Apply as Column**:

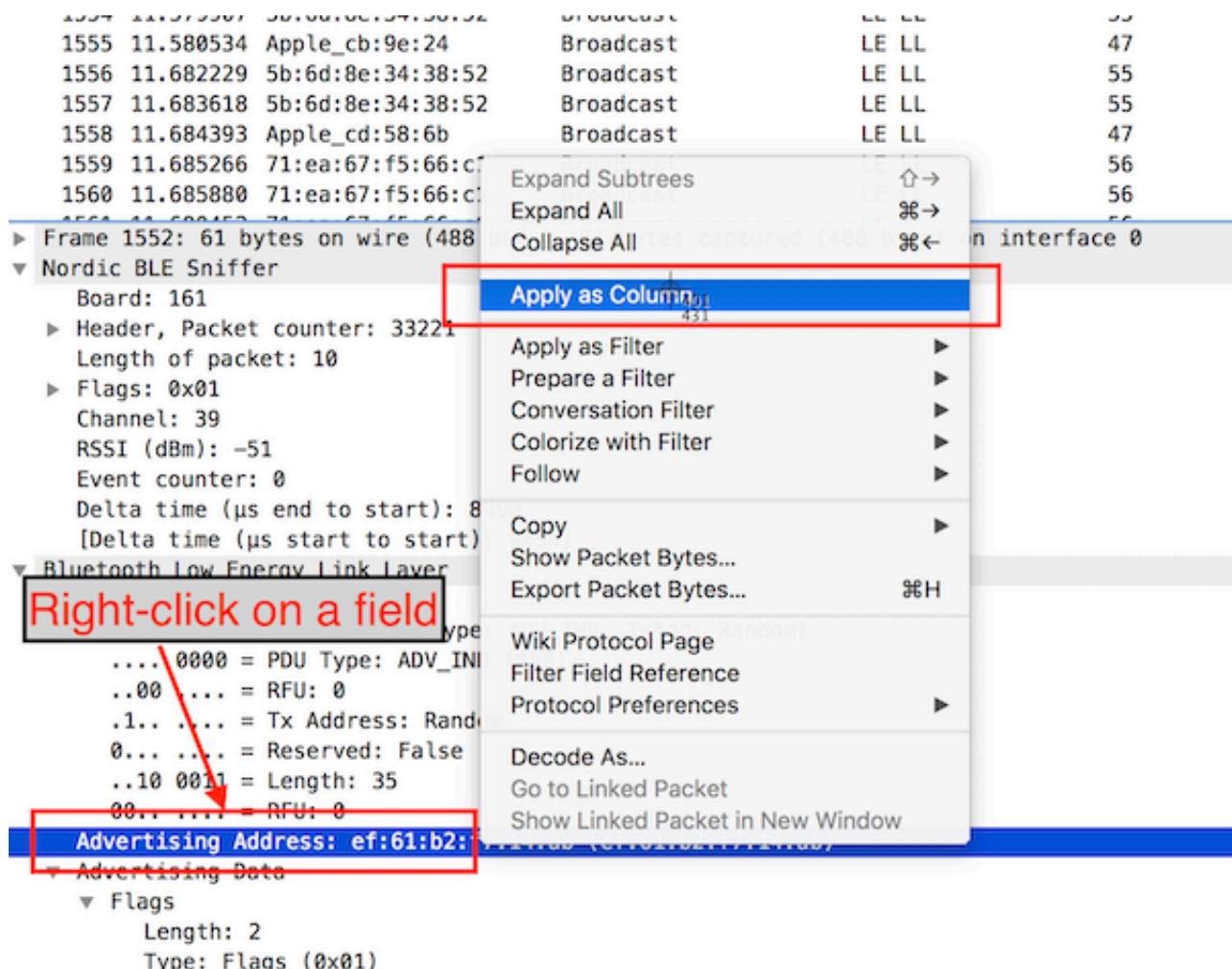


Figure 118: Wireshark Apply as Column Option

Sniffing Connections

In order to sniff a connection between two BLE devices, you need to select a device in the **Devices** drop-down menu from the **nRF Sniffer** toolbar. Once you've selected a device, the capture window will now start showing Connection Data Packets transferred between the connected devices.

Tip: Make sure you start the capture before the connection is established, otherwise the sniffer will not be able to follow the connection.

Here's a screenshot showing the capture of a connection, and more specifically the Connection Request Packet:

Frame Number	Source Address	Destination Address	Protocol	Link Layer Type	Length	Sequence Number	Timestamp	Source MAC	Destination MAC	Action
669	104.3150...	ef:61:b2:f7:14:ab	Broadcast	LE LL	61	39	-50	Thingy	ef:61:b2:f7:14:ab	ADV_IND
670	104.7205...	ef:61:b2:f7:14:ab	Broadcast	LE LL	61	37	-46	Thingy	ef:61:b2:f7:14:ab	ADV_IND
671	104.7210...	56:18:06:a3:d2:38	ef:61:b2:f7:14:ab	LE LL	60	37	-48	ef:61:b2:f7:14:ab	CONNECT_REQ	
672	104.7214...	Master_0xaf9a9de2	Slave_0xaf9a9de2	LE LL	26	6	-48			Empty PDU
673	104.7218...	Slave_0xaf9a9de2	Master_0xaf9a9de2	LE LL	26	6	-46			Empty PDU
674	104.7222...	Master_0xaf9a9de2	Slave_0xaf9a9de2	LE LL	32	6	-48			Control Opcode: l
675	104.8269...	Master_0xaf9a9de2	Slave_0xaf9a9de2	LE LL	32	12	-51			Control Opcode: l
676	104.8275...	Slave_0xaf9a9de2	Master_0xaf9a9de2	LE LL	26	12	-46			Empty PDU
677	104.8294...	Master_0xaf9a9de2	Slave_0xaf9a9de2	LE LL	26	34	-52			Empty PDU
678	104.8300...	Slave_0xaf9a9de2	Master_0xaf9a9de2	LE LL	32	34	-50			Control Opcode: l
679	104.8304...	Master_0xaf9a9de2	Slave_0xaf9a9de2	LE LL	35	4	-49			Control Opcode: l

► Frame 671: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0
 ► Nordic BLE Sniffer
 ▾ Bluetooth Low Energy Link Layer
 Access Address: 0x8e89bed6
 ▼ Packet Header: 0x22c5 (PDU Type: CONNECT_REQ, TxAdd: Random, RxAdd: Random)
 0101 = PDU Type: CONNECT_REQ (0x5)
 ..00 = RFU: 0
 .1.. = Tx Address: Random
 1... = Rx Address: Random
 ..10 0010 = Length: 34
 00.. = RFU: 0
 Initiator Address: 56:18:06:a3:d2:38 (56:18:06:a3:d2:38)
 Advertising Address: ef:61:b2:f7:14:ab (ef:61:b2:f7:14:ab)
 ▼ Link Layer Data
 Access Address: 0xaf9a9de2
 CRC Init: 0x63fd6
 Window Size: 3 (3.75 msec)
 Window Offset: 14 (17.5 msec)
 Interval: 24 (30 msec)
 Latency: 0
 Timeout: 72 (90 msec)
 Channel Map: ffff030014
 ...0 0110 = Hop: 6
 001. = Sleep Clock Accuracy: 151 ppm to 250 ppm (1)
 CRC: 0x3e3d2b

Details

Figure 119: Connection Request Packet

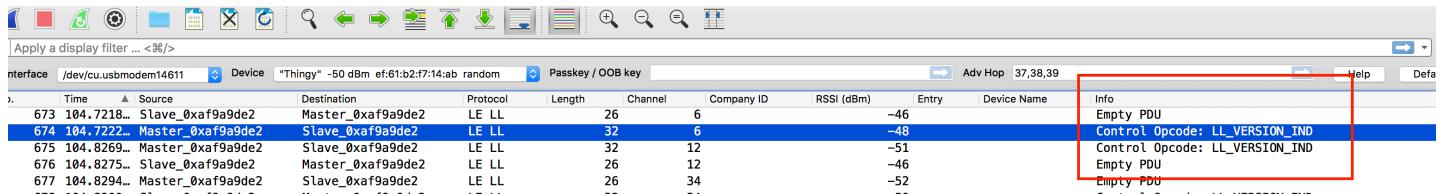
Once you've succeeded in following a connection, you can see all the Data Packets being exchanged between the two devices.

The most important columns to look at include:

- **Time:** absolute time from the beginning of the capture (in seconds). This field is very helpful in debugging issues related to timing.
- **Source & Destination:** determines whether the packet was transmitted from the Master to the Slave, or vice versa.
- **Protocol:** tells you whether this packet was a Link Layer Packet (something the Controller probably handled in isolation of the Application/Host layer), or an ATT packet (a packet that was initiated or handled by your application).
- **Info:** displays more information about the payload of the packet.

Note: You will usually see many Empty PDU packets being exchanged between the two devices. This is due to the requirement that each device needs to transmit some data at each Connection Interval, even if they don't have meaningful data to transmit.

Let's examine some of the packets captured in their order of transmission. We'll be focusing on the **Info** field to determine if a packet is important or not.

**Figure 120: Wireshark Info Column**

The purpose of the packet shown above is to exchange information about the Bluetooth version and Manufacturer of each device. In this case, the device has a Broadcom Bluetooth chipset running Bluetooth version 4.2.

```

Bluetooth Low Energy Link Layer
Access Address: 0xaf9a9de2
[Master Address: 56:18:06:a3:d2:38 (56:18:06:a3:d2:38)]
[Slave Address: ef:61:b2:f7:14:ab (ef:61:b2:f7:14:ab)]
▶ Data Header: 0x060b
Control Oopcode: LL VERSION IND (0x0c)
Version Number: 4.2 (0x08)
Company Id: Broadcom Corporation (0x0f)
Subversion Number: 0x6103
CRC: 0xeb59e0

```

Figure 121: Advertisement Bluetooth version and Company ID

In the following packet, the Master device is sending the Slave a FEATURE_REQ packet. The packet details views shows the set of flags that indicate which features are supported by the Master device. In response to this packet, the Slave will send a FEATURE_RSP packet including a set of flags indicating its supported features.

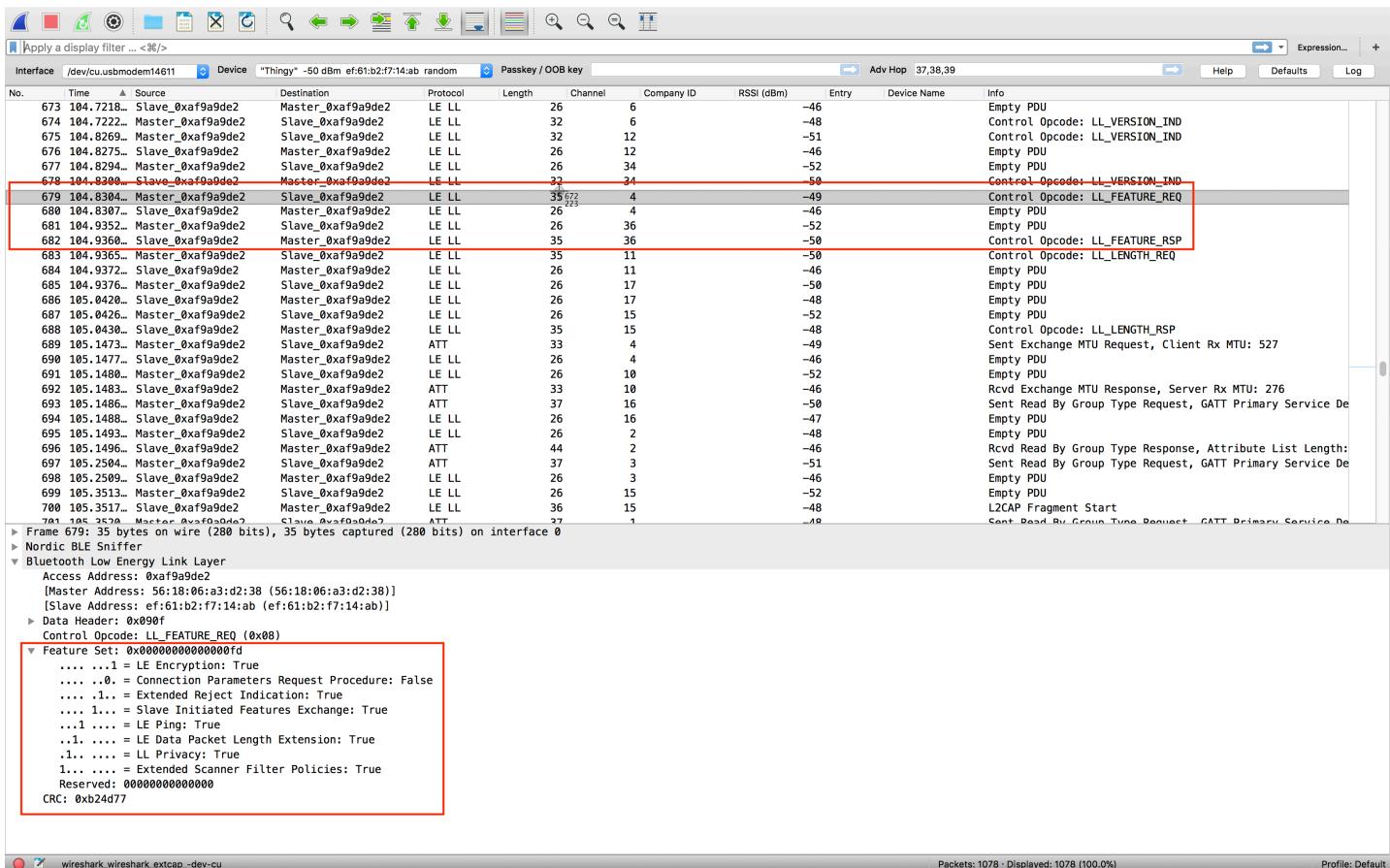


Figure 122: Feature Request and Response Packets

In the following packet, the Master is sending the Slave a LENGTH_REQ packet which includes the ATT Maximum Transmission Unit (MTU) size supported by the Master. In response to this packet, the Slave sends a LENGTH_RSP packet including the MTU size it supports. After the Master receives this packet, the smaller of these sizes is chosen to be the maximum MTU size of packets exchanged between the two devices.

5880	69.945255	Slave_0xat9a8c95	Master_0xat9a8c95	LE LL	20	23	-44	Empty PDU
5887	69.945441	Master_0xaf9a8c95	Slave_0xaf9a8c95	LE LL	26	1	-46	Empty PDU
5888	69.945645	Slave_0xaf9a8c95	Master_0xaf9a8c95	LE LL	35	1	-54	Control Opcode: LL_FEATURE_RSP
5889	69.945859	Master_0xaf9a8c95	Slave_0xaf9a8c95	LE LL	35	16	-44	Control Opcode: LL_LENGTH_REQ
5890	69.946063	Slave_0xaf9a8c95	Master_0xaf9a8c95	LE LL	26	16	-53	Empty PDU
5891	70.049347	Master_0xaf9a8c95	Slave_0xaf9a8c95	LE LL	26	31	-45	Empty PDU
5892	70.049777	Slave_0xaf9a8c95	Master_0xaf9a8c95	LE LL	26	31	-56	Empty PDU
5893	70.050117	Master_0xaf9a8c95	Slave_0xaf9a8c95	LE LL	26	9	-46	Empty PDU
5894	70.050396	Slave_0xaf9a8c95	Master_0xaf9a8c95	LE LL	35	9	-54	Control Opcode: LL_LENGTH_RSP
5895	70.050626	Master_0xaf9a8c95	Slave_0xaf9a8c95	ATT	33	24	-45	Sent Exchange MTU Request, Client
5896	70.050839	Slave_0xaf9a8c95	Master_0xaf9a8c95	LE LL	26	24	-56	Empty PDU
5897	70.051059	Master_0xaf9a8c95	Slave_0xaf9a8c95	LE LL	26	2	-45	Empty PDU
5898	70.051279	Slave_0xaf9a8c95	Master_0xaf9a8c95	ATT	33	2	-56	Rcvd Exchange MTU Response, Server
5899	70.153773	Master_0xaf9a8c95	Slave_0xaf9a8c95	ATT	37	17	-46	Sent Read By Group Type Request, G
5900	70.154349	Slave_0xaf9a8c95	Master_0xaf9a8c95	LE LL	26	17	-58	Empty PDU
5901	70.154792	Master_0xaf9a8c95	Slave_0xaf9a8c95	LE LL	26	32	-46	Empty PDU
5902	70.155190	Slave_0xaf9a8c95	Master_0xaf9a8c95	ATT	44	32	-67	Rcvd Read By Group Type Response,
5903	70.155662	Master_0xaf9a8c95	Slave_0xaf9a8c95	ATT	37	10	-48	Sent Read By Group Type Request, G
5904	70.156035	Slave_0xaf9a8c95	Master_0xaf9a8c95	LE LL	26	10	-60	Empty PDU
5905	70.156397	Master_0xaf9a8c95	Slave_0xaf9a8c95	LE LL	26	25	-47	Empty PDU
5906	70.156651	Slave_0xaf9a8c95	Master_0xaf9a8c95	LE LL	36	25	-66	L2CAP Fragment Start
5907	70.156895	Master_0xaf9a8c95	Slave_0xaf9a8c95	ATT	37	3	-47	Sent Read By Group Type Request, G

► Frame 5889: 35 bytes on wire (280 bits), 35 bytes captured (280 bits) on interface 0
 ► Nordic BLE Sniffer
 ► Bluetooth Low Energy Link Layer
 Access Address: 0xaf9a8c95
 [Master Address: 76:47:fd:cd:e8:53 (76:47:fd:cd:e8:53)]
 [Slave Address: ef:61:b2:f7:14:ab (ef:61:b2:f7:14:ab)]
 ► Data Header: 0x090f
 Control Opcode: LL_LENGTH_REQ (0x14)
 Max RX octets: 251
 Max RX time: 2120 microseconds
 Max TX octets: 251
 Max TX time: 2120 microseconds
 CRC: 0xa0fb6

Figure 123: Length Request and Response Packets

In the following packet, the Master is requesting to read the Attributes exposed by the Slave. The Master specifies the range of attribute handles it wants to know about. Notice the arrows next to the packet sequence number (far left column). These indicate the Request and the corresponding Response. In response to the Request, the Slave send back a list of UUIDs, each with the Handle range that they span.

Interface	/dev/cu.usbmodem14611	Device	"Thingy" -48 dBm ef:61:b2:f7:14:ab random	Passkey / OOB key	Adv Hop	37.38.39	Help	Defaults	Log
No.	Time	Source	Destination	Protocol	Length	Channel	RSSI (dBm)	Entry	Info
5890	70.051279	Slave_0xaf9a8c95	Master_0xaf9a8c95	ATT	33	2	-56		Rcvd Exchange MTU Response, Server Rx MTU: 276
5899	70.153773	Master_0xaf9a8c95	Slave_0xaf9a8c95	ATT	37	17	-46		Sent Read By Group Type Request, GATT Primary Service Declaration, Handles: 0x0001..0xffff
5900	70.154349	Slave_0xaf9a8c95	Master_0xaf9a8c95	LE LL	26	17	-58		Empty PDU
5901	70.154792	Master_0xaf9a8c95	Slave_0xaf9a8c95	LE LL	26	32	-46		Empty PDU
5902	70.155190	Slave_0xaf9a8c95	Master_0xaf9a8c95	ATT	44	32	-67		Rcvd Read By Group Type Response, Attribute List Length: 2, Generic Access Profile, Generic Attribute Prof
5903	70.155662	Master_0xaf9a8c95	Slave_0xaf9a8c95	ATT	37	10	-48		Sent Read By Group Type Request, GATT Primary Service Declaration, Handles: 0x000e..0xffff
5904	70.156035	Slave_0xaf9a8c95	Master_0xaf9a8c95	LE LL	26	10	-60		Empty PDU
5905	70.156397	Master_0xaf9a8c95	Slave_0xaf9a8c95	LE LL	26	25	-47		Empty PDU
5906	70.156651	Slave_0xaf9a8c95	Master_0xaf9a8c95	LE LL	36	25	-66		L2CAP Fragment Start
5907	70.156895	Master_0xaf9a8c95	Slave_0xaf9a8c95	ATT	37	3	-47		Sent Read By Group Type Request, GATT Primary Service Declaration, Handles: 0x0060..0xffff
5908	70.157126	Slave_0xaf9a8c95	Master_0xaf9a8c95	I/F/I	26	3	-65		Fmtrv PnII

Figure 124: Reading Attributes and Handles

For example, in the following screenshot, you can see that the Slave returns two Attributes along with their Handle ranges: UUID 0x1800 (Generic Access Profile) which has a Handle range of 0x0001-0x0009, and UUID 0x1801 (Generic Attribute Profile) which has a Handle range of 0x000a-0x000d.

5900	70.154349	Slave_0xa9a8c95	Master_0xa9a8c95	LE LL	26	17	-58	Empty PDU
5901	70.154792	Master_0xa9a8c95	Slave_0xa9a8c95	LE LL	26	32	-46	Empty PDU
5902	70.155198	Slave_0xa9a8c95	Master_0xa9a8c95	ATT	44	32	-67	Rcvd Read By Group Type Response, Attribute List Length: 2, Generic Access Profile, Generic Attribute Prof
5903	70.155662	Master_0xa9a8c95	Slave_0xa9a8c95	ATT	37	10	-48	Sent Read By Group Type Request, GATT Primary Service Declaration, Handles: 0x000e..0xffff
5904	70.156035	Slave_0xa9a8c95	Master_0xa9a8c95	LE LL	26	10	-60	Empty PDU
5905	70.156397	Master_0xa9a8c95	Slave_0xa9a8c95	LE LL	26	25	-47	Empty PDU
5906	70.156651	Slave_0xa9a8c95	Master_0xa9a8c95	LE LL	36	25	-66	L2CAP Fragment Start
5907	70.156895	Master_0xa9a8c95	Slave_0xa9a8c95	ATT	37	3	-47	Sent Read By Group Type Request, GATT Primary Service Declaration, Handles: 0x0060..0xffff
5908	70.157126	Slave_0xa9a8c95	Master_0xa9a8c95	LE LL	26	3	-65	Empty PDU
5909	70.259233	Master_0xa9a8c95	Slave_0xa9a8c95	LE LL	26	18	-47	Empty PDU
5910	70.259732	Slave_0xa9a8c95	Master_0xa9a8c95	ATT	44	18	-62	Rcvd Read By Group Type Response, Attribute List Length: 2, Battery Service, Nordic Semiconductor ASA
5911	70.260372	Master_0xa9a8c95	Slave_0xa9a8c95	ATT	37	33	-58	Sent Read By Type Request, GATT Characteristic Declaration, Handles: 0x000a..0x000d
5912	70.260705	Slave_0xa9a8c95	Master_0xa9a8c95	LE LL	26	33	-59	Empty PDU
5913	70.260946	Master_0xa9a8c95	Slave_0xa9a8c95	LE LL	26	11	-48	Empty PDU
5914	70.261168	Slave_0xa9a8c95	Master_0xa9a8c95	ATT	39	11	-56	Rcvd Read By Type Response, Attribute List Length: 1, Service Changed
5915	70.261393	Master_0xa9a8c95	Slave_0xa9a8c95	ATT	35	26	-48	Sent Find Information Request, Handles: 0x000d..0x000d
5916	70.261607	Slave_0xa9a8c95	Master_0xa9a8c95	LE LL	26	26	-59	Empty PDU
5917	70.362562	Master_0xa9a8c95	Slave_0xa9a8c95	LE LL	26	4	-48	Empty PDU
5918	70.364059	Slave_0xa9a8c95	Master_0xa9a8c95	ATT	36	4	-57	Rcvd Find Information Response, Handle: 0x000d (Generic Attribute Profile: Service Changed: Client Charact
5919	70.364548	Master_0xa9a8c95	Slave_0xa9a8c95	ATT	35	19	-46	Sent Write Request, Handle: 0x000d (Generic Attribute Profile: Service Changed: Client Characteristic Conf
5920	70.366782	Slave_0xa9a8c95	Master_0xa9a8c95	LE LL	26	19	-56	Empty PDU
5921	70.369493	Master_0xa9a8c95	Slave_0xa9a8c95	LE LL	26	34	-46	Empty PDU
5922	70.370873	Slave_0xa9a8c95	Master_0xa9a8c95	ATT	31	34	-56	Rcvd Write Response, Handle: 0x000d (Generic Attribute Profile: Service Changed: Client Characteristic Config

Access Address: 0xa9a8c95
[Master Address: 76:47:fd:cd:e8:53 (76:47:fd:cd:e8:53)]
[Slave Address: ef:61:b2:f7:14:ab (ef:61:b2:f7:14:ab)]
> Data Header: [0x120a
[L2CAP Index: 3]
CRC: 0x26e0e6]
▼ Bluetooth L2CAP Protocol
Length: 14
CID: Attribute Protocol (0x0004)
▼ Bluetooth Attribute Protocol
 ▼ Opcode: Read By Group Type Response (0x11)
 0... = Authentication Signature: False
 .0.. = Command: False
 ..01 0001 = Method: Read By Group Type Response (0x11)
 Length: 6
 Attribute Data, Handle: 0x0001, Group End Handle: 0x0009, UUID: Generic Access Profile
 Handle: 0x0001 (Generic Access Profile)
 Group End Handle: 0x0009
 UUID: Generic Access Profile (0x1800)
Attribute Data, Handle: 0x000a, Group End Handle: 0x000d, UUID: Generic Attribute Profile
 Handle: 0x000a (Generic Attribute Profile)
 Group End Handle: 0x000d
 UUID: Generic Attribute Profile (0x1801)
 [UUID: GATT Primary Service Declaration (0x2800)]
 [Request in Frame: 5899]

Figure 125: Response to Attribute Read

In the following packet, the Master is requesting a read of the Battery level Characteristic of the Slave. In the response, you can see that the reported Battery level has a value of 91%.

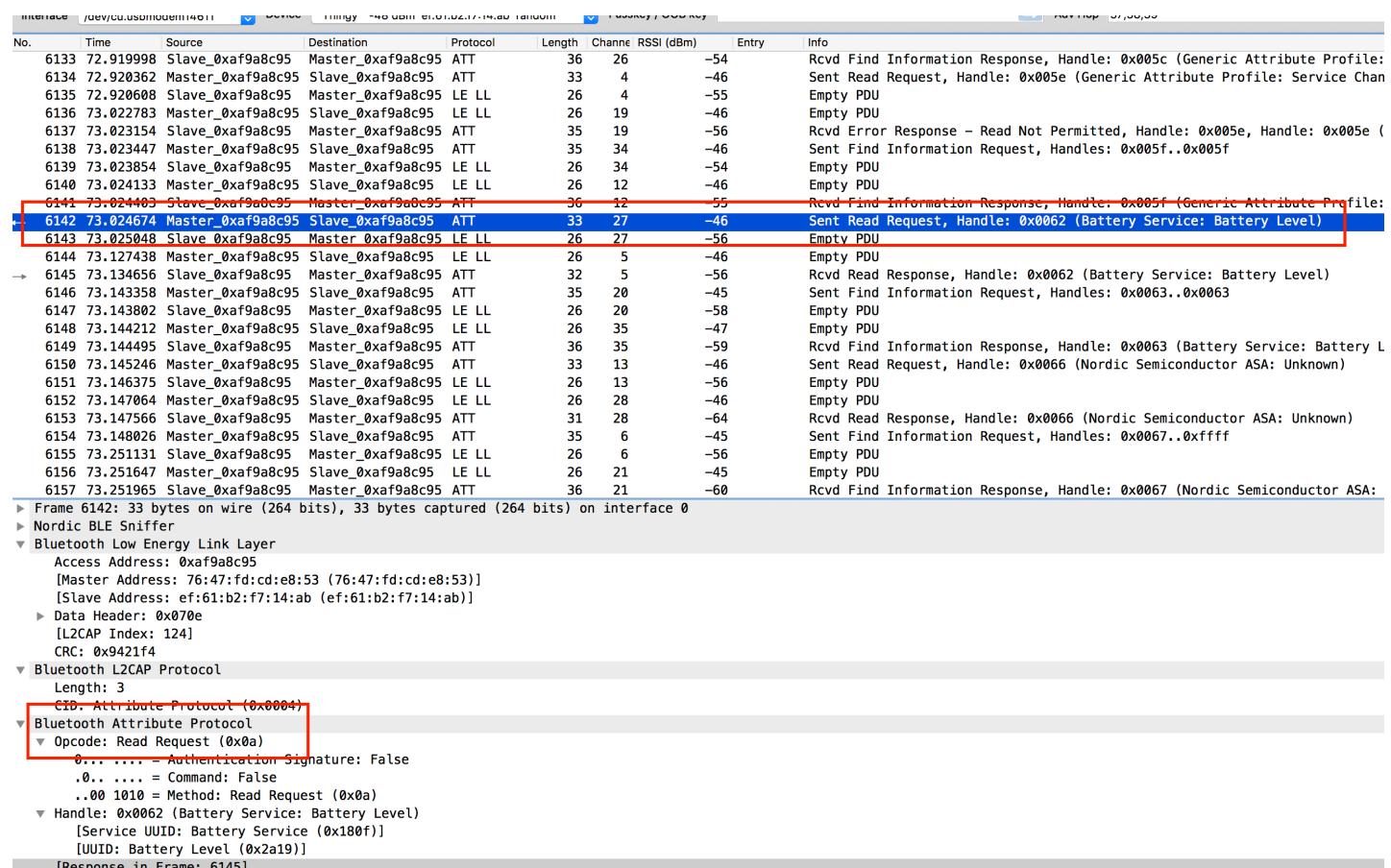


Figure 126: Battery Level Read Request

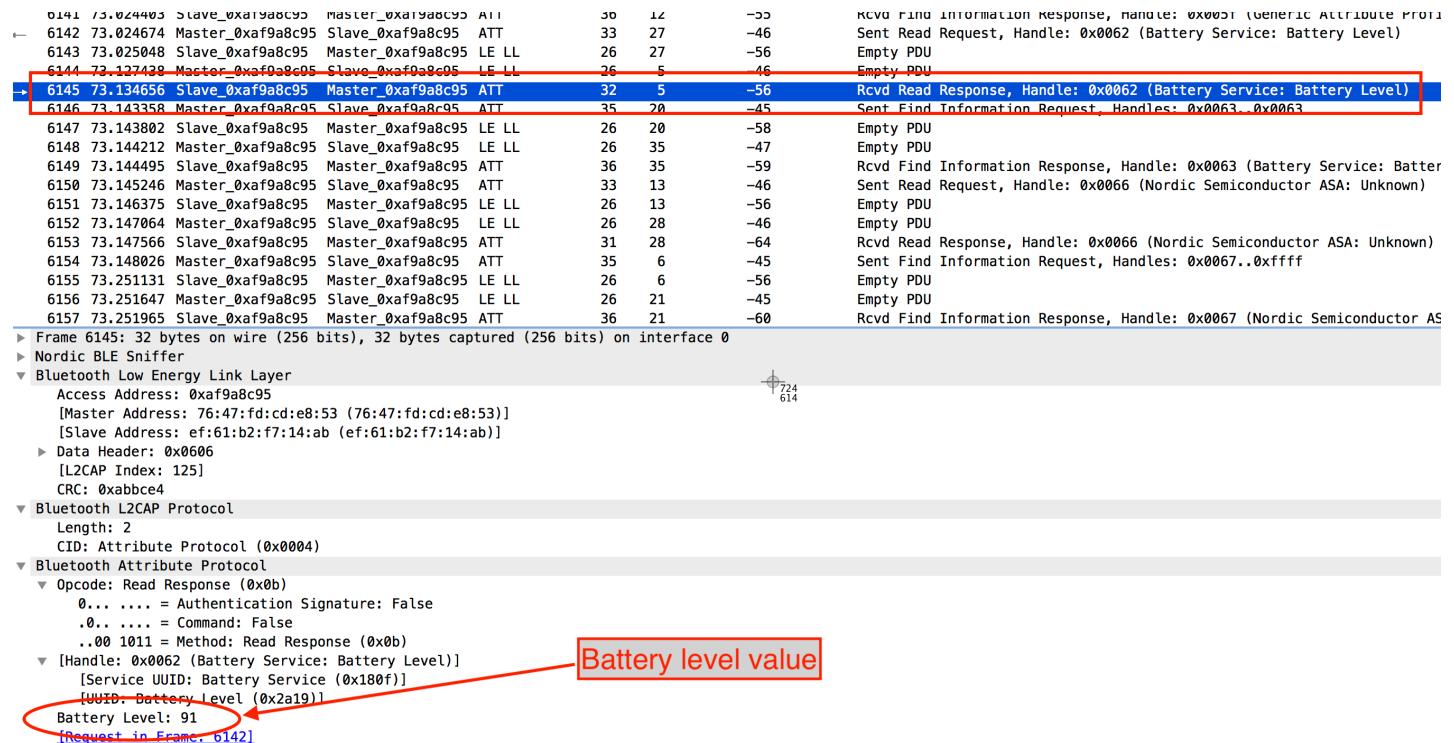


Figure 127: Battery Level Read Response

Ellisys Bluetooth Tracker

The [Ellisys Bluetooth Tracker](#) is a commercial sniffer that has a wide range of powerful capabilities. One of the main advantages of using a sniffer such as the Ellisys Bluetooth Tracker is that its software is “update-able” to support new versions of the Bluetooth specification as well as the addition of new features — all without requiring a hardware change.

For example, support for Bluetooth 5 and Bluetooth mesh was provided as a software update to the sniffer firmware with no need for upgraded hardware.

The cost of this sniffer and other similar ones is quite high, but the investment will pay off significantly in terms of time saved while debugging critical and complex issues (especially when they relate to timing of packets and communications).

I chose to include instructions for using the Ellisys sniffer, even though it's quite expensive (\$10,000+) for two reasons:

- Even if you do not have access to the sniffer hardware, you can request a FREE download of the Sniffer PC software (*instructions provided below*) and analyze saved sniffer capture files.
- To provide a comparison of what a commercial sniffer can do compared to a low-cost development-kit-based sniffer.

Free Download of Ellisys Sniffer Software

The Ellisys sniffer software is not available for download by the general public. However, you can request a link to download it for FREE by sending an email to download@ellisys.com and mentioning in the body/subject that you are a "Bluetooth 5 & Bluetooth Low Energy: A Developer's Guide" reader. The download link will then be sent to you in response.

Along with this e-book, I've provided a few capture files that you can download and analyze to better understand and follow along with the previous examples. You will find the folder in the GitHub repository accompanying this book, titled **Ellisys sniffer captures**.

Capture Files

The Ellisys Bluetooth Tracker software provides you with the capability to save captures for later viewing or sharing with others. To view a previously saved capture file, you do not need the sniffer hardware — you can simply run the software application and open the file for viewing and analysis.

Sniffing Advertisements

One of the powerful features of Ellisys's user interface is that Advertisements and Data Packets sent by one device are layered and sorted hierarchically to make it easy to navigate.

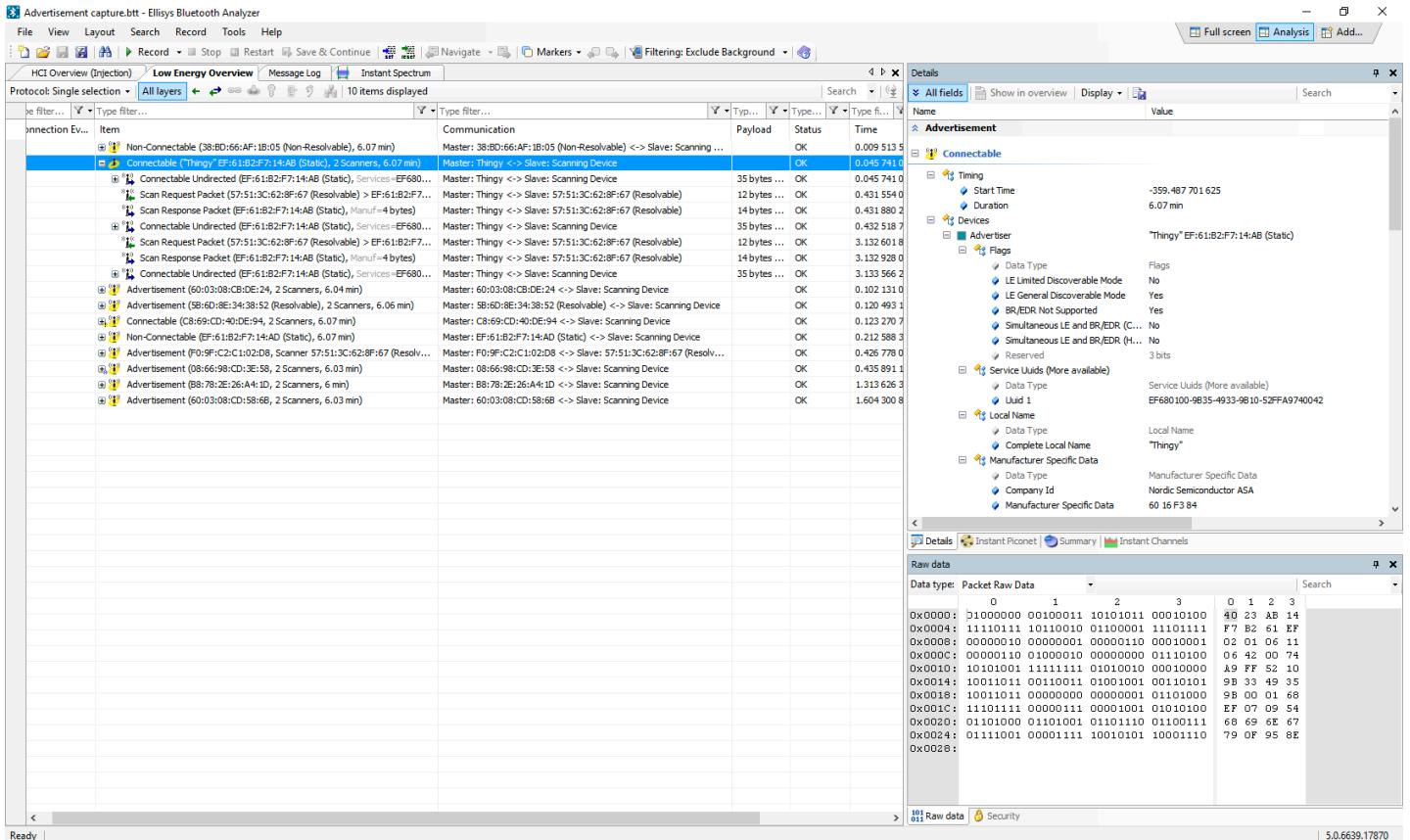


Figure 128: Ellisys Analyzer Software

In the previous screenshot, you can see that all Advertisement Packets sent by the Thingy:52 are organized in a tree view. This allows you to see all Advertising devices in one view, making it easier to find the device in question, and giving you the ability to expand the packets as needed.

Another view that is useful in showing Advertisements and the relationship between surrounding BLE devices is the **Instant Piconet** view which shows the different BLE devices and how they are interacting with each other. As you can see, the capture shows multiple scanning devices and the surrounding Peripherals that are discovered by the Centrals.

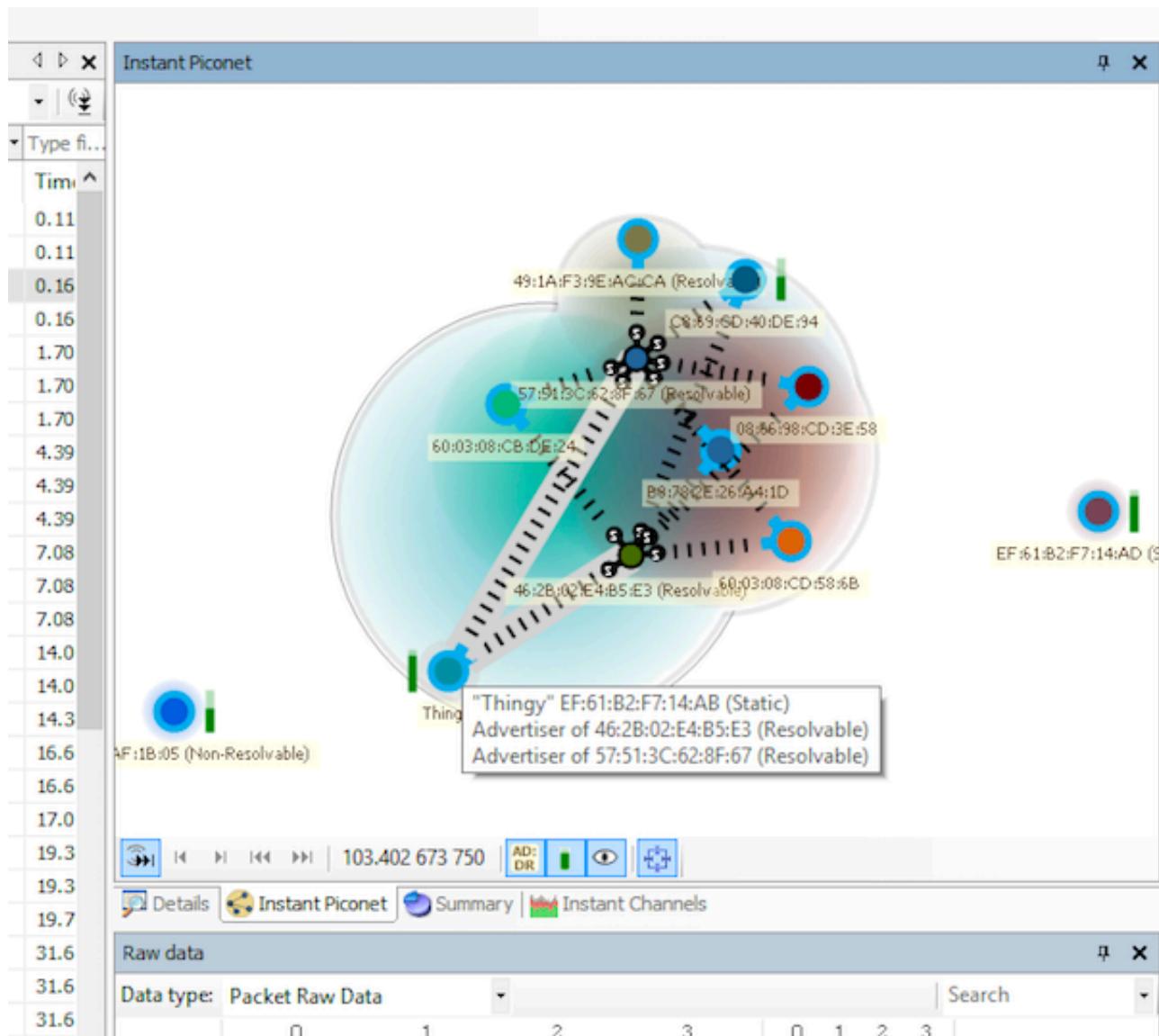


Figure 129: Piconet View

In order to view Advertisement packets and their details, click on one of the Advertisements and make sure the **Details** view is selected in the right hand-side. Here, you are presented with a lot of details related to the selected packet:

Details

All fields | Show in overview | Display | Search

Name	Value
Link-Layer Information	
Sniffer Radio	
RX Strength (RSSI)	-33.0 dBm
RX Quality	High
RF Gain	6.0 dB
RF Channel	
RF Channel Frequency	2402 Mhz
RF Channel Number	0
RF Channel Index	37 (adv)
Initial Center Frequency Offset	+54.7 kHz
Link Layer	
PHY	LE 1M
Coding Scheme	Uncoded (1 Mbps)
Access Address	0x8E89BED6
Received Access Address	0x8E89BED6
CRC Initial Seed	0x555555
Physical Channel	Advertisement ("Thingy" EF:61:B2:F7:14:AB (Static))
Timing	
Start Time	0.045 741 000
Duration	360 us
Delta from Previous	First
Devices	
Originator	Master
Transmitter	Master: "Thingy" EF:61:B2:F7:14:AB (Static)
Receiver	Slave: "Scanning Device"
Master Address	FF:61:B2:F7:14:AB (Static)

Details Instant Piconet | Summary | Instant Channels

Figure 130: Details View Example 1

Details

All fields | Show in overview | Display | Search

Name	Value
Link-Layer Packet	
Header	
PDU Type	ADV_IND
RFU	Reserved (0)
Channel Selection Algorithm	#1 (Legacy)
TxAdd	Random
RFU (RxAdd)	Reserved (0)
Payload Length	35
Advertiser Address	EF:61:B2:F7:14:AB (Static)
Advertising Data	
Flags	
Length	2
Data Type	Flags
LE Limited Discoverable Mode	No
LE General Discoverable Mode	Yes
BR/EDR Not Supported	Yes
Simultaneous LE and BR/EDR (Cont...)	No
Simultaneous LE and BR/EDR (Host)	No
Reserved	3 bits
Service Uuids (More available)	
Length	17
Data Type	Service Uuids (More available)
Uuid 1	EF680100-9B35-4933-9B10-52FFA9740042
Local Name	
Length	7
Data Type	Local Name
Complete Local Name	"Thingy"
Non-significant Part	0 bytes
CRC	Valid
Raw Content	
Raw Data	40 23 AB 14 F7 B2 61 EF 02 01 06 11 06 42 00 74 A9 FF 52

Details Instant Piconet | Summary | Instant Channels

Figure 131: Details View Example 2

Some of the most important details include:

- **Sniffer radio information:** RSSI - strength of the signal received by the sniffer for the selected packet. *-33.0 dBm in this case*
- **RF Channel:** which channel the packet was transmitted on. *Channel 37 in this case.*
- **Link layer information:** PHY, Coding Scheme, CRC Initial Seed.
- **Timing information:** Start Time, Duration, and Delta from previous.
- **Link Layer information:** PDU Type.
- **Advertising Data:** Flags, Service UUIDs, Local Name.

The tree view also shows any Scan Requests and Scan Responses exchanged between the Thingy:52 and other devices.

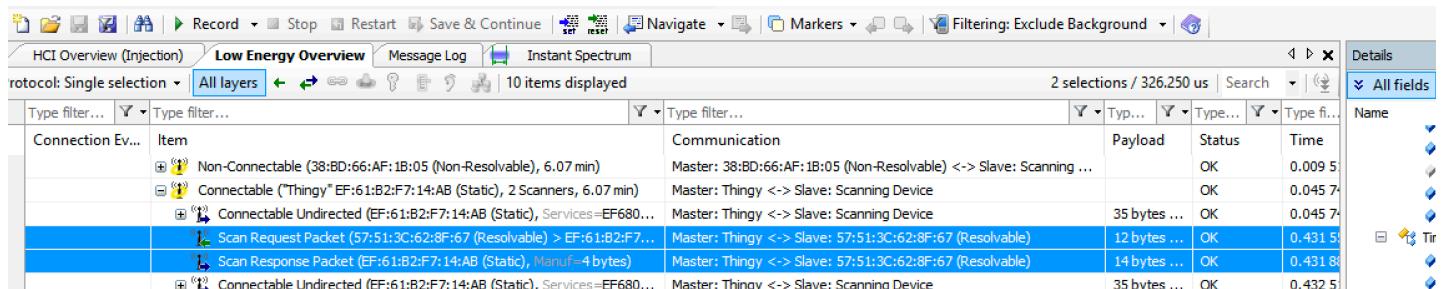


Figure 132: Scan Request and Response

We previously mentioned how a sniffer like the Ellisys Bluetooth Tracker can help in finding issues related to timing of packets. Here's an example of a timing issue where a Scan Response was sent too soon after the previous packet - *per the Bluetooth specification*. The sniffer software analyzes the timing, data integrity (CRC value), and other data points for the different packets and displays warnings when issues are discovered.

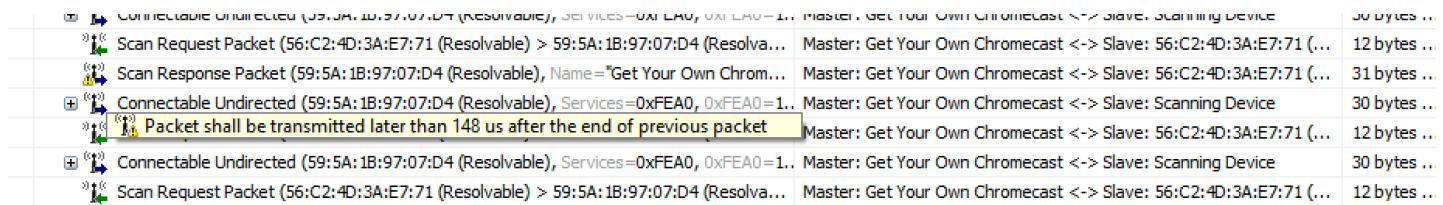


Figure 133: Incorrect Packet Timing

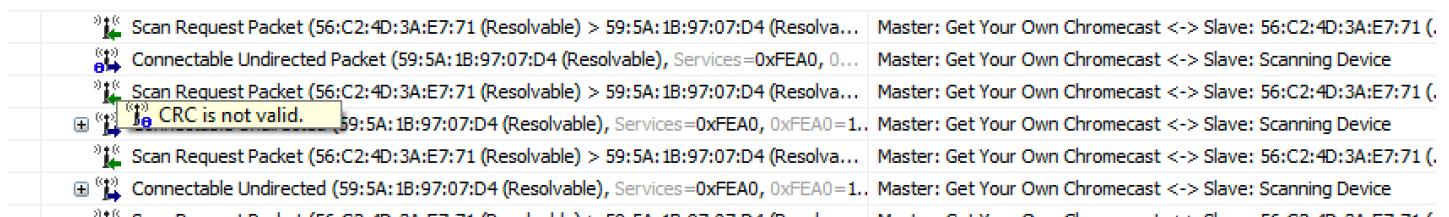


Figure 134: Invalid Packet CRC

You can get more details on the issue found with a specific packet by looking at the **Details** view:

Service UUID	0xFEAO
Service Data	13 bytes
Non-significant Part	0 bytes
CRC	Invalid, should be 0xEA0532
Raw Content	
Raw Data	40 1E D4 07 97 1B 5A 59 02 01 02 03 0

Figure 135: Invalid CRC Details View

Physical Channel	Advertisement (Get Your Own C...
Timing	
Start Time	11.061 886 375
Duration	328 us
Delta from Previous	323.13 us (0.5 slots)
TIFS	147.125 us
TIFS Violation	0.875 us too short
Devices	
Originator	Master

Figure 136: Incorrect Timing Details View

If you point to the top-level root of a group of Advertisements, you can see a summary of issues found:

Issues Summary Popup	Master: C8:69:CD:40:DE:94, 3 Scanners, 44.5 min	Master: 38:B
Connectable (C8:69:CD:40:DE:94, 3 Scanners, 44.5 min)	Master: C8:69:CD:40:DE:94, 3 Scanners, 44.5 min	Master: C8:6
Non-Connectable (38:BD:66:AF:1B:05 (Non-Resolvable), 44.5 min)	Non-Connectable (38:BD:66:AF:1B:05 (Non-Resolvable), 44.5 min)	Master: 38:B
Connectable ("Get Your Own Chromecast" 59:5A:1B:97:07:D4 (Resolvable), Scanner 56:C...)	Connectable ("Get Your Own Chromecast" 59:5A:1B:97:07:D4 (Resolvable), Scanner 56:C...)	Master: Get 1
Packet shall be transmitted earlier than 152 us after the end of previous packet	Packet shall be transmitted earlier than 152 us after the end of previous packet	Master: Get 1
Packet shall be transmitted later than 148 us after the end of previous packet	Packet shall be transmitted later than 148 us after the end of previous packet	Master: Get 1
Missing Field	Missing Field	Master: Get 1
CRC is not valid.	CRC is not valid.	Master: Get 1
Connectable Undirected (59:5A:1B:97:07:D4 (Resolvable), Services=0xFEAO, 0xFEAO=1..)	Connectable Undirected (59:5A:1B:97:07:D4 (Resolvable), Services=0xFEAO, 0xFEAO=1..)	Master: Get 1
Scan Response Packet (59:5A:1B:97:07:D4 (Resolvable), Name="Get Your Own Chrom...")	Scan Response Packet (59:5A:1B:97:07:D4 (Resolvable), Name="Get Your Own Chrom...")	Master: Get 1

Figure 137: Issues Summary Popup

Some other useful features include: colorizing of certain packet types, filtering of specific device packets, setting time

references, and others. To see a list of options available, right-click a certain packet:

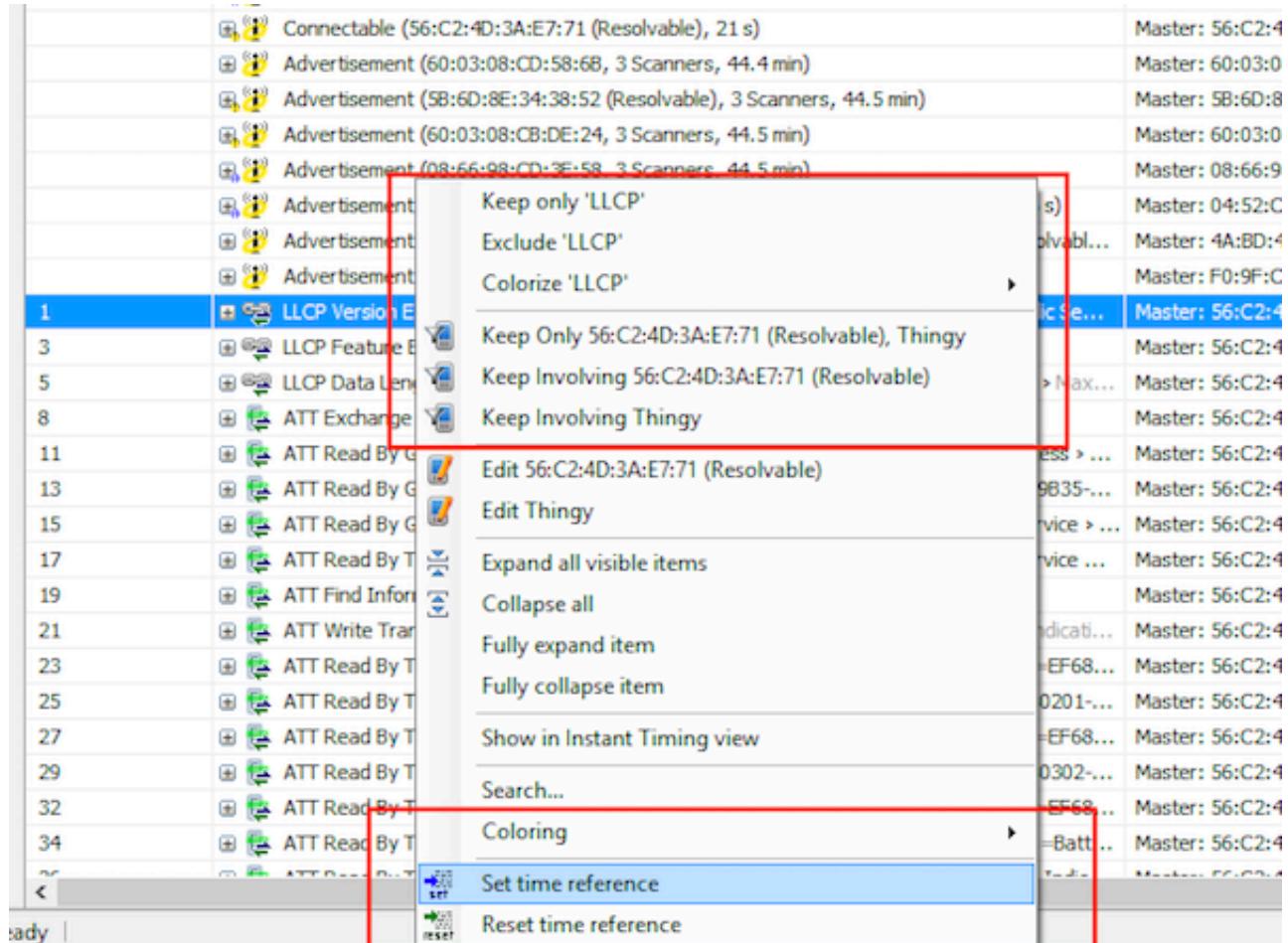


Figure 138: Right-Click Packet Options

Sniffing Connections

Once you have a capture started, you can initiate a connection between your Central and Peripheral devices. The sniffer will then be able to capture all data packets exchanged during a connection between the two devices.

Let's take a look at some of the connection-related packets. In this use case, we connected to the Thingy:52 using the Nordic nRF Connect mobile app:

- Connection Request Packet:

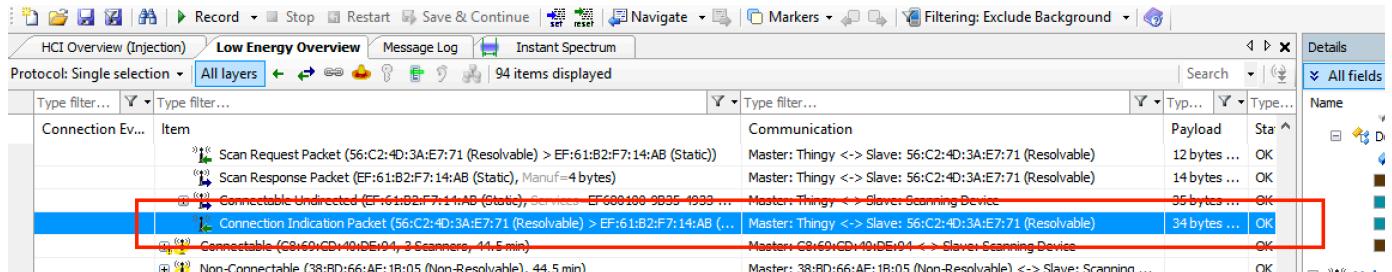


Figure 139: Connection Request Packet

- Version Exchange Packet:

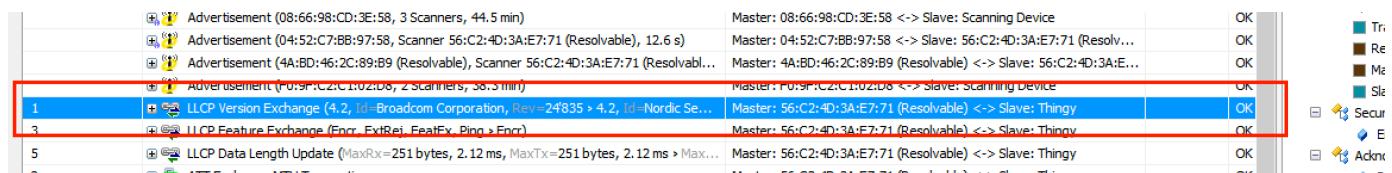


Figure 140: Version Exchange Packet

- Battery level Characteristic Read:

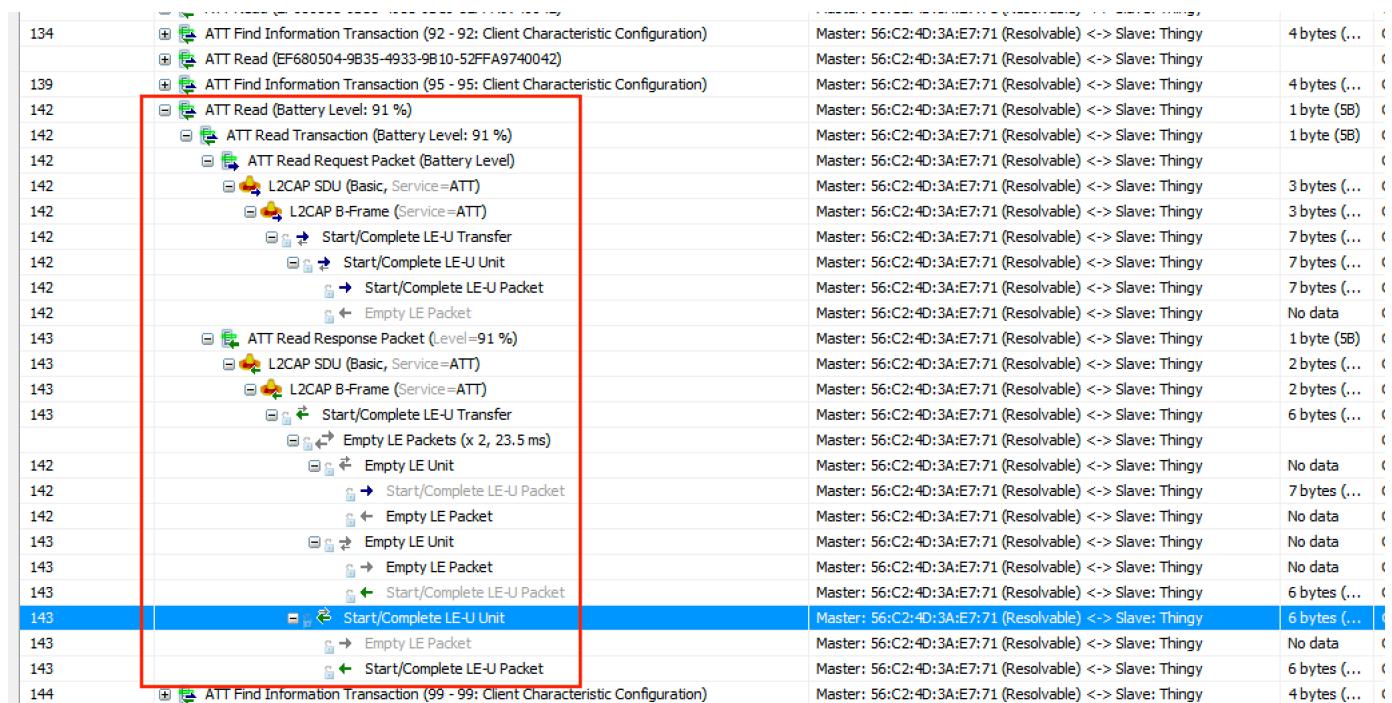


Figure 141: Battery Level Characteristic Value Read

For any given packet, you can view the full sequence of packets that completed the operation including the hierarchy of BLE packet layers by expanding the tree structure rooted at the packet. For example, the above screenshot of the Battery level Read shows all packets involved with this operation.

Reverse Engineering a Bluetooth Lightbulb

In our project, we will connect to, and control, the Bluetooth Playbulb candlelight. The device has a companion mobile app that allows the user to control and read information from the device.

In our project though, it's quite difficult to automate control using a mobile app, especially since we don't have access to the app's source code. Instead, we would like to connect to the device and automate some of its functionality such as turning it on or off from the Gateway device.

We are going to use this task as an exercise to understand how one can reverse engineer the operation of a Bluetooth Low Energy device. For this purpose, we would need to understand the GATT structure of the device, and how packets get sent to the device along with the specific values that map to the different operations we're interested in analyzing.

Required Hardware

In order to reverse engineer the lightbulb, we need the following:

- [The Playbulb Bluetooth candle](#).
- The companion mobile app (iOS or Android).
- A Bluetooth Low Energy sniffer.
- A computer to interface with the sniffer and run the sniffer software.

In our exercise, we will be using the Nordic nRF sniffer, but practically any BLE sniffer can be used to accomplish this task.

Steps to Reverse Engineer a Bluetooth Low Energy Device

There are a few steps required to take in order to reverse engineer a BLE device. First, let's go over the objectives of this exercise:

- Understand the GATT structure of the data exposed by the device and what each Attribute (Service, Characteristic) represents.
- Understand the format of the "commands" that can be used to control the device.
- Based on the understanding of the GATT and its components, we can now control the device from another Central device/app of our choice.

Note: Keep in mind that if the connection between the mobile app and the device is encrypted and/or utilizes security, then the task becomes much more difficult one (sniffing encrypted connections is sometimes possible, but is beyond the scope of this book). In our example, the Playbulb candle does not implement security for communication with the companion mobile app.

The basic steps we will take to reverse engineer our BLE lightbulb:

- Set up the Playbulb (out of the box).
- Download the companion mobile app.
- Run the mobile app and connect to the Playbulb.
- Connect the BLE sniffer hardware to a computer and run the sniffer software.
- Start the packet capture.
- Run through each of the commands and operations we're interested in while the sniffer capture session is running.
- Analyze the packet capture to understand the sequence of commands that achieve a certain task (such as turning on the Playbulb).
- The goal here is to "decipher" the packets being exchanged between the mobile device and the Playbulb – so that we can program our own remote control, for the Playbulb.

In our example, we are only interested in two simple operations: turning the Playbulb on, and turning it off. However, we'll be going through an expanded list of operations to make the exercise more engaging.

Reverse Engineering the Playbulb Candle

First, let's look at the mobile app to see what operations can be performed and are supported by the device. The main operations are:

- Choose color (including Color On/Off)
- Color/brightness intensity
- Shake On/Off
- Effects:
 - Flashing:
 - Speed: Slow <-> Fast
 - Color Switch: On/Off
 - Choose Color
 - Pulse:
 - Speed: Slow <-> Fast
 - Choose Color
 - Rainbow:
 - Speed: Slow <-> Fast
 - Rainbow Fade:
 - Speed: Slow <-> Fast
 - Candle:
 - Color Switch: On/Off
 - Choose Color

There are other operations related to a Timer and Security features, but for simplicity we won't be going into details on those.

Operations

Next, we go through the exercise of running the different operations and capturing the BLE packets associated with them using the nRF sniffer.

Note: One thing to keep in mind is that the nRF sniffer cycles between the three Primary Advertising Channels (37, 38, & 39), so it may take a couple of tries to capture a connection between the Playbulb and its companion app.

nRF Sniffer Setup

To make the task a little easier, we will apply a couple of filters to the capture:

- Once you start the capture, and before you connect from the mobile app, navigate to the **Devices** drop-down menu from the sniffer toolbar. When testing this, I've found that the sniffer does not display the device's name ("Playbulb Candle"), so instead I've placed the Playbulb in close proximity to the sniffer and chose the device that had the highest RSSI (-44 dBm in my case).

It turns out that the Playbulb is sending the **Device Name** in the Scan Response packet, which may explain why the sniffer did not display the name in the capture window and in the Devices list. Scan Responses are sent in response to a Scan Request. So if there are no Centrals currently searching for Peripherals and sending Scan Requests, the Scan Responses won't exist.

Interface		Device	Passkey / OOB key		
No.	Time	Source	Length	Channel	RSSI (dBm)
5794	74.744961	Master_0x50655bd6	26	12	-48
5795	74.745253	Slave_0x50655bd6	26	12	-44
5796	74.745732	Master_0x50655bd6	26	22	-48
5797	74.746391	Slave_0x50655bd6	26	22	-44
5798	74.846824	Master_0x50655bd6	26	32	-44
5799	74.847191	Slave_0x50655bd6	26	32	-44
5800	74.849266	Master_0x50655bd6	26	5	-45
5801	74.849724	Slave_0x50655bd6	26	5	-44
5802	74.850023	Master_0x50655bd6	26	15	-49
5803	74.850310	Slave_0x50655bd6	26	15	-44
5804	74.850587	Master_0x50655bd6	26	25	-48
5805	74.850860	Slave_0x50655bd6	26	25	-44
5806	74.953171	Master_0x50655bd6	26	35	-45
5807	74.953547	Slave_0x50655bd6	MASTER_0x50655bd6	LE_LL	-44

Figure 142: Device Name missing

- After selecting the Playbulb, the capture will filter out Advertisements from other devices. You can verify this by selecting one of the SCAN_RSP packets and looking at the details of that packet:

67344 1325.973...	c7:15:4b:0f:ac:e6 Broadcast	LE LL
67345 1326.073...	c7:15:4b:0f:ac:e6 Broadcast	LE LL
67346 1326.176...	c7:15:4b:0f:ac:e6 Broadcast	LE LL

..... .0... = Encrypted: No
..... ..0. = Direction: Slave -> Master
..... ...1 = CRC: OK

Channel: 38
RSSI (dBm): -44
Event counter: 0
Delta time (µs end to start): 1117
[Delta time (µs start to start): 1565]

▼ Bluetooth Low Energy Link Layer

Access Address: 0x8e89bed6

▼ Packet Header: 0x1704 (PDU Type: SCAN_RSP, TxAdd: Public)

.... 0100 = PDU Type: SCAN_RSP (0x4)
..00 = RFU: 0
.0... = Tx Address: Public
0... = Reserved: False
..01 0111 = Length: 23
00.. = RFU: 0

Advertising Address: c7:15:4b:0f:ac:e6 (c7:15:4b:0f:ac:e6)

▼ Scan Response Data: 1009506c617962756c622043616e646c65

▼ Advertising Data

▼ Device Name: Playbulb Candle

Length: 16
Type: Device Name (0x09)
Device Name: Playbulb Candle

CRC: 0x41840b

Figure 143: Playbulb Candle Device Name

- Next, we attempt to connect to the Playbulb from the mobile app, and make sure we are capturing Connection Data Packets. You should start seeing packets labeled LE LL in the **Protocol** column.
- Now, you'll see there are many **EMPTY PDU** packets. To filter those out and show only meaningful packets, we

can apply the following filter:

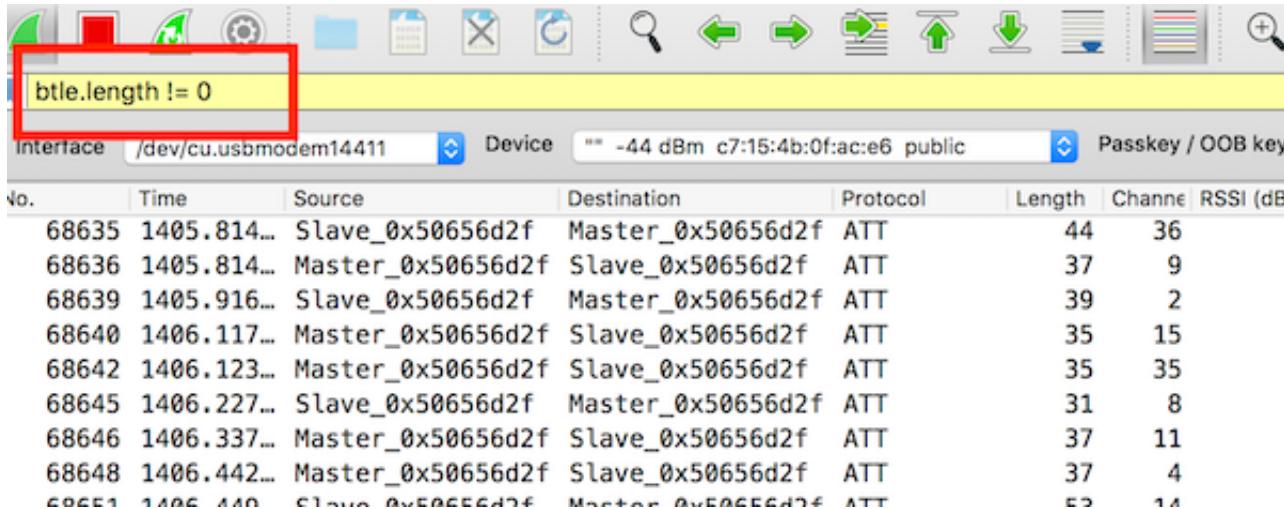


Figure 144: Filtering out Empty Packets

- The next step is to perform the commands we're interested in from the mobile app. Let's start with setting the different colors: White

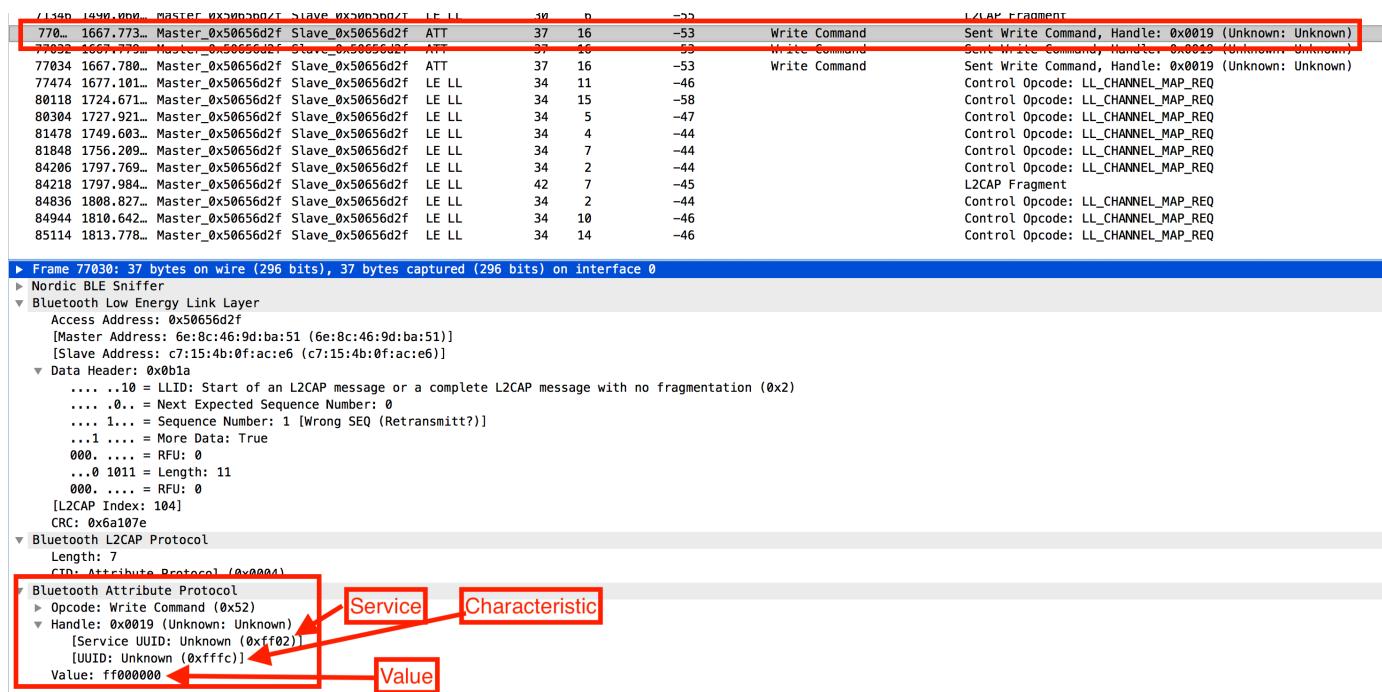


Figure 145: White Color Command

- Now, that we've detected one of the write commands, we can add another column to the capture window to make it easier to see the Write values. Select the Value field in the Details window under Bluetooth Attribute Protocol.

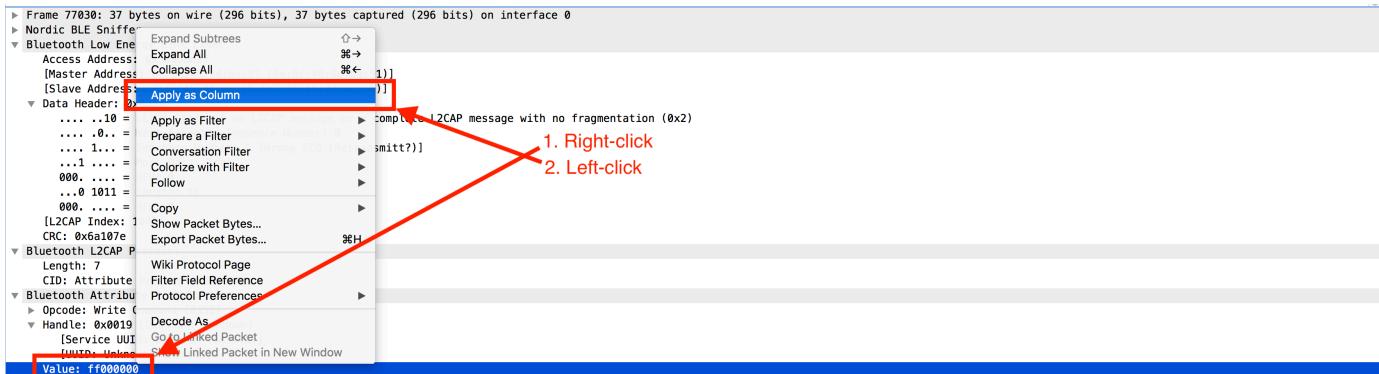


Figure 146: Add Value as Column

- Repeat the operation for setting the remaining colors: Red, Green, and Blue.

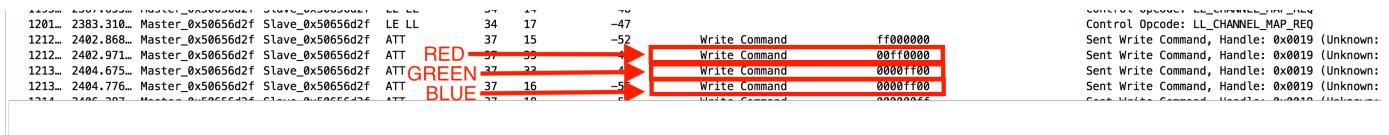


Figure 147: Red, Green, and Blue commands

By analyzing the captured packets, we conclude the following:

- The Characteristic associated with changing and setting the color is the **0xFFFFC** Characteristic under the **0xFF02** Service.
- The operation used for Writing to the Characteristic is the **Write Command**, which is also called a **Write Without Response** - this is an operation that does not require a Response packet from the peer device.
- The value for the Characteristic is in the following format: **0x[WW] 0x[RR] 0x[GG] 0x[BB]**, where:
 - WW is the White level
 - RR is the Red level
 - GG is the Green level
 - BB is the Blue level
- For example, when the color is set to **Red**, the value will be set to **0x00 0xFF 0x00 0x00**. For **Blue**, the value will be set to **0x00 0x00 0x00 0xFF**, and so on.

Color Brightness/Intensity

Using the brightness slider (OFF <----> ON), we can adjust the brightness/intensity level of the current color of the lightbulb. To find out how these values are getting set on the lightbulb, we run the capture during this operation. Sliding the scale from right (ON) to left (OFF), we see the following values being written:

1235..	53.099091	Master_0xaf9ab3ae	Slave_0xaf9ab3ae	ATT	37	11	-50	Write Command	ff000000	Sent Write Command, Handle: 0x0019 (Unknown: Un
1236..	55.595904	Master_0xaf9ab3ae	Slave_0xaf9ab3ae	ATT	37	20	-54	Write Command	ed000000	Sent Write Command, Handle: 0x0019 (Unknown: Un
1236..	55.596388	Master_0xaf9ab3ae	Slave_0xaf9ab3ae	ATT	37	20	-54	Write Command	ed000000	Sent Write Command, Handle: 0x0019 (Unknown: Un
1236..	55.596827	Master_0xaf9ab3ae	Slave_0xaf9ab3ae	ATT	37	20	-54	Write Command	ed000000	Sent Write Command, Handle: 0x0019 (Unknown: Un
1236..	55.597252	Master_0xaf9ab3ae	Slave_0xaf9ab3ae	ATT	37	35	-50	Write Command	ed000000	Sent Write Command, Handle: 0x0019 (Unknown: Un
1236..	55.597672	Master_0xaf9ab3ae	Slave_0xaf9ab3ae	ATT	37	35	-50	Write Command	e3000000	Sent Write Command, Handle: 0x0019 (Unknown: Un
1237..	55.700407	Master_0xaf9ab3ae	Slave_0xaf9ab3ae	ATT	37	13	-52	Write Command	dc000000	Sent Write Command, Handle: 0x0019 (Unknown: Un
1237..	55.707831	Master_0xaf9ab3ae	Slave_0xaf9ab3ae	ATT	37	33	-49	Write Command	d3000000	Sent Write Command, Handle: 0x0019 (Unknown: Un
1237..	55.707655	Master_0xaf9ab3ae	Slave_0xaf9ab3ae	ATT	37	33	-48	Write Command	c9000000	Sent Write Command, Handle: 0x0019 (Unknown: Un
1237..	55.911302	Master_0xaf9ab3ae	Slave_0xaf9ab3ae	ATT	37	7	-50	Write Command	b9000000	Sent Write Command, Handle: 0x0019 (Unknown: Un
1237..	55.917901	Master_0xaf9ab3ae	Slave_0xaf9ab3ae	ATT	37	7	-50	Write Command	a5000000	Sent Write Command, Handle: 0x0019 (Unknown: Un
1237..	55.918843	Master_0xaf9ab3ae	Slave_0xaf9ab3ae	ATT	37	22	-51	Write Command	a5000000	Sent Write Command, Handle: 0x0019 (Unknown: Un
1237..	55.919630	Master_0xaf9ab3ae	Slave_0xaf9ab3ae	ATT	37	22	-50	Write Command	93000000	Sent Write Command, Handle: 0x0019 (Unknown: Un
1237..	55.920207	Master_0xaf9ab3ae	Slave_0xaf9ab3ae	ATT	37	0	-52	Write Command	93000000	Sent Write Command, Handle: 0x0019 (Unknown: Un
1237..	55.920988	Master_0xaf9ab3ae	Slave_0xaf9ab3ae	ATT	37	0	-52	Write Command	7e000000	Sent Write Command, Handle: 0x0019 (Unknown: Un
1237..	55.921492	Master_0xaf9ab3ae	Slave_0xaf9ab3ae	ATT	37	15	-49	Write Command	7e000000	Sent Write Command, Handle: 0x0019 (Unknown: Un
1237..	55.922116	Master_0xaf9ab3ae	Slave_0xaf9ab3ae	ATT	37	15	-49	Write Command	6c000000	Sent Write Command, Handle: 0x0019 (Unknown: Un
1237..	55.922825	Master_0xaf9ab3ae	Slave_0xaf9ab3ae	ATT	37	15	-49	Write Command	5c000000	Sent Write Command, Handle: 0x0019 (Unknown: Un
1237..	55.923458	Master_0xaf9ab3ae	Slave_0xaf9ab3ae	ATT	37	15	-49	Write Command	4d000000	Sent Write Command, Handle: 0x0019 (Unknown: Un
1237..	56.024295	Master_0xaf9ab3ae	Slave_0xaf9ab3ae	ATT	37	30	-48	Write Command	4d000000	Sent Write Command, Handle: 0x0019 (Unknown: Un
1237..	56.027127	Master_0xaf9ab3ae	Slave_0xaf9ab3ae	ATT	37	30	-49	Write Command	3e000000	Sent Write Command, Handle: 0x0019 (Unknown: Un
1237..	56.029251	Master_0xaf9ab3ae	Slave_0xaf9ab3ae	ATT	37	23	-51	Write Command	3e000000	Sent Write Command, Handle: 0x0019 (Unknown: Un
1237..	56.030935	Master_0xaf9ab3ae	Slave_0xaf9ab3ae	ATT	37	23	-51	Write Command	30000000	Sent Write Command, Handle: 0x0019 (Unknown: Un
1237..	56.032500	Master_0xaf9ab3ae	Slave_0xaf9ab3ae	ATT	37	23	-51	Write Command	23000000	Sent Write Command, Handle: 0x0019 (Unknown: Un
1237..	56.034028	Master_0xaf9ab3ae	Slave_0xaf9ab3ae	ATT	37	23	-51	Write Command	14000000	Sent Write Command, Handle: 0x0019 (Unknown: Un
1237..	56.035349	Master_0xaf9ab3ae	Slave_0xaf9ab3ae	ATT	37	23	-51	Write Command	08000000	Sent Write Command, Handle: 0x0019 (Unknown: Un
1237..	56.036468	Master_0xaf9ab3ae	Slave_0xaf9ab3ae	ATT	37	23	-51	Write Command	00000000	Sent Write Command, Handle: 0x0019 (Unknown: Un

Figure 148: Sliding Scale Write Values

We notice that the values range from a high value with a maximum of 0xFF for the specific color value (corresponding to the highest brightness level moving towards the "ON" label), and then a low value with a minimum of 0x00 for the same color bytes. In our example, we have chosen **White** as the color, and we are seeing a value of **0xFF 0x00 0x00 0x00** being written. When the slider is in the middle position, we see a value of **0x80 0x00 0x00 0x00** being written to the Characteristic. Finally, when the slider is at the "OFF" position, the value being written becomes **0x00 0x00 0x00 0x00**. The same process applies to each color in its dedicated value position.

Testing

Let's run a simple experiment to check if the results are accurate. We'd like to see if we can set the colors of the light, as well as switch it off, without using the official Playbulb mobile app. We will:

- Connect to the Playbulb candle light via the Nordic nRF Connect Application:

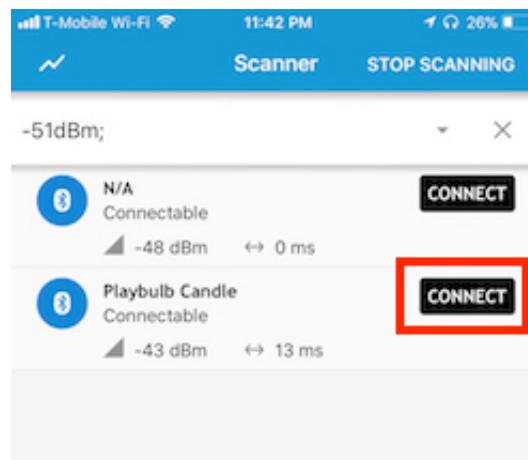
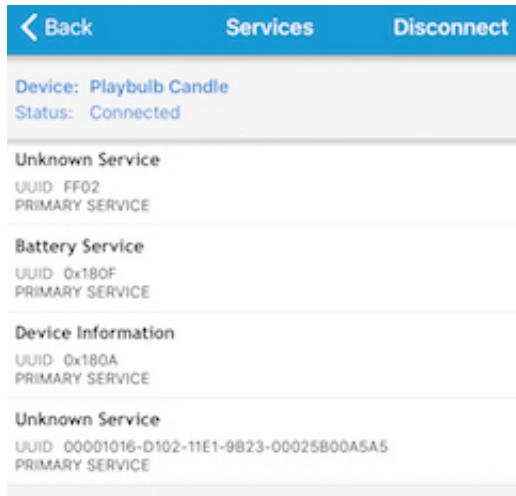
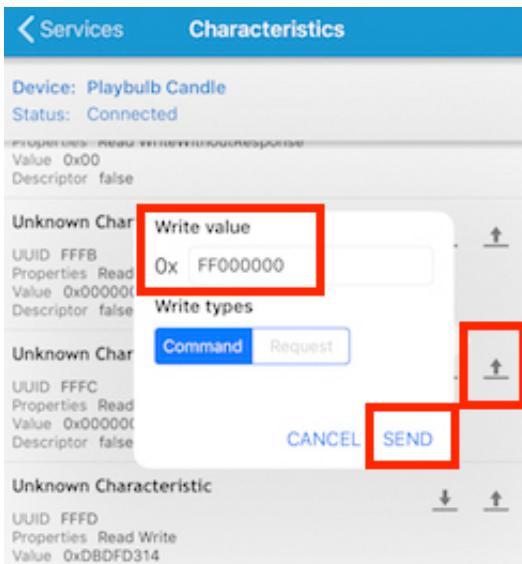


Figure 149: Connecting to the Playbulb Candle

*Figure 150: Playbulb GATT*

- Look for the **0xFFFFC** Characteristic under the **0xFF02** Service.
- Click the **Write** action (one arrow pointing up)
- Enter the value **FF000000**.
- Hit **SEND**.

*Figure 151: Changing Color to White*

- Notice that the light turns to **White**.
- Now test if we can change the color to **Red**.

- Write the value **00FF0000** to the same Characteristic **0xFFFFC**.



Figure 152: Changing Color to Red

- Notice that the light turns to **Red**.
- Finally, let's test turning off the light.
- Write the value **00000000** to the same Characteristic **0xFFFFC**.
- Notice that the light has turned off!

HCI Logging

An alternative to using a Bluetooth sniffer for over-the-air capture is to perform HCI Logging on the mobile device running the companion mobile app.

As we've talked about before, HCI is the standardized interface between the upper levels of BLE (on the Host) and the lower levels (on the Controller), hence the name Host Controller Interface (HCI). On Android, it is relatively simple to capture the log of HCI Commands, Events, and messages between the Host and the Controller. This provides a powerful method to reverse engineer BLE mobile apps and a useful method for debugging issues as well.

Android HCI Logs

The Android Operating System provides a method to capture the HCI Log. Once you've captured the HCI log file, you can transfer the file off the device to a computer and import it to another program that can parse the log file.

The steps to capture the HCI log in Android are:

- First, you'll need to enable "Developer options". You'll only have to do this once. Enable it by going to Settings --> About Phone. Then navigate down to the Build Number, and tap it continuously till you get a message indicating that you're now a developer.
- Navigate to Developer options, and make sure the "**Enable Bluetooth HCI snoop log**" is turned ON. This option will enable saving all HCI Commands and Events to a log file on the device, which you can then transfer to your PC

to import into a program of choice for analysis.

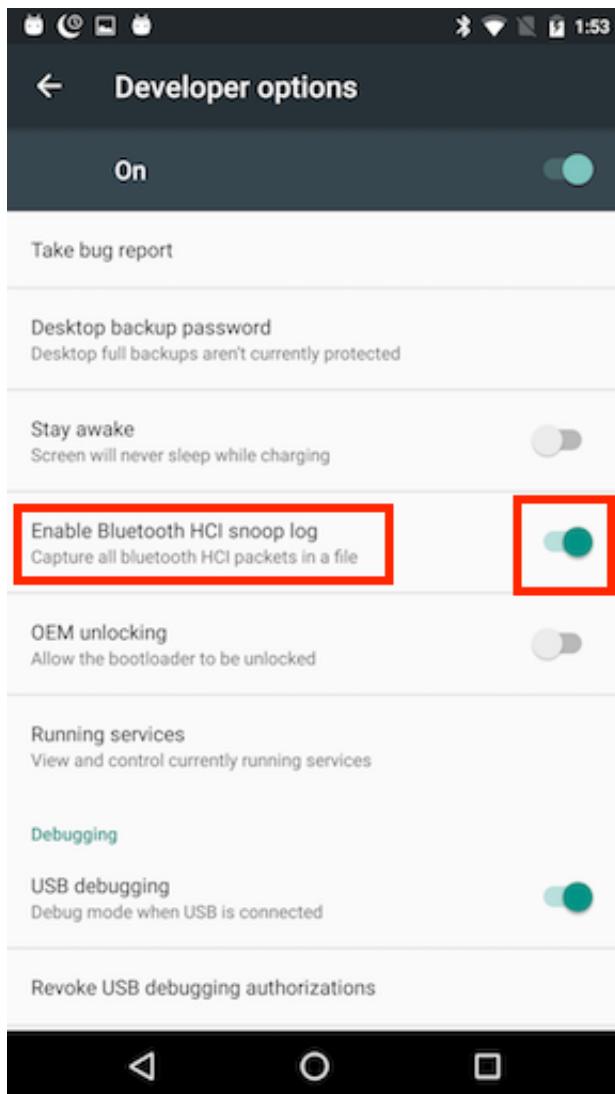


Figure 153: Enable Bluetooth HCI Snoop Log

- Run the Playbulb companion app on your Android phone.
- Connect to the Playbulb candlelight via the application.
- Perform the different functions and operations you're interested in capturing the HCI log for.
- Make sure the phone's USB connection is set up as a file transfer (MTP) connection.
 - To enable this, go to your phone's **Settings -> Developer options**.
 - Under **Networking**, you will find an option labeled "**Select USB Configuration**".

- Tap it and select “MTP (Media Transfer Protocol)”.

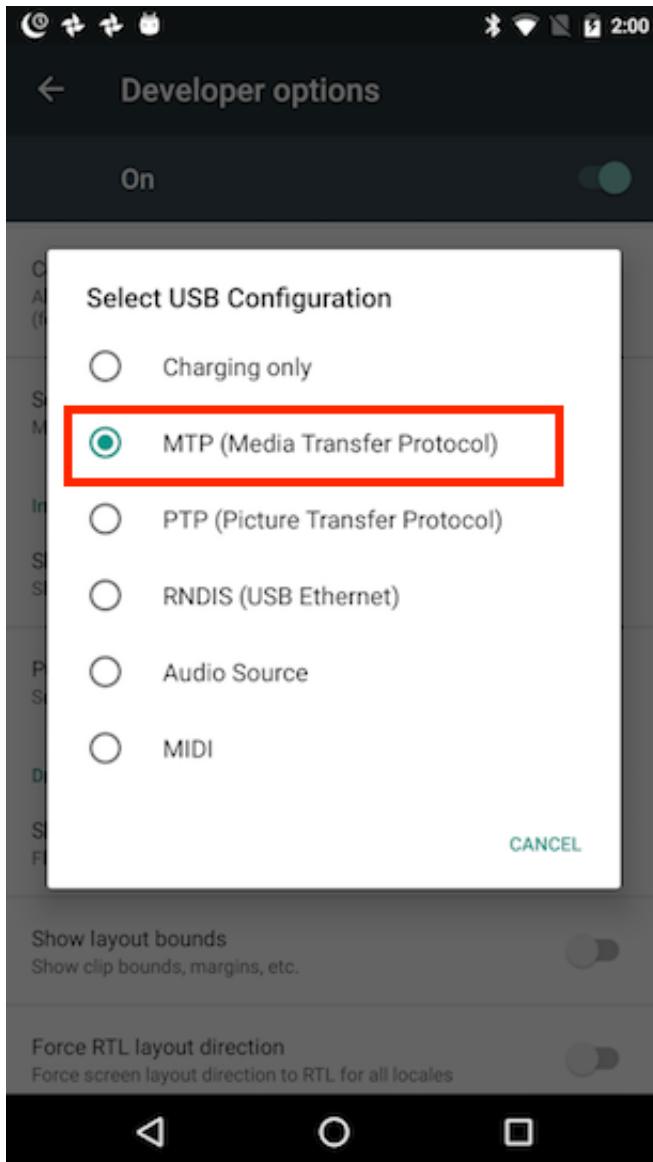


Figure 154: Enable MTP

- Connect your Android phone to your PC via a USB cable.
If you're on a Mac you'll want to use a program called "Android File Transfer" which can be downloaded [here](#).
- If you face issues accessing the Android file system from your computer, you may have to re-enable the **MTP** USB setting again.
- Now navigate to the Android filesystem and look for the file named `btsnoop_hci.log`.
- Drag the file and copy it somewhere on your PC.

- Now you have a few options for viewing this file.
 - Use a Bluetooth sniffer software such as the Ellisys Bluetooth Analyzer software.
 - Use Wireshark to view the file.
 - Use a parser/library for HCI Log files (e.g. [Bluetooth HCI decoder library](#)).
- Using a software application such as Wireshark or your sniffer's software is the preferred way since it usually gives you more flexibility in viewing and analyzing the data.
- Using Wireshark:
 - Navigate to File -> Open
 - Locate the file and open it.

Here's a screenshot of the file open in Wireshark and showing the captured HCI Commands for changing the Playbulb light color, and finally turning it off (via the slider).

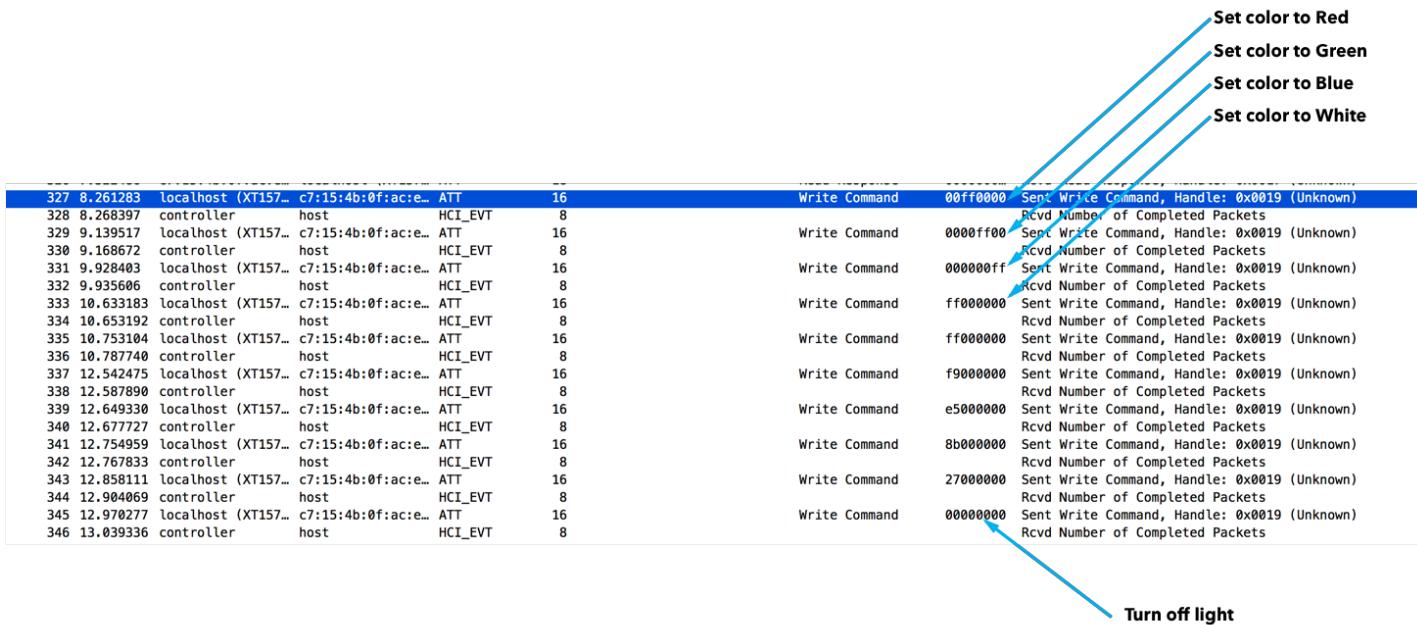


Figure 155: HCI Command Capture in Wireshark

Bluetooth Sniffer Versus HCI Logging

There are different pros and cons for each approach:

- An advantage of HCI Logging is that you're guaranteed to not miss any Commands and Events.
- A downside of HCI Logging, since it is captured offline, is that it's hard to match the logged HCI Commands/Events to the operations you performed manually through the mobile app.

- Over-the-air capture using a Bluetooth sniffer on the other hand, allows you to see the Commands in real-time, hence making it easier to match the Commands to the packets captured by the sniffer.
- A downside of using a sniffer is that you may miss some packets depending on the environment and the positioning of the devices.

How to Choose a BLE Module/Chipset

Module Versus Chipset

It's important to first understand the difference between a chipset and a module. A chipset contains the Microcontroller Unit (MCU) as well as the RF section and the software stack.

A module, on the other hand, is a stand-alone packaging of a chipset and its supporting components, on a Printed Circuit Board (PCB). The goal of the module is to provide a "ready-to-integrate" package of the chipset, RF, as well as an antenna (usually). The benefits of using a module include:

- Pre-certification of the module which can cut a lot of high costs associated with the testing and the certification process.
- Pre-designed RF section. The module usually takes into account designing for maximum performance in the antenna design.
- Time savings compared to designing your circuit with a "raw" chipset. (*Certifications and testing can be a time-consuming task and cause significant delays in getting your product to market*)

Keep in mind that module may cost an order of magnitude more than its packaged chipset, but that cost is usually offset in the cost-savings that come with a prepackaged module versus a separate chipset (especially when producing at high volumes).

It may be obvious now, but for any given chipset you'll probably find multiple options available for modules that integrate the chipset. However, depending on the physical size requirements and other product design requirements, choosing a chipset may prove to be the way to go for a new design.

Considerations for Choosing a BLE Chipset

Let's focus on choosing a chipset for a BLE product and consider the different aspects that affect the decision. *A module is most likely based on a specific chipset, so choosing the chipset first only makes sense.*

Some of the factors to consider include:

- **Data rate:** this could depend on the version of Bluetooth supported and the stack, but it also depends on the design of the chipset itself. You will generally find information regarding data rate in the chipset's data sheet.
- **Range:** range is affected by the maximum transmit power the chipset can support, but it also depends on the supported version of Bluetooth as well as the chipset design itself (e.g. long-range mode in Bluetooth 5). You will generally find information regarding range in the chipset's data sheet.
- **Package type & physical size:** chipsets come in different sizes and different package types, and it varies depending on the vendor. If your design has some restrictions in terms of physical size, make sure you keep that

into consideration when evaluating the different chipsets.

- **Bluetooth version supported:** There is a lot of confusion in the marketplace when it comes to the different versions of Bluetooth. Bluetooth versions (so far) including 4.0, 4.1, 4.2 and 5.0 are all backwards compatible in the general, meaning that devices with different versions can still communicate with each other. However, this communication may be limited in some aspects. For example, if you want to utilize Bluetooth 5's new long-range feature, then both devices must support Bluetooth 5 and specifically the long-range feature. Another point of confusion is that a Bluetooth 5-compliant device does not necessarily support all of the Bluetooth 5 features. Some of these features (including the long-range and higher speed features) are optional and do not affect compliance with the Bluetooth 5 spec.
- **Power consumption:** power consumption is probably one of the most important factors to consider. It may even have been the main reason you are considering Bluetooth Low Energy as a wireless technology in the first place. Datasheets for different chipsets usually provide some power consumption numbers such as sleep current, current draw at full transmit power,...etc. However, this only paints part of the picture. These figures can be a starting point, but the only way to really find out the power consumption of your product is by testing under the typical application environment and in a realistic testing environment that mimics the end-user's environment.
- **Chipset vendor support and reputation:** when considering a chipset from a vendor, you have to weigh in the level and quality of support the vendor provides to your hardware and software engineers. This can be critical to the success and release of your product, especially if you run into problems while developing it. Different vendors will have varying levels of turn-around times for tickets and support questions.
- **Application & environment:** depending on the application and use case of your product, specific chipsets may be more suitable than others. The expected end user environment also affects which chipsets are appropriate. Some examples of such variables include: operating temperature range, humidity, outdoor vs. indoor, as well as other related factors.
- **End-of-life:** every chipset has an end-of-life where the chipset will stop being produced and support may not be available anymore. This is an important aspect to consider and bring up with any vendor you're considering. Ask about end-of-life expectancy and any near-term issues they may have with the specific chipset in question. If volumes are big enough, you can usually conduct deals with vendors to maintain production for a set period of time.
- **Pricing & volume:** pricing usually goes down and the purchased quantity rises. Keep that in mind when speaking to vendors and distributors about pricing. Usually they will set lower prices if they have commitments of higher annual chipset volumes.
- **Interface, I/O peripherals and pin count:** depending on the application and your product's requirements and features, you may find that some chipsets do not satisfy your needs. Also keep in mind that the existence of extra features and pin count of a chipset affect physical size and power consumption, so a more powerful processor is not always better.
- **Software and tools:** last, but not least, take into consideration the software package and tools supported by the chipset vendor for software/firmware development. Some vendors support open-source software tools and compilers, possibly bringing the development cost down significantly. Of course, there are pros and cons for each path. Consider your team's experience and level of familiarity with the tools supported by the chipset. Another aspect is the ability to customize the development tools for the project to support automated testing which can help make your product much more robust, stable, and secure. Software tools that provide a command

line interface (CLI) are easier to work with and integrate into existing build and test systems.

- **Host-less vs. Hosted mode:** some chipsets can operate and run the full application and the BLE stack together, while others may require a separate host to run the main application while the chipset runs the BLE stack only.
- **Stage of Product Development:** This is last, but could easily be first. If you are new to BLE, you may have a lot to learn. Choose chipsets that are well-supported, easy to develop-for, and easy to place in prototypes. Later, when cost-per-unit is more important (>100 quantity) a different chipset may be warranted.

A List of BLE Chipsets & Modules

Chipsets (A-Z list)

- CSR/Qualcomm
 - [CSR1010](#)
 - [CSR1011](#)
 - [CSR1012](#)
 - [CSR1013](#)
 - [CSR1020](#)
 - [CSR1021](#)
 - [CSR1024](#)
 - [CSR8311](#)
 - [CSR8350](#)
 - [CSR8510](#)
 - [CSR8811](#)
 - [CSRB5341](#)
 - [CSRB5342](#)
 - [CSRB5348](#)
 - [QCA4020](#)
 - [QCA4024](#)
- Dialog Semiconductor
 - [DA14580](#)
 - [DA14581](#)
 - [DA14582](#)
 - [DA14583](#)
 - [DA14585](#)
 - [DA14586](#)
 - [DA14680](#)
- Microchip
 - [IS1678](#)

- [IS1870](#)
- [IS1871](#)
- [IS2062](#)
- [IS2063](#)
- [IS2064](#)
- **Nordic Semiconductor**
 - [nRF51422](#)
 - [nRF51822](#)
 - [nRF51824](#)
 - [nRF51422](#)
 - [nRF52810](#)
 - [nRF52832](#)
 - [nRF52840](#)
- **NXP/Freescale**
 - [KW41Z](#)
 - [KW31Z](#)
 - [KW30Z](#)
 - [QN9020](#)
 - [QN9021](#)
 - [QN9022](#)
 - [QN9080](#)
- **Silicon Labs**
 - [EFR32 chipsets](#)
- **Texas Instruments**
 - [CC2540](#)
 - [CC2541](#)
 - [CC2564](#)
 - [CC2640](#)
 - [CC2642R](#)
 - [CC2652R](#)

Modules (A-Z list)

There are way too many modules to fully list here, so following are links to a few of the most popular BLE modules available in the market:

- [Nordic nRF-based modules list](#)
- NXP/Freescale modules
 - [Kemsys KT1010](#)

- [Rigado R41Z](#)
- Silicon Labs-based modules
 - [BGM13P](#)
 - [BGM11S](#)
 - [BGM111](#)
 - [BGM121](#)
 - [BGM113](#)
 - [BGM123](#)
 - [BLE112](#)
 - [BLED112 Dongle](#)
 - [BLE113](#)
 - [BLE121LR](#)
- [Texas Instruments modules list](#)
- [Texas Instruments 3rd party modules list](#)

Navigating the Bluetooth Specification Document

The Bluetooth specification document serves as the *ultimate reference* for any developer or tester working on a Bluetooth product. With that said, navigating the 2,800+ page document can be quite daunting!

The Bluetooth specification document can be downloaded from this website:

<https://www.bluetooth.com/specifications/bluetooth-core-specification>

In this section, we'll go over some tips that can help you navigate the document as well as other important resources for other useful information. The main document that contains most of the specification details is the **Core** document. The **Core Specification Supplement** document covers the different Advertisement Data Types (e.g., Local Name, Tx Power, Advertisement Flags, and others).

Here are some important notes about the Core Bluetooth specification document:

- The Bluetooth specification document covers both **Bluetooth Low Energy** and **Bluetooth Classic**.
- Bluetooth Low Energy is sometimes referred to as simply **LE** or **Low Energy** in the document.
- Bluetooth Classic is referred to as **BR/EDR**.
- Some of the most important sections that relate to Bluetooth Low Energy include:
 - Volume 1, Part A, Section 1.2 (Overview of Bluetooth Low Energy Operation)
 - Volume 1, Part A, Section 4.1.2 (LE Topology)
 - Volume 3 (Core System Package [Host Volume])
 - Volume 6 (Core System Package [Low Energy Controller Volume])

Some other useful resources (from the Bluetooth.com website) include:

- [Adopted GATT Profile and Service specifications](#)
- [Adopted GATT Services](#)
- [Adopted GATT Characteristics](#)

The Bluetooth Certification Process

There are two phases that your product needs to go through in order to get it certified by the Bluetooth SIG and give you permission to use the Bluetooth® trademark. They are **Qualification** and **Declaration**:

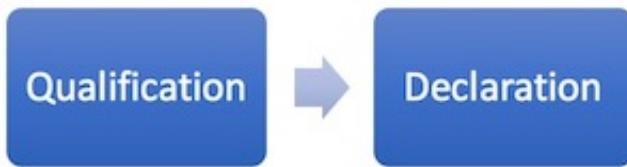


Figure 156: Main Steps in Bluetooth Certification

There is no need to Qualify if you are using an existing module or stack (that's already Qualified) and you are not making any modifications or additions. If you are designing your own Bluetooth device or making any modifications to an existing (Qualified) design, then Qualification is mandatory.

The details of this process is beyond the scope of this book, but here you will find a few of useful resources you can refer to. You can also contact the Bluetooth SIG if you have any questions or concerns.

Useful Resources:

- [Bluetooth Product Qualification & Declaration webpage](#)
- [How to Certify your Bluetooth Product](#)
(Texas Instruments wikipage)

Remote Control Source Code Walkthrough

In this chapter we'll be going through the full source code for Remote Control device within the main project. The Remote Control device (as defined in the **Main Project** chapter) is implemented on an nRF52840 development kit. We'll cover the most important aspects of the project-specific functionality.

Make sure you refer to the chapter titled "**Main Project**" before reading this chapter. It is important to understand the overall architecture and design of the Main Project before digging into the source code for the Remote Control device.

Important Note: The source code listed here is subject to change and may differ from the source code in the GitHub repository. Updates may be added to the code on GitHub to fix various bugs and add new features. The purpose of this "source code walkthrough" exercise is to guide the reader through the most important parts of the implementation for the nRF52 series platform.

Source Code Walkthrough

button_service.h

This file serves as the header file for the Button Service that exposes the ON and OFF button presses on the Remote Control device.

- A conditional include for the header file:

```
#ifndef BUTTON_SERVICE_H
#define BUTTON_SERVICE_H
```

- Includes for standard and nRF specific header files:

```
#include <stdint.h>
#include "boards.h"
#include "ble.h"
#include "ble_srv_common.h"
```

- Next, we define the macro for instantiating the Button Service:

```
#define BLE_BUTTON_SERVICE_OBSERVER_PRIO 2
#define BLE_BUTTON_SERVICE_DEF(_name) \
```

```
static ble_button_service_t _name;                                \
NRF_SDH_BLE_OBSERVER(_name ## _obs,                                \
                      BLE_BUTTON_SERVICE_OBSERVER_PRIO,      \
                      ble_button_service_on_ble_evt,        \
                      &_name)
```

- Here, we define the UUIDs for the Service and its Characteristics (refer to the **Generic Attribute Profile (GATT)** chapter for more information on how these UUIDs were derived):

```
// Button service:           E54B0001-67F5-479E-8711-B3B99198CE6C
// ON Button press characteristic: E54B0002-67F5-479E-8711-B3B99198CE6C
// OFF Button press characteristic: E54B0003-67F5-479E-8711-B3B99198CE6C

// The bytes are stored in little-endian format, meaning the
// Least Significant Byte is stored first
// (reversed from the order they're displayed as)

// Base UUID: E54B0000-67F5-479E-8711-B3B99198CE6C
#define BLE_UUID_BUTTON_SERVICE_BASE_UUID {0x6C, 0xCE, 0x98, 0x91, 0xB9, 0xB3, 0x11, 0x87, 0x9E, 0x47,
                                         0xF5, 0x67, 0x00, 0x00, 0x4B, 0xE5}

// Service & characteristics UUIDs
#define BLE_UUID_BUTTON_SERVICE_UUID      0x0001
#define BLE_UUID_BUTTON_ON_PRESS_CHAR_UUID 0x0002
#define BLE_UUID_BUTTON_OFF_PRESS_CHAR_UUID 0x0003
```

- We “forward-declare” the Button Service data structure so we can use it within other data structures ahead of its definition:

```
// Forward declaration of the custom_service_t type.
typedef struct ble_button_service_s ble_button_service_t;
```

- Next, we define an enumeration list of the different events such as ON button press Notifications enabled/disabled, and OFF button press Notifications enabled/disabled. These values get assigned locally in the Button Service module and get passed up to the application layer to handle these events:

```
/**@brief Button Service event type.*/
typedef enum
{
    BLE_BUTTON_ON_EVT_NOTIFICATION_ENABLED,    /**< Button ON press value notification enabled event. */
    BLE_BUTTON_ON_EVT_NOTIFICATION_DISABLED,   /**< Button ON press value notification disabled event. */
    BLE_BUTTON_OFF_EVT_NOTIFICATION_ENABLED,   /**< Button OFF press value notification enabled event. */
    BLE_BUTTON_OFF_EVT_NOTIFICATION_DISABLED  /**< Button OFF press value notification disabled event.
} ble_button_evt_type_t;
```

- Data structure to hold the Button Service event:

```
/**@brief Button Service event. */
typedef struct
{
    ble_button_evt_type_t evt_type;          /**< Type of event. */
    uint16_t              conn_handle;        /**< Connection handle. */
} ble_button_evt_t;
```

- Button service event handler function definition:

```
/**@brief Button Service event handler type. */
typedef void (*ble_button_evt_handler_t) (ble_button_service_t * p_button_service, ble_button_evt_t * p_evt);
```

- This is the main Button Service data structure that is instantiated to hold the different parameters related to this Service:

```
/**@brief Button Service structure.
 *      This contains various status information
 *      for the service.
 */
typedef struct ble_button_service_s
{
    uint16_t              conn_handle;
    uint16_t              service_handle;
    uint8_t               uuid_type;
    ble_button_evt_handler_t evt_handler;
    ble_gatts_char_handles_t button_on_press_char_handles;
    ble_gatts_char_handles_t button_off_press_char_handles;

} ble_button_service_t;
```

- The following function handles initializing the Service, and is called from the application level when initializing all of the device's Services:

```
/**@brief Function for initializing the Button Service.
 *
 * @param[out] p_button_service  Button Service structure. This structure will have to be supplied by
 *                               the application. It will be initialized by this function, and will
 * later
 *                               be used to identify this particular service instance.
 *
 * @return      NRF_SUCCESS on successful initialization of service, otherwise an error code.
 */

```

```
uint32_t ble_button_service_init(ble_button_service_t * p_button_service, ble_button_evt_handler_t
evt_handler);
```

- The following function gets called from the application level to pass on BLE events to be processed by the Button Service. " /**@brief Function for handling the Application's BLE Stack events.

```
/**@brief Function for handling the Application's BLE Stack events.
*
* @details Handles all events from the BLE stack of interest to the Button Service.
*
*
* @param[in]    p_button_service  Button Service structure.
* @param[in]    p_ble_evt        Event received from the BLE stack.
*/
void ble_button_service_on_ble_evt(ble_evt_t const * p_ble_evt, void * p_context);
```

-This function is called to send a Notification to a subscribed Client to indicate a change in value of the ON or OFF buttons:

```
void button_characteristic_update(ble_button_service_t * p_button_service, uint8_t pin_no, uint8_t
*button_action, bool button_notifications_enabled);
```

- End of the conditional include:

```
#endif /* BUTTON_SERVICE_H */
```

button_service.c

This is the Button Service module source file. It handles adding the Button Service, along with the Characteristics, and handling events related to the Button Service.

- Include any standard header files as well as the Button Service header file:

```
#include <string.h>

#include "nrf_log.h"
#include "button_service.h"
```

- These are string variables that get assigned to the Characteristic Descriptors to make it possible to present a

human-readable name. The strings can be seen in a Central Emulator application by observing the Characteristic Descriptors for a given Characteristic.

```
static const uint8_t ButtonOnCharName[] = "Button ON press";
static const uint8_t ButtonOffCharName[] = "Button OFF press";
```

- This is a local (static) function that handles the Connection Event and assigns the connection handle to the Button Service instance:

```
/**@brief Function for handling the Connect event.
*
* @param[in]    p_bas      Button service structure.
* @param[in]    p_ble_evt   Event received from the BLE stack.
*/
static void on_connect(ble_button_service_t * p_button_service, ble_evt_t const * p_ble_evt)
{
    p_button_service->conn_handle = p_ble_evt->evt.gap_evt.conn_handle;
}
```

- This is a local (static) function that handles the BLE disconnection event:

```
/**@brief Function for handling the Disconnect event.
*
* @param[in]    p_bas      Button service structure.
* @param[in]    p_ble_evt   Event received from the BLE stack.
*/
static void on_disconnect(ble_button_service_t * p_button_service, ble_evt_t const * p_ble_evt)
{
    UNUSED_PARAMETER(p_ble_evt);
    p_button_service->conn_handle = BLE_CONN_HANDLE_INVALID;
}
```

- A function to handle the Write events. It handles the following events and passes them up to the application layer by calling the application-assigned event handler:

```
/**@brief Function for handling the Write event.
*
* @param[in]    p_button_service  Button Service structure.
* @param[in]    p_ble_evt         Event received from the BLE stack.
*/
static void on_write(ble_button_service_t * p_button_service, ble_evt_t const * p_ble_evt)
{
    // Only continue if we have an event handler to call
    if (p_button_service->evt_handler == NULL)
    {
```

```

        return;
    }

ble_gatts_evt_write_t const * p_evt_write = &p_ble_evt->evt.gatts_evt.params.write;
NRF_LOG_INFO("Write event received");

// Handle the Button Off Notification enabled/disabled event
if (    (p_evt_write->handle == p_button_service->button_off_press_char_handles.cccd_handle)
    && (p_evt_write->len == 2))
{
    if (p_button_service->evt_handler == NULL)
    {
        NRF_LOG_INFO("Event handler is NULL");
        return;
    }

    ble_button_evt_t evt;

    if (ble_srv_is_notification_enabled(p_evt_write->data))
    {
        NRF_LOG_INFO("Button Off Notification enabled");
        evt.evt_type = BLE_BUTTON_OFF_EVT_NOTIFICATION_ENABLED;
    }
    else
    {
        NRF_LOG_INFO("Button Off Notification disabled");
        evt.evt_type = BLE_BUTTON_OFF_EVT_NOTIFICATION_DISABLED;
    }
    evt.conn_handle = p_ble_evt->evt.gatts_evt.conn_handle;

    // CCCD written, call application event handler.
    p_button_service->evt_handler(p_button_service, &evt);
}
// Handle the Button On Notification enabled/disabled event
else if (    (p_evt_write->handle == p_button_service->button_on_press_char_handles.cccd_handle)
    && (p_evt_write->len == 2))
{
    if (p_button_service->evt_handler == NULL)
    {
        NRF_LOG_INFO("Event handler is NULL");
        return;
    }

    ble_button_evt_t evt;

    if (ble_srv_is_notification_enabled(p_evt_write->data))
    {
        NRF_LOG_INFO("Button On Notification enabled");
        evt.evt_type = BLE_BUTTON_ON_EVT_NOTIFICATION_ENABLED;
    }
    else
    {
        NRF_LOG_INFO("Button On Notification disabled");
    }
}

```

```

        evt.evt_type = BLE_BUTTON_ON_EVT_NOTIFICATION_DISABLED;
    }
    evt.conn_handle = p_ble_evt->evt.gatts_evt.conn_handle;

    // CCCD written, call application event handler.
    p_button_service->evt_handler(p_button_service, &evt);
}
}

```

- A local (static) function that gets called from the `service_init()` function to add the ON button Characteristic. It defines a few things for the Characteristic:
 - The UUID
 - The different permissions
 - Adding the Characteristic to the database

```

/**@brief Function for adding the Button ON press characteristic.
*/
static uint32_t button_on_press_char_add(ble_button_service_t * p_button_service)
{
    ble_gatts_char_md_t char_md;
    ble_gatts_attr_md_t cccd_md;
    ble_gatts_attr_t    attr_char_value;
    ble_uuid_t          ble_uuid;
    ble_gatts_attr_md_t attr_md;

    memset(&char_md, 0, sizeof(char_md));
    memset(&cccd_md, 0, sizeof(cccd_md));
    memset(&attr_md, 0, sizeof(attr_md));
    memset(&attr_char_value, 0, sizeof(attr_char_value));

    // Set permissions on the CCCD and Characteristic value
    BLE_GAP_CONN_SEC_MODE_SET_OPEN(&cccd_md.read_perm);
    BLE_GAP_CONN_SEC_MODE_SET_OPEN(&cccd_md.write_perm);
    BLE_GAP_CONN_SEC_MODE_SET_NO_ACCESS(&attr_md.write_perm);
    BLE_GAP_CONN_SEC_MODE_SET_OPEN(&attr_md.read_perm);

    // CCCD settings (needed for notifications and/or indications)
    cccd_md.vloc = BLE_GATTS_VLOC_STACK;
    char_md.char_props.read      = 1;
    char_md.char_props.notify    = 1;
    char_md.p_char_user_desc    = ButtonOnCharName;
    char_md.char_user_desc_size = sizeof(ButtonOnCharName);
    char_md.char_user_desc_max_size = sizeof(ButtonOnCharName);
    char_md.p_char_pf           = NULL;
    char_md.p_user_desc_md      = NULL;
    char_md.p_cccd_md           = &cccd_md;
    char_md.p_sccd_md           = NULL;

    // Define the Button ON press Characteristic UUID

```

```

ble_uuid.type = p_button_service->uuid_type;
ble_uuid.uuid = BLE_UUID_BUTTON_ON_PRESS_CHAR_UUID;

// Attribute Metadata settings
attr_md.vloc      = BLE_GATTS_VLOC_STACK;
attr_md.rd_auth   = 0;
attr_md.wr_auth   = 0;
attr_md.vlen      = 0;

// Attribute Value settings
attr_char_value.p_uuid      = &ble_uuid;
attr_char_value.p_attr_md    = &attr_md;
attr_char_value.init_len     = sizeof(uint8_t);
attr_char_value.init_offs    = 0;
attr_char_value.max_len      = sizeof(uint8_t);
attr_char_value.p_value      = NULL;

return sd_ble_gatts_characteristic_add(p_button_service->service_handle, &char_md,
                                       &attr_char_value,
                                       &p_button_service->button_on_press_char_handles);
}

```

- The following function adds the OFF button press Characteristic. It is very similar to the above function for adding the ON button Characteristic:

```

/**@brief Function for adding the Button OFF press characteristic.
*/
static uint32_t button_off_press_char_add(ble_button_service_t * p_button_service)
{
    ble_gatts_char_md_t char_md;
    ble_gatts_attr_md_t cccd_md;
    ble_gatts_attr_t    attr_char_value;
    ble_uuid_t          ble_uuid;
    ble_gatts_attr_md_t attr_md;

    memset(&char_md, 0, sizeof(char_md));
    memset(&cccd_md, 0, sizeof(cccd_md));
    memset(&attr_md, 0, sizeof(attr_md));
    memset(&attr_char_value, 0, sizeof(attr_char_value));

    // Set permissions on the CCCD and Characteristic value
    BLE_GAP_CONN_SEC_MODE_SET_OPEN(&cccd_md.read_perm);
    BLE_GAP_CONN_SEC_MODE_SET_OPEN(&cccd_md.write_perm);
    BLE_GAP_CONN_SEC_MODE_SET_NO_ACCESS(&attr_md.write_perm);
    BLE_GAP_CONN_SEC_MODE_SET_OPEN(&attr_md.read_perm);

    // CCCD settings (needed for notifications and/or indications)
    cccd_md.vloc = BLE_GATTS_VLOC_STACK;
    char_md.char_props.read      = 1;
    char_md.char_props.notify    = 1;
    char_md.p_char_user_desc     = ButtonOffCharName;
}

```

```

char_md.char_user_desc_size      = sizeof(ButtonOffCharName);
char_md.char_user_desc_max_size = sizeof(ButtonOffCharName);
char_md.p_char_pf               = NULL;
char_md.p_user_desc_md          = NULL;
char_md.p_cccd_md               = &cccd_md;
char_md.p_sccd_md               = NULL;

// Define the Button OFF press Characteristic UUID
ble_uuid.type = p_button_service->uuid_type;
ble_uuid.uuid = BLE_UUID_BUTTON_OFF_PRESS_CHAR_UUID;

// Attribute Metadata settings
attr_md.vloc      = BLE_GATTS_VLOC_STACK;
attr_md.rd_auth   = 0;
attr_md.wr_auth   = 0;
attr_md.vlen      = 0;

// Attribute Value settings
attr_char_value.p_uuid      = &ble_uuid;
attr_char_value.p_attr_md    = &attr_md;
attr_char_value.init_len     = sizeof(uint8_t);
attr_char_value.init_offs    = 0;
attr_char_value.max_len      = sizeof(uint8_t);
attr_char_value.p_value      = NULL;

return sd_ble_gatts_characteristic_add(p_button_service->service_handle, &char_md,
                                       &attr_char_value,
                                       &p_button_service->button_off_press_char_handles);
}

```

- The following function is a “public” function that is exposed by the button service module, and gets called to initialize the Service along with adding all its Characteristics. It handles the following:
 - Adding the Service UUID
 - Adding the Service to the database
 - Add the ON Button Characteristic
 - Add the OFF Button Characteristic

```

uint32_t ble_button_service_init(ble_button_service_t * p_button_service, ble_button_evt_handler_t
evt_handler)
{
    uint32_t err_code;
    ble_uuid_t ble_uuid;

    // Initialize service structure
    p_button_service->conn_handle = BLE_CONN_HANDLE_INVALID;
    p_button_service->evt_handler = evt_handler;

    // Add service UUID
    ble_uuid128_t base_uuid = {BLE_UUID_BUTTON_SERVICE_BASE_UUID};
    err_code = sd_ble_uuid_vs_add(&base_uuid, &p_button_service->uuid_type);
}

```

```

if (err_code != NRF_SUCCESS)
{
    return err_code;
}

// Set up the UUID for the service (base + service-specific)
ble_uuid.type = p_button_service->uuid_type;
ble_uuid.uuid = BLE_UUID_BUTTON_SERVICE_UUID;

// Set up and add the service
err_code = sd_ble_gatts_service_add(BLE_GATTS_SRVC_TYPE_PRIMARY, &ble_uuid, &p_button_service-
>service_handle);
if (err_code != NRF_SUCCESS)
{
    return err_code;
}

// Add the different characteristics in the service:
//    ON Button press characteristic: E54B0002-67F5-479E-8711-B3B99198CE6C
//    OFF Button press characteristic: E54B0003-67F5-479E-8711-B3B99198CE6C
err_code = button_on_press_char_add(p_button_service);
if (err_code != NRF_SUCCESS)
{
    return err_code;
}

err_code = button_off_press_char_add(p_button_service);
if (err_code != NRF_SUCCESS)
{
    return err_code;
}

return NRF_SUCCESS;
}

```

- The following function gets called by the SoftDevice to pass on any relevant BLE events. In our case, we are interested in handling the following events:
 - Connection event
 - Disconnection event
 - Write event

```

void ble_button_service_on_ble_evt(ble_evt_t const * p_ble_evt, void * p_context)
{
    if ((p_context == NULL) || (p_ble_evt == NULL))
    {
        return;
    }

    ble_button_service_t * p_button_service = (ble_button_service_t *)p_context;

```

```

switch (p_ble_evt->header.evt_id)
{
    case BLE_GAP_EVT_CONNECTED:
        on_connect(p_button_service, p_ble_evt);
        break;

    case BLE_GAP_EVT_DISCONNECTED:
        on_disconnect(p_button_service, p_ble_evt);
        break;

    case BLE_GATTS_EVT_WRITE:
        on_write(p_button_service, p_ble_evt);
        break;

    default:
        // No implementation needed.
        break;
}
}

```

- This function allows the application level to update the value of the ON or OFF button Characteristic and Notify any subscribed Clients of the new value. This function handles both button press status updates:

```

void button_characteristic_update(ble_button_service_t * p_button_service, uint8_t pin_no, uint8_t
*button_action, bool button_notifications_enabled)
{
    if (p_button_service->conn_handle != BLE_CONN_HANDLE_INVALID)
    {
        uint32_t err_code;
        uint16_t          len = sizeof(uint8_t);
        ble_gatts_value_t gatts_value;

        // Initialize value struct.
        memset(&gatts_value, 0, sizeof(gatts_value));

        gatts_value.len      = sizeof(uint8_t);
        gatts_value.offset   = 0;
        gatts_value.p_value = button_action;

        // Update database.
        if (pin_no == BUTTON_1)
        {
            err_code = sd_ble_gatts_value_set(BLE_CONN_HANDLE_INVALID,
                                              p_button_service-
>button_on_press_char_handles.value_handle,
                                              &gatts_value);
        }
        else if (pin_no == BUTTON_2)
        {
            err_code = sd_ble_gatts_value_set(BLE_CONN_HANDLE_INVALID,
                                              p_button_service-
>button_off_press_char_handles.value_handle,
                                              &gatts_value);
        }
    }
}

```

```

        &gatts_value);
    }

// Only send a Notification if the Client has subscribed
if (button_notifications_enabled)
{
    ble_gatts_hvx_params_t hvx_params;
    memset(&hvx_params, 0, sizeof(hvx_params));

    if (pin_no == BUTTON_1)
    {
        hvx_params.handle = p_button_service->button_on_press_char_handles.value_handle;
    }
    else
    {
        hvx_params.handle = p_button_service->button_off_press_char_handles.value_handle;
    }
    hvx_params.type    = BLE_GATT_HVX_NOTIFICATION;
    hvx_params.offset  = 0;
    hvx_params.p_len   = &len;
    hvx_params.p_data  = (uint8_t*)button_action;

    sd_ble_gatts_hvx(p_button_service->conn_handle, &hvx_params);
}
}
}

```

main.h

This file serves as the header file for the main application file `main.c`.

- Conditional include for header file (prevents from including the file twice causing redefined macros):

```
#ifndef MAIN_H
#define MAIN_H
```

- Including standard C files as well as the nRF SDK files needed:

```
#include <stdbool.h>
#include <stdint.h>
#include <string.h>
#include "nordic_common.h"
#include "nrf.h"
#include "app_error.h"
#include "ble.h"
```

```
#include "ble_hci.h"
#include "ble_srv_common.h"
#include "ble_advdata.h"
#include "ble_advertising.h"
#include "ble_conn_params.h"
#include "nrf_sdh.h"
#include "nrf_sdh_soc.h"
#include "nrf_sdh_ble.h"
#include "app_timer.h"
#include "fds.h"
#include "peer_manager.h"
#include "bsp_btn_ble.h"
#include "sensorsim.h"
#include "ble_conn_state.h"
#include "nrf_ble_gatt.h"
#include "ble_gap.h"

#include "nrf_log.h"
#include "nrf_log_ctrl.h"
#include "nrf_log_default_backends.h"
```

- Including Battery Service module, Battery Voltage Reading module, and Button Service Module:

```
#include "ble_bas.h"
#include "services/button_service.h"

#include "battery_level/battery_voltage.h"
```

- Here, we define the Device Name that gets included in the Advertisement Packet to make it easy to find our Peripheral by a Central:

```
#define DEVICE_NAME          "NovelBits RC"           /**< Name of device. Will be included
in the advertising data. */
```

- Define the Manufacturer name, which gets included in the Device Information Service:

```
#define MANUFACTURER_NAME      "NordicSemiconductor"    /**< Manufacturer. Will
be passed to Device Information Service. */
```

- Here, we define the Advertising Parameters. Keep in mind that the Advertising Interval value is a multiple of 0.625 ms, so the value here corresponds to $300 \times 0.625 = 187.5$ ms:

```
#define APP_ADV_INTERVAL      300                  /**< The advertising
interval (in units of 0.625 ms. This value corresponds to 187.5 ms). */
```

- Here, we define the Advertising Timeout of the Peripheral. We choose a non-zero value to optimize battery consumption. *To Advertise again, we can choose to implement assigning a button press that triggers the Advertising state.*

```
#define APP_ADV_TIMEOUT_IN_SECONDS      180          /**< The advertising
timeout in units of seconds. */
```

- Next, we define the Connection Parameters. Since we are operating in the Peripheral Role, these values will simply be included in the GAP Service as Peripheral Preferred Connection Parameters (PPCP), which are suggested values to the Central. In the end, the Central makes the ultimate decision in accepting or rejecting these values and choosing different ones:
 - Minimum connection interval = 15 ms
 - Maximum connection interval = 15 ms
 - Slave latency = 9
 - Connection supervision timeout = 4 sec

```
#define MIN_CONN_INTERVAL      MSEC_TO_UNITS(15, UNIT_1_25_MS)    /**< Minimum acceptable
connection interval (15 msec). */
#define MAX_CONN_INTERVAL      MSEC_TO_UNITS(15, UNIT_1_25_MS)    /**< Maximum acceptable
connection interval (15 msec). */
#define SLAVE_LATENCY         9          /**< Slave latency. */
#define CONN_SUP_TIMEOUT       MSEC_TO_UNITS(4000, UNIT_10_MS)    /**< Connection
supervisory timeout (4 seconds). */
```

- The following macros define timing configurations for the Connection Parameters update message that can be sent by the peripheral. *For more information on this topic, refer to [this Nordic InfoCenter section].*

```
#define FIRST_CONN_PARAMS_UPDATE_DELAY  APP_TIMER TICKS(5000)           /**< Time from initiating
event (connect or start of notification) to first time sd_ble_gap_conn_param_update is called (5
seconds). */
#define NEXT_CONN_PARAMS_UPDATE_DELAY   APP_TIMER TICKS(30000)           /**< Time between each
call to sd_ble_gap_conn_param_update after the first call (30 seconds). */
#define MAX_CONN_PARAMS_UPDATE_COUNT   3          /**< Number of attempts
before giving up the connection parameter negotiation. */
```

- The following value is used to make it easier to find a location on the stack if it gets corrupted. In the case of an error, the value `0xDEADBEEF` is written to memory, which makes its easy to locate when viewing the contents of memory displayed in HEX:

```
#define DEAD_BEEF          0xDEADBEEF          /**< Value used as error
code on stack dump, can be used to identify stack location on stack unwind. */
```

- We define the period at which battery levels are measured from the coin cell battery installed on the development kit, at 120×1000 ms = 120 seconds:

```
#define BATTERY_LEVEL_MEAS_INTERVAL      APP_TIMER TICKS(120000) /*< Battery level measurement interval
(ticks). */
```

- The following values are taken from an example that Nordic provides for reading correct battery levels of the onboard (development kit) battery:

```
#define ADC_REF_VOLTAGE_IN_MILLIVOLTS      1200
#define ADC_PRE_SCALING_COMPENSATION        3
#define DIODE_FWD_VOLT_DROP_MILLIVOLTS     270
#define ADC_RESULT_IN_MILLI_VOLTS(ADC_VALUE) (((ADC_VALUE) * ADC_REF_VOLTAGE_IN_MILLIVOLTS) / 1023) *
ADC_PRE_SCALING_COMPENSATION)
#define ADC_BUFFER_SIZE 6
for ADC samples. */ /*< Size of buffer
```

main.c

This file contains our main application code:

- Include the header file for `main.c` :

```
#include "main.h"
```

- Here, we instantiate the Battery Service module (which gets passed to the `init()` function for the Battery Service module):

```
/*< Structure used to identify the battery service. */
BLE_BAS_DEF(m_bas);
```

- Instantiate the Button Service via the macro we defined in `button_service.h` :

```
// Define the Button Service
BLE_BUTTON_SERVICE_DEF(button_service);
```

- Here, we are defining the number of buttons we want to handle in our application (which gets used in the array defined below). In our case, we have two buttons: Button #1 and Button #2:

```
#define NUM_OF_BUTTONS 2
```

—Forward-declaration” for the button event handler (in order to use it in the button configuration array below):

```
static void button_event_handler(uint8_t pin_no, uint8_t button_action);
```

- We define the button configuration that gets passed to the BSP library to handle button events such as presses and releases:

```
//Configure 2 buttons with pullup and detection on low state
static const app_button_cfg_t app_buttons[NUM_OF_BUTTONS] =
{
    {BUTTON_1, false, BUTTON_PULL, button_event_handler},
    {BUTTON_2, false, BUTTON_PULL, button_event_handler},
};
```

- Define a timer that gets used for battery level measurements:

```
/**< Battery timer. */
APP_TIMER_DEF(m_battery_timer_id);
```

- Instantiate the GATT module and the Advertising module (both of which are provided by the SoftDevice):

```
NRF_BLE_GATT_DEF(m_gatt);                                /**< GATT module
instance. */
BLE_ADVERTISING_DEF(m_advertising);                     /**< Advertising module
instance. */
```

- Define a variable for saving the connection handle:

```
static uint16_t m_conn_handle = BLE_CONN_HANDLE_INVALID;      /**< Handle of the
current connection. */
```

- Define the UUIDs included in the Advertisement packet. We are including the Device Information Service UUID:

```
static ble_uuid_t m_adv_uuids[] =                               /**< Universally unique
```

```
service identifiers.*/
{
    {BLE_UUID_DEVICE_INFORMATION_SERVICE, BLE_UUID_TYPE_BLE}
};
```

- Forward declarations for local (static) functions used by other application functions:

```
static void advertising_start();
static void battery_level_meas_timeout_handler(void * p_context);
static void battery_level_update(void);
```

- Callback function for asserts returned by the SoftDevice. This helps in locating errors in the stack dump:

```
/**@brief Callback function for asserts in the SoftDevice.
*
* @details This function will be called in case of an assert in the SoftDevice.
*
* @warning This handler is an example only and does not fit a final product. You need to analyze
*          how your product is supposed to react in case of Assert.
* @warning On assert from the SoftDevice, the system can only recover on reset.
*
* @param[in] line_num    Line number of the failing ASSERT call.
* @param[in] file_name   File name of the failing ASSERT call.
*/
void assert_nrf_callback(uint16_t line_num, const uint8_t * p_file_name)
{
    app_error_handler(DEAD_BEEF, line_num, p_file_name);
}
```

- Function for initializing the Timer module. It handles initializing the Timer module as well as creating the timer we use for the Battery level measurements.

```
/**@brief Function for the Timer initialization.
*
* @details Initializes the timer module. This creates and starts application timers.
*/
static void timers_init(void)
{
    // Initialize timer module.
    ret_code_t err_code = app_timer_init();
    APP_ERROR_CHECK(err_code);

    // Create timers.
    err_code = app_timer_create(&m_battery_timer_id,
                               APP_TIMER_MODE_REPEAT,
                               battery_level_meas_timeout_handler);
    APP_ERROR_CHECK(err_code);
```

```
}
```

- Initialize the GAP parameters for our Peripheral. This includes:
 - Device name
 - Appearance value
 - Peripheral Preferred Connection Parameters (PPCP)

```
/**@brief Function for the GAP initialization.
*/
* @details This function sets up all the necessary GAP (Generic Access Profile) parameters of the
*         device including the device name, appearance, and the preferred connection parameters.
*/
static void gap_params_init(void)
{
    ret_code_t          err_code;
    ble_gap_conn_params_t  gap_conn_params;
    ble_gap_conn_sec_mode_t sec_mode;

    BLE_GAP_CONN_SEC_MODE_SET_OPEN(&sec_mode);

    err_code = sd_ble_gap_device_name_set(&sec_mode,
                                         (const uint8_t *)DEVICE_NAME,
                                         strlen(DEVICE_NAME));
    APP_ERROR_CHECK(err_code);

    err_code = sd_ble_gap_appearance_set(BLE_APPEARANCE_UNKNOWN);
    APP_ERROR_CHECK(err_code);

    memset(&gap_conn_params, 0, sizeof(gap_conn_params));

    gap_conn_params.min_conn_interval = MIN_CONN_INTERVAL;
    gap_conn_params.max_conn_interval = MAX_CONN_INTERVAL;
    gap_conn_params.slave_latency     = SLAVE_LATENCY;
    gap_conn_params.conn_sup_timeout = CONN_SUP_TIMEOUT;

    err_code = sd_ble_gap_ppcp_set(&gap_conn_params);
    APP_ERROR_CHECK(err_code);
}
```

- Function to initialize the GATT module:

```
/**@brief Function for initializing the GATT module.
*/
static void gatt_init(void)
{
    ret_code_t err_code = nrf_ble_gatt_init(&m_gatt, NULL);
    APP_ERROR_CHECK(err_code);
}
```

- The event handler passed to the Button Service during initialization to handle Button Service specific events including Notifications being enabled/disabled for both buttons 1 & 2:

```
// Button Service Event Handler
static void on_button_service_evt(ble_button_service_t * p_button_service, ble_button_evt_t * p_evt)
{
    ret_code_t err_code;

    switch (p_evt->evt_type)
    {
        case BLE_BUTTON_OFF_EVT_NOTIFICATION_ENABLED:
            m_button_off_notification_enabled = true;
            NRF_LOG_INFO("Button OFF notification enabled");
            break;

        case BLE_BUTTON_OFF_EVT_NOTIFICATION_DISABLED:
            m_button_off_notification_enabled = false;
            NRF_LOG_INFO("Button OFF notification disabled");
            break;

        case BLE_BUTTON_ON_EVT_NOTIFICATION_ENABLED:
            m_button_on_notification_enabled = true;
            NRF_LOG_INFO("Button ON notification enabled");
            break;

        case BLE_BUTTON_ON_EVT_NOTIFICATION_DISABLED:
            m_button_on_notification_enabled = false;
            NRF_LOG_INFO("Button ON notification disabled");
            break;
    }
}
```

- Function for initializing the different Services defined/used by our application. We have two services that we initialize: the Battery Service and the Button Service. For the Battery Service, we make sure that we enable Reads, Notifications and reading of Descriptors.

```
/**@brief Function for initializing services that will be used by the application.
 */
static void services_init(void)
{
    uint32_t      err_code;
    ble_bas_init_t bas_init;

    // 1. Initialize the Button service
    err_code = ble_button_service_init(&button_service, on_button_service_evt);
    APP_ERROR_CHECK(err_code);

    // 2. Initialize Battery Service.
    memset(&bas_init, 0, sizeof(bas_init));
```

```

// Here the sec level for the Battery Service can be changed/increased.
bas_init.bl_rd_sec      = SEC_OPEN;
bas_init.bl_cccd_wr_sec = SEC_OPEN;
bas_init.bl_report_rd_sec = SEC_OPEN;

bas_init.evt_handler      = NULL;
bas_init.support_notification = true;
bas_init.p_report_ref     = NULL;
bas_init.initial_batt_level = 100;

err_code = ble_bas_init(&m_bas, &bas_init);
APP_ERROR_CHECK(err_code);
}

```

- This function handles all events related to the Connection parameter events. We are only interested in this Module's failure event, in which case we disconnect.

```

/**@brief Function for handling the Connection Parameters Module events.
*
* @details This function will be called for all events in the Connection Parameters Module which
*          are passed to the application.
*          @note All this function does is to disconnect. This could have been done by simply
*          setting the disconnect_on_fail config parameter, but instead we use the event
*          handler mechanism to demonstrate its use.
*
* @param[in] p_evt  Event received from the Connection Parameters Module.
*/
static void on_conn_params_evt(ble_conn_params_evt_t * p_evt)
{
    ret_code_t err_code;

    if (p_evt->evt_type == BLE_CONN_PARAMS_EVT_FAILED)
    {
        err_code = sd_ble_gap_disconnect(m_conn_handle, BLE_HCI_CONN_INTERVAL_UNACCEPTABLE);
        APP_ERROR_CHECK(err_code);
    }
}

```

- A function to print the Connection Parameters error:

```

/**@brief Function for handling a Connection Parameters error.
*
* @param[in] nrf_error  Error code containing information about what went wrong.
*/
static void conn_params_error_handler(uint32_t nrf_error)
{
    APP_ERROR_HANDLER(nrf_error);
}

```

- Function for initializing the Connection Parameters module, which sets the Connection Parameter Update related parameters:

```
/**@brief Function for initializing the Connection Parameters module.
 */
static void conn_params_init(void)
{
    ret_code_t          err_code;
    ble_conn_params_init_t cp_init;

    memset(&cp_init, 0, sizeof(cp_init));

    cp_init.p_conn_params           = NULL;
    cp_init.first_conn_params_update_delay = FIRST_CONN_PARAMS_UPDATE_DELAY;
    cp_init.next_conn_params_update_delay = NEXT_CONN_PARAMS_UPDATE_DELAY;
    cp_init.max_conn_params_update_count = MAX_CONN_PARAMS_UPDATE_COUNT;
    cp_init.start_on_notify_cccd_handle = BLE_GATT_HANDLE_INVALID;
    cp_init.disconnect_on_fail       = false;
    cp_init.evt_handler             = on_conn_params_evt;
    cp_init.error_handler           = conn_params_error_handler;

    err_code = ble_conn_params_init(&cp_init);
    APP_ERROR_CHECK(err_code);
}
```

- Here, we start the timers needed for our application. In our case, we are using one timer only — for triggering battery level measurements (at a period of `BATTERY_LEVEL_MEAS_INTERVAL` seconds):

```
/**@brief Function for starting application timers.
 */
static void application_timers_start(void)
{
    uint32_t err_code;

    // Start application timers.
    err_code = app_timer_start(m_battery_timer_id, BATTERY_LEVEL_MEAS_INTERVAL, NULL);
    APP_ERROR_CHECK(err_code);

}
```

- The following function handles the event of our battery level measurement timer expiring, in which case we call the Battery Level update function.

```
/**@brief Function for handling the Battery measurement timer timeout.
 *
 * @details This function will be called each time the battery level measurement timer expires.
```

```

/*
 * @param[in] p_context  Pointer used for passing some arbitrary information (context) from the
 *                      app_start_timer() call to the timeout handler.
 */
static void battery_level_meas_timeout_handler(void * p_context)
{
    UNUSED_PARAMETER(p_context);
    NRF_LOG_INFO("Battery Level timeout event");
    battery_level_update();
}

```

- This function enters low power (sleep) mode:

```

/**@brief Function for putting the chip into sleep mode.
 *
 * @note This function will not return.
 */
static void sleep_mode_enter(void)
{
    ret_code_t err_code;

    err_code = bsp_indication_set(BSP_INDICATE_IDLE);
    APP_ERROR_CHECK(err_code);

    // Prepare wakeup buttons.
    err_code = bsp_btn_ble_sleep_mode_prepare();
    APP_ERROR_CHECK(err_code);

    // Go to system-off mode (this function will not return; wakeup will cause a reset).
    #ifdef DEBUG_NRF
        (void) sd_power_system_off();
        while(1);
    #else
        APP_ERROR_CHECK(sd_power_system_off());
    #endif // DEBUG_NRF
}

```

- Next, we define a function for handling Advertising Events:

```

/**@brief Function for handling advertising events.
 *
 * @details This function will be called for advertising events which are passed to the application.
 *
 * @param[in] ble_adv_evt  Advertising event.
 */
static void on_adv_evt(ble_adv_evt_t ble_adv_evt)
{
    ret_code_t err_code;

    switch (ble_adv_evt)

```

```
{
    case BLE_ADV_EVT_FAST:
        NRF_LOG_INFO("Fast advertising.");
        err_code = bsp_indication_set(BSP_INDICATE_ADVERTISING);
        APP_ERROR_CHECK(err_code);
        break;

    case BLE_ADV_EVT_IDLE:
        sleep_mode_enter();
        break;

    default:
        break;
}
}
```

- A function for handling generic BLE events. It's taken (mostly) from other examples within the nRF5 SDK. We customize a couple of events:
 - We clear the Notification enabled flags when we get disconnected.
 - We clear the Notification enabled flags when we have a fresh connection.

```
/**@brief Function for handling BLE events.
 *
 * @param[in] p_ble_evt Bluetooth stack event.
 * @param[in] p_context Unused.
 */
static void ble_evt_handler(ble_evt_t const * p_ble_evt, void * p_context)
{
    ret_code_t err_code = NRF_SUCCESS;

    ble_gap_evt_t const * p_gap_evt = &p_ble_evt->evt.gap_evt;

    switch (p_ble_evt->header.evt_id)
    {
        case BLE_GAP_EVT_DISCONNECTED:
            NRF_LOG_INFO("Disconnected.");
            m_conn_handle = BLE_CONN_HANDLE_INVALID;
            m_button_off_notification_enabled = false;
            m_button_on_notification_enabled = false;
            break;

        case BLE_GAP_EVT_CONNECTED:
            NRF_LOG_INFO("Connected.");
            err_code = bsp_indication_set(BSP_INDICATE_CONNECTED);
            APP_ERROR_CHECK(err_code);
            m_conn_handle = p_ble_evt->evt.gap_evt.conn_handle;
            m_button_off_notification_enabled = false;
            m_button_on_notification_enabled = false;
            break;

        case BLE_GATTC_EVT_TIMEOUT:
```

```

// Disconnect on GATT Client timeout event.
NRF_LOG_DEBUG("GATT Client Timeout.");
err_code = sd_ble_gap_disconnect(p_ble_evt->evt.gattc_evt.conn_handle,
                                 BLE_HCI_REMOTE_USER_TERMINATED_CONNECTION);
APP_ERROR_CHECK(err_code);
break;

case BLE_GATTS_EVT_TIMEOUT:
// Disconnect on GATT Server timeout event.
NRF_LOG_DEBUG("GATT Server Timeout.");
err_code = sd_ble_gap_disconnect(p_ble_evt->evt.gatts_evt.conn_handle,
                                 BLE_HCI_REMOTE_USER_TERMINATED_CONNECTION);
APP_ERROR_CHECK(err_code);
break;

case BLE_GAP_EVT_CONN_PARAM_UPDATE:
{
    NRF_LOG_INFO("Connection interval updated: 0x%x, 0x%x.",
                 p_gap_evt->params.conn_param_update.conn_params.min_conn_interval,
                 p_gap_evt->params.conn_param_update.conn_params.max_conn_interval);
} break;

case BLE_GAP_EVT_CONN_PARAM_UPDATE_REQUEST:
{
    // Accept parameters requested by the peer.
    ble_gap_conn_params_t params;
    params = p_gap_evt->params.conn_param_update_request.conn_params;
    err_code = sd_ble_gap_conn_param_update(p_gap_evt->conn_handle, &params);
    APP_ERROR_CHECK(err_code);

    NRF_LOG_INFO("Connection interval updated (upon request): 0x%x, 0x%x.",
                 p_gap_evt->params.conn_param_update_request.conn_params.min_conn_interval,
                 p_gap_evt->params.conn_param_update_request.conn_params.max_conn_interval);
} break;

case BLE_GAP_EVT_DATA_LENGTH_UPDATE_REQUEST:
{
    ble_gap_data_length_params_t dl_params;

    // Clearing the struct will effectively set members to @ref BLE_GAP_DATA_LENGTH_AUTO.
    memset(&dl_params, 0, sizeof(ble_gap_data_length_params_t));
    err_code = sd_ble_gap_data_length_update(p_ble_evt->evt.gap_evt.conn_handle, &dl_params,
NULL);
    APP_ERROR_CHECK(err_code);
} break;

case BLE_GATTS_EVT_SYS_ATTR_MISSING:
{
    NRF_LOG_DEBUG("BLE_GATTS_EVT_SYS_ATTR_MISSING");
    err_code = sd_ble_gatts_sys_attr_set(p_gap_evt->conn_handle, NULL, 0, 0);
    APP_ERROR_CHECK(err_code);
} break;

case BLE_GAP_EVT_PHY_UPDATE:

```

```

    {
        ble_gap_evt_phy_update_t const * p_phy_evt = &p_ble_evt->evt.gap_evt.params.phy_update;

        if (p_phy_evt->status == BLE_HCI_STATUS_CODE_LMP_ERROR_TRANSACTION_COLLISION)
        {
            // Ignore LL collisions.
            NRF_LOG_DEBUG("LL transaction collision during PHY update.");
            break;
        }

        ble_gap_phys_t phys = { 0 };
        phys.tx_phys = p_phy_evt->tx_phys;
        phys.rx_phys = p_phy_evt->rx_phys;

    } break;

    case BLE_GAP_EVT_PHY_UPDATE_REQUEST:
    {
        NRF_LOG_DEBUG("PHY update request.");
        ble_gap_phys_t const phys =
        {
            .rx_phys = BLE_GAP_PHY_AUTO,
            .tx_phys = BLE_GAP_PHY_AUTO,
        };
        err_code = sd_ble_gap_phy_update(p_ble_evt->evt.gap_evt.conn_handle, &phys);
        APP_ERROR_CHECK(err_code);
    } break;

    default:
        // No implementation needed.
        break;
    }
}

```

- Handle the button events. We handle these events for both Buttons 1 & 2 (presses and releases). Whenever we receive any of these events, we pass them on to the Button Service to update the value and send Notifications if necessary.

```

static void button_event_handler(uint8_t pin_no, uint8_t button_action)
{
    ret_code_t err_code;

    switch (pin_no)
    {
        case BUTTON_1:
            NRF_LOG_INFO("Button 1 %s\r\n", button_action == 1 ? "pressed":"released");
            button_characteristic_update(&button_service, BUTTON_1, &button_action,
m_button_off_notification_enabled);
            break;
        case BUTTON_2:
            NRF_LOG_INFO("Button 2 %s\r\n", button_action == 1 ? "pressed":"released");

```

```

        button_characteristic_update(&button_service, BUTTON_2, &button_action,
m_button_off_notification_enabled);
        break;

    default:
        APP_ERROR_HANDLER(pin_no);
        break;
    }

}

```

- A function to initialize the BLE stack (SoftDevice) — *standard code for nRF applications:*

```

/**@brief Function for initializing the BLE stack.
 *
 * @details Initializes the SoftDevice and the BLE event interrupt.
 */
static void ble_stack_init(void)
{
    ret_code_t err_code;

    err_code = nrf_sdh_enable_request();
    APP_ERROR_CHECK(err_code);

    // Configure the BLE stack using the default settings.
    // Fetch the start address of the application RAM.
    uint32_t ram_start = 0;
    err_code = nrf_sdh_ble_default_cfg_set(APP_BLE_CONN_CFG_TAG, &ram_start);
    APP_ERROR_CHECK(err_code);

    // Enable BLE stack.
    err_code = nrf_sdh_ble_enable(&ram_start);
    APP_ERROR_CHECK(err_code);

    // Register a handler for BLE events.
    NRF_SDH_BLE_OBSERVER(m_ble_observer, APP_BLE_OBSERVER_PRIO, ble_evt_handler, NULL);
}

```

- Function to initialize the Advertising module (also common to BLE Peripheral applications):

```

/**@brief Function for initializing the Advertising functionality.
 */
static void advertising_init(void)
{
    ret_code_t          err_code;
    ble_advertising_init_t init;

    memset(&init, 0, sizeof(init));

    init.advdata.name_type          = BLE_ADVDATA_FULL_NAME;

```

```

init.advdata.include_appearance      = true;
init.advdata.flags                 = BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE;
init.advdata.uuids_complete.uuid_cnt = sizeof(m_adv_uuids) / sizeof(m_adv_uuids[0]);
init.advdata.uuids_complete.p_uuids = m_adv_uuids;

init.config.ble_adv_fast_enabled   = true;
init.config.ble_adv_fast_interval = APP_ADV_INTERVAL;
init.config.ble_adv_fast_timeout  = APP_ADV_TIMEOUT_IN_SECONDS;

init.evt_handler = on_adv_evt;

err_code = ble_advertising_init(&m_advertising, &init);
APP_ERROR_CHECK(err_code);

ble_advertising_conn_cfg_tag_set(&m_advertising, APP_BLE_CONN_CFG_TAG);
}

```

- Function for initializing the buttons and LEDs:

```

/**@brief Function for initializing buttons and leds.
 *
 * @param[out] p_erase_bonds  Will be true if the clear bonding button was pressed to wake the
application up.
 */
static void buttons_leds_init()
{
    ret_code_t err_code;
    bsp_event_t startup_event;

    err_code = bsp_init(BSP_INIT_LEDS, NULL);
    APP_ERROR_CHECK(err_code);

    //init app_button module, 50ms detection delay (button debouncing)
    err_code = app_button_init((app_button_cfg_t *)app_buttons,
                               NUM_OF_BUTTONS,
                               APP_TIMER TICKS(50));
    APP_ERROR_CHECK(err_code);

    err_code = app_button_enable();
    APP_ERROR_CHECK(err_code);
}

```

- This function is for updating/reading the battery level (taken from the Proximity Application, which is documented at [this link], and located at nRF5_SDK_current\examples\ble_peripheral\ble_app_proximity in the GitHub repository that accompanies this book).The function is called every 2 minutes by the timer timeout handler:

```

/**@brief Function for updating the Battery Level measurement*/
static void battery_level_update(void)
{

```

```

ret_code_t err_code;

uint8_t battery_level;
uint16_t vbatt; // Variable to hold voltage reading
battery_voltage_get(&vbatt); // Get new battery voltage

battery_level = battery_level_in_percent(vbatt); //Transform the millivolts value into
battery level percent.
printf("ADC result in percent: %d\r\n", battery_level);

err_code = ble_bas_battery_level_update(&m_bas, battery_level, m_conn_handle);
if ((err_code != NRF_SUCCESS) &&
    (err_code != NRF_ERROR_INVALID_STATE) &&
    (err_code != NRF_ERROR_RESOURCES) &&
    (err_code != BLE_ERROR_GATTS_SYS_ATTR_MISSING)
    )
{
    APP_ERROR_HANDLER(err_code);
}
}

```

- This function initializes the Logger module:

```

/**@brief Function for initializing the nrf log module.
 */
static void log_init(void)
{
    ret_code_t err_code = NRF_LOG_INIT(NULL);
    APP_ERROR_CHECK(err_code);

    NRF_LOG_DEFAULT_BACKENDS_INIT();
}

```

- Power management function that simply goes to sleep until an event needs to be sent to our application:

```

/**@brief Function for the Power manager.
 */
static void power_manage(void)
{
    ret_code_t err_code = sd_app_evt_wait();
    APP_ERROR_CHECK(err_code);
}

```

- Start the Advertising operation:

```

/**@brief Function for starting advertising.
 */

```

```
static void advertising_start()
{
    ret_code_t err_code = ble_advertising_start(&m_advertising, BLE_ADV_MODE_FAST);

    APP_ERROR_CHECK(err_code);
}
```

- Main application function:

```
/**@brief Function for application main entry.
 */
int main(void)
{
    // Initialize.
    log_init();
    timers_init();
    buttons_leds_init();
    ble_stack_init();
    gap_params_init();
    gatt_init();
    advertising_init();
    battery_voltage_init();
    services_init();
    conn_params_init();

    // Start execution.
    NRF_LOG_INFO("Novel Bits Remote Control started.");

    application_timers_start();
    advertising_start();

    // Enter main loop.
    for (;;)
    {
        if (NRF_LOG_PROCESS() == false)
        {
            power_manage();
        }
    }
}
```

The following files provide functions for reading the battery level using the ADC. The source code is taken from the example application labeled [Proximity], located at `nRF5_SDK_current/examples/ble_peripheral/ble_app_proximity` within the GitHub repository accompanied with the book. We won't explain this code line-by-line — we'll simply list it.

battery_voltage.h

```
#ifndef BATTERY_VOLTAGE_H_
#define BATTERY_VOLTAGE_H_

#include <stdint.h>

/**@brief Function for initializing the battery voltage module.
 */
void battery_voltage_init(void);

/**@brief Function for reading the battery voltage.
 *
 * @param[out] p_vbatt Pointer to the battery voltage value.
 */
void battery_voltage_get(uint16_t * p_vbatt);

#endif // BATTERY_VOLTAGE_H_
```

battery_voltage.c

```
#include "battery_voltage.h"
#include "nrf_drv_saadc.h"
#include "sdk_macros.h"
#include "nrf_log.h"

#define ADC_REF_VOLTAGE_IN_MILLIVOLTS 600 //!< Reference voltage (in milli volts) used by ADC while
                                         // doing conversion.
#define DIODE_FWD_VOLT_DROP_MILLIVOLTS 270 //!< Typical forward voltage drop of the diode (Part no:
                                         // SD103ATW-7-F) that is connected in series with the voltage supply. This is the voltage drop when the
                                         // forward current is 1mA. Source: Data sheet of 'SURFACE MOUNT SCHOTTKY BARRIER DIODE ARRAY' available at
                                         // www.diodes.com.
#define ADC_RES_10BIT 1024 //!< Maximum digital value for 10-bit ADC conversion.
#define ADC_PRE_SCALING_COMPENSATION 6 //!< The ADC is configured to use VDD with 1/3 prescaling as
                                         // input. And hence the result of conversion is to be multiplied by 3 to get the actual value of the battery
                                         // voltage.
#define ADC_RESULT_IN_MILLI_VOLTS(ADC_VALUE) \
    (((ADC_VALUE) *ADC_REF_VOLTAGE_IN_MILLIVOLTS) / ADC_RES_10BIT) * ADC_PRE_SCALING_COMPENSATION

static nrf_saadc_value_t adc_buf; //!< Buffer used for storing ADC value.
static uint16_t m_batt_lvl_in_milli_volts; //!< Current battery level.

/**@brief Function handling events from 'nrf_drv_saadc.c'.
 */
* @param[in] p_evt SAADC event.
*/
```

```

static void saadc_event_handler(nrf_drv_saadc_evt_t const * p_evt)
{
    if (p_evt->type == NRF_DRV_SAADC_EVT_DONE)
    {
        nrf_saadc_value_t adc_result;

        adc_result = p_evt->data.done.p_buffer[0];

        m_batt_lvl_in_milli_volts = ADC_RESULT_IN_MILLI_VOLTS(adc_result) +
DIODE_FWD_VOLT_DROP_MILLIVOLTS;

        NRF_LOG_INFO("ADC reading - ADC:%d, In Millivolts: %d\r\n", adc_result,
m_batt_lvl_in_milli_volts);
    }
}

void battery_voltage_init(void)
{
    ret_code_t err_code = nrf_drv_saadc_init(NULL, saadc_event_handler);

    APP_ERROR_CHECK(err_code);

    nrf_saadc_channel_config_t config =
        NRF_DRV_SAADC_DEFAULT_CHANNEL_CONFIG_SE(NRF_SAADC_INPUT_VDD);
    err_code = nrf_drv_saadc_channel_init(0, &config);
    APP_ERROR_CHECK(err_code);

    err_code = nrf_drv_saadc_buffer_convert(&adc_buf, 1);
    APP_ERROR_CHECK(err_code);

    err_code = nrf_drv_saadc_sample();
    APP_ERROR_CHECK(err_code);
}

void battery_voltage_get(uint16_t * p_vbatt)
{
    VERIFY_PARAM_NOT_NULL_VOID(p_vbatt);

    *p_vbatt = m_batt_lvl_in_milli_volts;
    if (!nrf_drv_saadc_is_busy())
    {
        ret_code_t err_code = nrf_drv_saadc_buffer_convert(&adc_buf, 1);
        APP_ERROR_CHECK(err_code);

        err_code = nrf_drv_saadc_sample();
        APP_ERROR_CHECK(err_code);
    }
}

```

Gateway Source Code Walkthrough

In this chapter we'll go through the full source code for the Gateway device within the Main Project. We'll cover the most important aspects of the project-specific functionality.

Make sure you refer to the chapter titled "**Main Project**" before reading this chapter. It is important to understand the overall architecture and design of the Main Project before digging into the source code for the Gateway device.

We've already covered setting up the Peripheral side with different Services and Characteristics a couple of times already, so we'll be skipping that part for the Gateway. The focus on here will be on the Central side setup and operations, so those portions of the code will be covered. These include:

- Scanning for Advertisements
- Discovering devices
- Connecting to Peripherals
- Setting the Connection Parameters
- Discovering Services and Characteristics for each of the Peripherals

The files we'll be walking through are:

- `main.c`
- `main.h`
- `central/central.h`
- `central/central.c`
- `central/thingy_client.h`
- `central/thingy_client.c`

The rest of the Gateway source code is very similar to source code we've already covered in previous exercises.

Important Note

The source code listed here is subject to change and may differ from the source code in the GitHub repository. Updates may be added to the code on GitHub to fix various bugs and add new features. The purpose of this "source code walkthrough" exercise is to guide the reader through the most important parts of the implementation for the nRF52 series platform.

Source code walkthrough

`main.c`

This file serves as the main file in the Project.

- First, we include the necessary header files. Many of these are standard nRF5 SDK header files that need to be included in all Projects.

```
#include "nrf_fstorage.h"

#include "central/central.h"
#include "peripheral/peripheral.h"

#include "nrf_log.h"
#include "nrf_log_ctrl.h"
#include "nrf_log_default_backends.h"

#include "main.h"
```

- The following function handles starting Advertising and Scanning since the Gateway acts in both a Peripheral and Central at the same time (multi-role).

We need to check to make sure no Flash operations are in progress because Radio operations take precedent over Flash operations. To learn more about priorities and scheduling of operations, refer to [this InfoCenter section](#).

```
/**@brief Function for initiating advertising and scanning.
 */
static void adv_scan_start(void)
{
    ret_code_t err_code;

    //check if there are no flash operations in progress
    if (!nrf_fstorage_is_busy(NULL))
    {
        // Start scanning for peripherals
        scan_start();

        // Turn on the LED to signal scanning.
        bsp_board_led_on(CENTRAL_SCANNING_LED);

        // Start advertising.
        advertising_start();
    }
}
```

- The following function handles BLE events from the SoftDevice for both Central and Peripheral related events. It calls the corresponding handler for each role (Peripheral vs. Central).

The Role variable will be checked to determine which handler to call. In the case of an Advertising Timeout event,

the event is passed on to the Peripheral handler. In the case of an Advertising Report, the event is passed on to the Central handler.

```
/**@brief Function for handling BLE events.
*
* @param[in] p_ble_evt Bluetooth stack event.
* @param[in] p_context Unused.
*/
static void ble_evt_handler(ble_evt_t const * p_ble_evt, void * p_context)
{
    uint16_t conn_handle = p_ble_evt->evt.gap_evt.conn_handle;
    uint16_t role        = ble_conn_state_role(conn_handle);

    // Based on the role this device plays in the connection, dispatch to the right handler.
    if (role == BLE_GAP_ROLE_PERIPH || ble_evt_is_advertising_timeout(p_ble_evt))
    {
        // Generic BLE event handler (passes on to services + generic handler)
        on_ble_peripheral_evt(p_ble_evt);
    }
    else if ((role == BLE_GAP_ROLE_CENTRAL) || (p_ble_evt->header.evt_id == BLE_GAP_EVT_ADV_REPORT))
    {
        // Generic BLE Central event handler - calls event handlers for each of the clients
        on_ble_central_evt(p_ble_evt);
    }
}
```

- The following function initializes the Timer module. It is needed for flashing of LEDs.

```
/**@brief Function for the Timer initialization.
*
* @details Initializes the timer module. This creates and starts application timers.
*/
static void timers_init(void)
{
    // Initialize timer module. NEEDED FOR LEDs and BUTTONS
    ret_code_t err_code = app_timer_init();
    APP_ERROR_CHECK(err_code);
}
```

- This function is responsible for putting the chipset in sleep mode (called by the BSP event handler function):

```
/**@brief Function for putting the chip into sleep mode.
*
* @note This function will not return.
*/
static void sleep_mode_enter(void)
{
    ret_code_t err_code;
```

```

err_code = bsp_indication_set(BSP_INDICATE_IDLE);
APP_ERROR_CHECK(err_code);

// Prepare wakeup buttons.
err_code = bsp_btn_ble_sleep_mode_prepare();
APP_ERROR_CHECK(err_code);

// Go to system-off mode (this function will not return; wakeup will cause a reset).
#ifndef DEBUG_NRF
(void) sd_power_system_off();
while(1);
#else
APP_ERROR_CHECK(sd_power_system_off());
#endif // DEBUG_NRF
}

```

- The following function initializes the BLE stack/SoftDevice. We've gone over this function before, and it's common for all Role configurations.

```

/**@brief Function for initializing the BLE stack.
 *
 * @details Initializes the SoftDevice and the BLE event interrupt.
 */
static void ble_stack_init(void)
{
    //ble_opt_t conn_event_length_extension;
    ret_code_t err_code;

    err_code = nrf_sdh_enable_request();
    APP_ERROR_CHECK(err_code);

    // Configure the BLE stack using the default settings.
    // Fetch the start address of the application RAM.
    uint32_t ram_start = 0;
    err_code = nrf_sdh_ble_default_cfg_set(APP_BLE_CONN_CFG_TAG, &ram_start);
    APP_ERROR_CHECK(err_code);

    // Enable BLE stack.
    err_code = nrf_sdh_ble_enable(&ram_start);
    APP_ERROR_CHECK(err_code);

    //conn_event_length_extension.common_opt.conn_evt_ext.enable = 1;
    //((void) sd_ble_opt_set(BLE_COMMON_OPT_CONN_EVT_EXT, &conn_event_length_extension));

    // Register a handler for BLE events.
    NRF_SDH_BLE_OBSERVER(m_ble_observer, APP_BLE_OBSERVER_PRIO, ble_evt_handler, NULL);
}

```

- The following function handles the BSP events from the SoftDevice.

```
/**@brief Function for handling events from the BSP module.
 *
 * @param[in] event Event generated when button is pressed.
 */
static void bsp_event_handler(bsp_event_t event)
{
    ret_code_t err_code;

    switch (event)
    {
        case BSP_EVENT_SLEEP:
            sleep_mode_enter();
            break; // BSP_EVENT_SLEEP

        default:
            break;
    }
}
```

- The following function initializes the LEDs and buttons.

```
/**@brief Function for initializing buttons and leds.
 *
 * @param[out] p_erase_bonds Will be true if the clear bonding button was pressed to wake the
application up.
 */
static void buttons_leds_init()
{
    ret_code_t err_code;
    bsp_event_t startup_event;

    err_code = bsp_init(BSP_INIT_LEDS, bsp_event_handler);
    APP_ERROR_CHECK(err_code);

    err_code = bsp_btn_ble_init(NULL, &startup_event);
    APP_ERROR_CHECK(err_code);
}
```

- The next function initializes the Logger module.

```
/**@brief Function for initializing the nrf log module.
 */
static void log_init(void)
{
    ret_code_t err_code = NRF_LOG_INIT(NULL);
    APP_ERROR_CHECK(err_code);

    NRF_LOG_DEFAULT_BACKENDS_INIT();
```

```
}
```

- The following function is needed for power management.

```
/**@brief Function for the Power manager.
 */
static void power_manage(void)
{
    ret_code_t err_code = sd_app_evt_wait();
    APP_ERROR_CHECK(err_code);
}
```

- Finally, we have the `main()` function. We start by initializing the different modules including: Logger module, Timer module, Buttons & LEDs, BLE stack, Connection Parameters, GATT module, Connection Update parameters, Database Discovery module.

```
/**@brief Function for application main entry.
 */
int main(void)
{
    // Initialize various services
    log_init();
    timers_init();
    buttons_leds_init();
    ble_stack_init();
    gap_params_init();
    gatt_init();
    conn_params_init();
    db_discovery_init();

    // Initialize the central clients (Remote Control, Thingy:52, Playbulb Candle)
    central_init();

    // Initialize services, then advertise
    services_init();
    advertising_init();

    // Start advertising and scanning
    adv_scan_start();

    // Start execution.
    NRF_LOG_INFO("Novel Bits Gateway started.");

    for (;;)
    {
        if (NRF_LOG_PROCESS() == false)
        {
            // Wait for BLE events.
            power_manage();
        }
    }
}
```

```

    }
}
}
```

main.h

We've already covered most of the contents of this header file in the Remote Control source code walkthrough. Here, we'll focus on the Gateway-specific parts.

- The following macros assign the specific board LEDs to the different operations.

```

#define PERIPHERAL_ADVERTISING_LED      BSP_BOARD_LED_2
#define PERIPHERAL_CONNECTED_LED        BSP_BOARD_LED_3
#define CENTRAL_SCANNING_LED           BSP_BOARD_LED_0
#define CENTRAL_CONNECTED_LED          BSP_BOARD_LED_1
```

central.h

This is the header file for the Central role-specific code for the Gateway.

- We only need to include the BLE header file.

```
#include "ble.h"
```

- The following prototype is for the function responsible for initializing the GATT database discovery module.

```

/***
 * @brief Database discovery initialization.
 */
void db_discovery_init(void);
```

- The following prototype is for the function responsible for starting the scan operation.

```

/**@brief Function for initiating scanning.
*/
void scan_start(void);
```

- This is the function for handling the different BLE events reported by the SoftDevice.

```
/**@brief Function for handling BLE events from central applications.
*
* @details This function parses scanning reports and initiates a connection to peripherals when a
*          target UUID is found. It updates the status of LEDs used to report central applications
*          activity.
*
* @param[in] p_ble_evt Bluetooth stack event.
*/
void on_ble_central_evt(ble_evt_t const * p_ble_evt);
```

- Finally, we have the prototype for the function responsible for initializing all Central-role related operations.

```
void central_init(void);
```

central.c

Just like the previous section, we will focus here primarily on the code that's specific to the Central role.

- The following are `#include` declarations for the needed header files for the Central. It includes all the Clients for the different GATT servers we are interfacing with as the Gateway.

```
// Application specific includes
#include "../tools.h"
#include "thingy_client.h"
#include "remote_control_client.h"
#include "playbulb_client.h"
#include "../peripheral/peripheral.h"
#include "central.h"
```

- The following macros define the different scan and connection parameters for the Gateway. As we learned previously, the Gateway dictates the timing of the different connection-related operations. So, these values are integral to the operation of the whole system involving the different Peripherals.

We have the different scan parameters:

- Scan Interval = 100 ms
- Scan Window = 50 ms
- Scan Timeout = 0 (no timeout)

We also have the different connection parameters:

- Minimum Connection Interval = 15 ms
- Maximum Connection Interval = 15 ms
- Slave Latency = 9
- Supervision Timeout = 4 sec

```
#define SCAN_INTERVAL          0x00A0          /**< Determines scan
interval in units of 0.625 millisecond. */
#define SCAN_WINDOW            0x0050          /**< Determines scan
window in units of 0.625 millisecond. */
#define SCAN_TIMEOUT            0

#define MIN_CONNECTION_INTERVAL (uint16_t) MSEC_TO_UNITS(15, UNIT_1_25_MS) /**< Determines
minimum connection interval in milliseconds. */
#define MAX_CONNECTION_INTERVAL (uint16_t) MSEC_TO_UNITS(15, UNIT_1_25_MS) /**< Determines
maximum connection interval in milliseconds. */
#define SLAVE_LATENCY           9                /**< Determines slave
latency in terms of connection events. */
#define SUPERVISION_TIMEOUT     (uint16_t) MSEC_TO_UNITS(4000, UNIT_10_MS) /**< Determines
supervision time-out in units of 10 milliseconds. */
```

- Instantiate the Database Discovery module passing it the number of clients (one for each GATT service we're interested in discovering):

```
BLE_DB_DISCOVERY_ARRAY_DEF(m_db_discovery, 6);          /**< Database discovery module
instances. */
```

- Define the connection handles for the different clients. We also make sure they are initialized to an invalid connection handle.

```
static uint16_t m_conn_handle_thingy_client = BLE_CONN_HANDLE_INVALID;      /**< Connection handle
for the Thingy client application */
static uint16_t m_conn_handle_remote_control_client = BLE_CONN_HANDLE_INVALID; /**< Connection handle
for the Remote Control client application */
static uint16_t m_conn_handle_playbulb_client = BLE_CONN_HANDLE_INVALID;      /**< Connection handle
for the Remote Control client application */
```

- Define objects for the Remote Control client, the Playbulb candlelight client, and the Thingy:52 client.

```
//Definition for each of the clients
static thingy_client_t      m_thingy_client;
static remote_control_client_t m_remote_control_client;
static playbulb_client_t     m_playbulb_client;
```

- Define a Battery Service client instance for each of the devices (Thingy:52, Playbulb candlelight, and the Remote Control).

```
BLE_BAS_C_DEF(m_bas_thingy_client);                                /**< Battery Service
client module instance. */
BLE_BAS_C_DEF(m_bas_remote_control_client);                         /**< Battery Service
client module instance. */
BLE_BAS_C_DEF(m_bas_playbulb_client);                                /**< Battery Service
client module instance. */
```

- We also define variables to hold the battery level readings for each of the devices.

```
uint8_t m_thingy_battery_level;
uint8_t m_playbulb_battery_level;
uint8_t m_remote_control_battery_level;
```

- We define an array to hold the target devices' Advertising names:

```
/**@brief names which the central applications will scan for, and which will be advertised by the
peripherals.
 * if these are set to empty strings, the UUIDs defined below will be used
 */
#define NUMBER_OF_TARGET_PERIPHERALS 3
static char const *m_target_periph_names[NUMBER_OF_TARGET_PERIPHERALS] = { "Thingy", "NovelBits RC",
"Playbulb Candle"};
```

- We store the scanning parameters in the appropriate data structure.

```
/**@brief Parameters used when scanning.*/
static ble_gap_scan_params_t const m_scan_params =
{
    .active     = 1,
    .interval   = SCAN_INTERVAL,
    .window     = SCAN_WINDOW,
    .timeout    = SCAN_TIMEOUT,
    .scan_phys   = BLE_GAP_PHY_1MBPS,
    .filter_policy = BLE_GAP_SCAN_FP_ACCEPT_ALL
};
```

- This is the buffer where we store all the Advertisements discovered.

```
static uint8_t          m_scan_buffer_data[BLE_GAP_SCAN_BUFFER_MIN]; /**< buffer where advertising
```

```

reports will be stored by the SoftDevice. */

/**@brief Pointer to the buffer where advertising reports will be stored by the SoftDevice. */
static ble_data_t m_scan_buffer =
{
    m_scan_buffer_data,
    BLE_GAP_SCAN_BUFFER_MIN
};

```

- We store the connection parameters in the appropriate data structure.

```

/**@brief Connection parameters requested for connection. */
static ble_gap_conn_params_t const m_connection_param =
{
    MIN_CONNECTION_INTERVAL,
    MAX_CONNECTION_INTERVAL,
    SLAVE_LATENCY,
    SUPERVISION_TIMEOUT
};

```

- The following is the function responsible for handling all the GATT discovery events. It passes on the discovery events for each of the services.

```

/**@brief Function for handling database discovery events.
 *
 * @details This function is callback function to handle events from the database discovery module.
 *          Depending on the UUIDs that are discovered, this function should forward the events
 *          to their respective services.
 *
 * @param[in] p_event  Pointer to the database discovery event.
 */
static void db_disc_handler(ble_db_discovery_evt_t * p_evt)
{
    // Call event handlers for each of the peripherals (Thingy:52, Remote Control, Playbulb Candle)
    thingy_on_db_disc_evt(&m_thingy_client, p_evt);
    ble_bas_on_db_disc_evt(&m_bas_thingy_client, p_evt);
    playbulb_on_db_disc_evt(&m_playbulb_client, p_evt);
    ble_bas_on_db_disc_evt(&m_bas_playbulb_client, p_evt);
    remote_control_on_db_disc_evt(&m_remote_control_client, p_evt);
    ble_bas_on_db_disc_evt(&m_bas_remote_control_client, p_evt);
}

```

- The following function initializes the Database Discovery module:

```

/**
 * @brief Database discovery initialization.
 */

```

```
void db_discovery_init(void)
{
    ret_code_t err_code = ble_db_discovery_init(db_disc_handler);
    APP_ERROR_CHECK(err_code);
}
```

- The following function starts the Scanning operation:

```
/**@brief Function for initiating scanning.
 */
void scan_start(void)
{
    ret_code_t err_code;

    (void) sd_ble_gap_scan_stop();

    err_code = sd_ble_gap_scan_start(&m_scan_params, &m_scan_buffer);
    // It is okay to ignore this error since we are stopping the scan anyway.
    if (err_code != NRF_ERROR_INVALID_STATE)
    {
        NRF_LOG_INFO("Scanning started\n");
        APP_ERROR_CHECK(err_code);
    }
}
```

- The following functions handle the Battery Service Client events for each of the Peripherals (Thingy:52, Playbulb, and Remote Control).

For each of these Battery Service Client event handler functions we only continue if we receive an event relevant to the specific Peripheral.

We also handle the Database Discovery Complete event by calling the `ble_bas_c_handles_assign()` function passing it the specific Peripheral Battery Service Client pointer `p_bas_c`. This event occurs when the discovery of the Battery Service is successful.

When a discovery of a Battery Service is complete, we:

- Read the Battery Level Characteristic
- Enable Notifications on the Battery Level value

If the event is a Notification event or a Read Response for the Battery Level value, then we:

- Report up to any subscribed Clients
- Store the Battery Level value

```
/**@brief Function for handling Battery Level Collector events.
 *
```

```

* @param[in] p_bas_c      Pointer to Battery Service Client structure.
* @param[in] p_bas_c_evt  Pointer to event structure.
*/
static void bas_c_thingy_evt_handler(ble_bas_c_t * p_bas_c, ble_bas_c_evt_t * p_bas_c_evt)
{
    ret_code_t err_code;

    NRF_LOG_INFO("Battery Service Client event handler for Thingy");

    // We are interested in the Battery Service on the Thingy only
    if ((p_bas_c_evt->conn_handle != m_conn_handle_thingy_client) ||
        (m_conn_handle_thingy_client == NULL))
    {
        return;
    }

    switch (p_bas_c_evt->evt_type)
    {
        case BLE_BAS_C_EVT_DISCOVERY_COMPLETE:
            err_code = ble_bas_c_handles_assign(p_bas_c,
                                                p_bas_c_evt->conn_handle,
                                                &p_bas_c_evt->params.bas_db);
            APP_ERROR_CHECK(err_code);

            // Battery service discovered. Enable notification of Battery Level.
            NRF_LOG_DEBUG("Battery Service discovered on Thingy. Reading battery level.");
            err_code = ble_bas_c_bl_read(p_bas_c);
            APP_ERROR_CHECK(err_code);

            NRF_LOG_DEBUG("Enabling Battery Level Notification on Thingy. ");
            err_code = ble_bas_c_bl_notif_enable(p_bas_c);
            APP_ERROR_CHECK(err_code);
            break;

        case BLE_BAS_C_EVT_BATT_NOTIFICATION:
            NRF_LOG_DEBUG("Battery Level received from Thingy %d %%", p_bas_c_evt->params.battery_level);
            m_thingy_battery_level = p_bas_c_evt->params.battery_level;
            send_garage_sensor_battery_level_to_client(m_thingy_battery_level);
            break;

        case BLE_BAS_C_EVT_BATT_READ_RESP:
            NRF_LOG_DEBUG("Battery Level of Thingy Read as %d %%", p_bas_c_evt->params.battery_level);
            m_thingy_battery_level = p_bas_c_evt->params.battery_level;
            send_garage_sensor_battery_level_to_client(m_thingy_battery_level);
            break;

        default:
            break;
    }
}

```

```

/**@brief Function for handling Battery Level Collector events.
 *
 * @param[in] p_bas_c      Pointer to Battery Service Client structure.
 * @param[in] p_bas_c_evt   Pointer to event structure.
 */
static void bas_c_remote_control_evt_handler(ble_bas_c_t * p_bas_c, ble_bas_c_evt_t * p_bas_c_evt)
{
    ret_code_t err_code;

    NRF_LOG_INFO("Battery Service Client event handler for Remote Control");

    // We are interested in the Battery Service on the Remote Control only
    if ((p_bas_c_evt->conn_handle != m_conn_handle_remote_control_client) ||
        (m_conn_handle_remote_control_client == NULL))
    {
        return;
    }

    switch (p_bas_c_evt->evt_type)
    {
        case BLE_BAS_C_EVT_DISCOVERY_COMPLETE:
            err_code = ble_bas_c_handles_assign(p_bas_c,
                                                p_bas_c_evt->conn_handle,
                                                &p_bas_c_evt->params.bas_db);
            APP_ERROR_CHECK(err_code);

            // Battery service discovered. Enable notification of Battery Level.
            NRF_LOG_DEBUG("Battery Service discovered on Remote Control. Reading battery level...");

            err_code = ble_bas_c_bl_read(p_bas_c);
            APP_ERROR_CHECK(err_code);

            NRF_LOG_DEBUG("Enabling Battery Level Notification on Remote Control.");
            err_code = ble_bas_c_bl_notif_enable(p_bas_c);
            APP_ERROR_CHECK(err_code);
            break;

        case BLE_BAS_C_EVT_BATT_NOTIFICATION:
            NRF_LOG_DEBUG("Battery Level received from Remote Control %d %%", p_bas_c_evt-
>params.battery_level);
            m_remote_control_battery_level = p_bas_c_evt->params.battery_level;
            send_remote_control_battery_level_to_client(m_remote_control_battery_level);
            break;

        case BLE_BAS_C_EVT_BATT_READ_RESP:
            NRF_LOG_INFO("Battery Level of Remote Control Read as %d %%", p_bas_c_evt-
>params.battery_level);
            m_remote_control_battery_level = p_bas_c_evt->params.battery_level;
            send_remote_control_battery_level_to_client(m_remote_control_battery_level);
            break;

        default:
            break;
    }
}

```

```

    }
}
```

```

/**@brief Function for handling Battery Level Collector events from the Playbulb.
*
* @param[in] p_bas_c      Pointer to Battery Service Client structure.
* @param[in] p_bas_c_evt   Pointer to event structure.
*/
static void bas_c_playbulb_evt_handler(ble_bas_c_t * p_bas_c, ble_bas_c_evt_t * p_bas_c_evt)
{
    ret_code_t err_code;

    NRF_LOG_INFO("Battery Service Client event handler for Playbulb");

    // We are interested in the Battery Service on the Playbulb only
    if ((p_bas_c_evt->conn_handle != m_conn_handle_playbulb_client) ||
        (m_conn_handle_playbulb_client == NULL))
    {
        return;
    }

    switch (p_bas_c_evt->evt_type)
    {
        case BLE_BAS_C_EVT_DISCOVERY_COMPLETE:
            err_code = ble_bas_c_handles_assign(p_bas_c,
                                                p_bas_c_evt->conn_handle,
                                                &p_bas_c_evt->params.bas_db);
            APP_ERROR_CHECK(err_code);

            // Battery service discovered. Enable notification of Battery Level.
            NRF_LOG_DEBUG("Battery Service discovered on Playbulb. Reading battery level...");

            err_code = ble_bas_c_bl_read(p_bas_c);
            APP_ERROR_CHECK(err_code);

            NRF_LOG_DEBUG("Enabling Battery Level Notification on Playbulb. ");
            err_code = ble_bas_c_bl_notif_enable(p_bas_c);
            APP_ERROR_CHECK(err_code);
            break;

        case BLE_BAS_C_EVT_BATT_NOTIFICATION:
            NRF_LOG_DEBUG("Battery Level received from Playbulb %d %%", p_bas_c_evt-
>params.battery_level);
            m_playbulb_battery_level = p_bas_c_evt->params.battery_level;
            send_playbulb_battery_level_to_client(m_playbulb_battery_level);
            break;

        case BLE_BAS_C_EVT_BATT_READ_RESP:
            NRF_LOG_INFO("Battery Level of Playbulb Read as %d %%", p_bas_c_evt->params.battery_level);
            m_playbulb_battery_level = p_bas_c_evt->params.battery_level;
            send_playbulb_battery_level_to_client(m_playbulb_battery_level);
            break;
    }
}
```

```

        default:
            break;
    }
}

```

- The following function serves as the event handler for events received from the Thingy Client module. The Thingy Client module handles searching for specific Services on the Thingy that we're interested in (in this case, it is only the Thingy Environment Service which includes Characteristics for Humidity and Temperature readings).

In this function, we:

- Handle the Discovery Complete event which indicates that we found the Thingy Environment Service
- Call the function for storing the Handles of the different Characteristics and Properties we're interested in (*described later when we look at the Thingy Client module*)
- Enable Notifications on both the Humidity Level and Temperature Level Characteristics
- In case of a received Notification for the Temperature and Humidity reading we send it to any subscribed Clients

```

/**@brief Handles events coming from the Thingy central module.
 */
static void thingy_c_evt_handler(thingy_client_t * p_thingy_c, thingy_client_evt_t * p_thingy_c_evt)
{
    switch (p_thingy_c_evt->evt_type)
    {
        case THINGY_CLIENT_EVT_DISCOVERY_COMPLETE:
        {
            if (m_conn_handle_thingy_client == BLE_CONN_HANDLE_INVALID)
            {
                ret_code_t err_code;

                m_conn_handle_thingy_client = p_thingy_c_evt->conn_handle;
                NRF_LOG_INFO("Thingy Environment Service discovered on conn_handle 0x%x",
m_conn_handle_thingy_client);

                err_code = thingy_client_handles_assign(p_thingy_c,
                                                m_conn_handle_thingy_client,
                                                &p_thingy_c_evt->params.peer_db);
                APP_ERROR_CHECK(err_code);

                // Environment service discovered. Enable notification of Temperature and Humidity
                // readings.
                err_code = thingy_client_temp_notify_enable(p_thingy_c);
                APP_ERROR_CHECK(err_code);

                err_code = thingy_client_humidity_notify_enable(p_thingy_c);
                APP_ERROR_CHECK(err_code);
            }
            break; // THINGY_CLIENT_EVT_DISCOVERY_COMPLETE
        }
    }
}

```

```

case THINGY_CLIENT_EVT_TEMP_NOTIFICATION:
{
    ret_code_t err_code;

    NRF_LOG_INFO("Temperature = %d.%d Celsius", p_thingy_c_evt->params.temp.temp_integer,
p_thingy_c_evt->params.temp.temp_decimal);

    // Send value to the Client device
    err_code = send_temperature_to_client(p_thingy_c_evt->params.temp.temp_integer);
    if ((err_code != NRF_SUCCESS) &&
        (err_code != NRF_ERROR_INVALID_STATE) &&
        (err_code != NRF_ERROR_RESOURCES) &&
        (err_code != BLE_ERROR_GATTS_SYS_ATTR_MISSING)
        )
    {
        APP_ERROR_HANDLER(err_code);
    }
} break; // THINGY_CLIENT_EVT_TEMP_NOTIFICATION

case THINGY_CLIENT_EVT_HUMIDITY_NOTIFICATION:
{
    ret_code_t err_code;

    NRF_LOG_INFO("Humidity percentage = %d %%", p_thingy_c_evt->params.humidity.humidity);

    // Send value to the Client device
    err_code = send_humidity_to_client(p_thingy_c_evt->params.humidity.humidity);
    if ((err_code != NRF_SUCCESS) &&
        (err_code != NRF_ERROR_INVALID_STATE) &&
        (err_code != NRF_ERROR_RESOURCES) &&
        (err_code != BLE_ERROR_GATTS_SYS_ATTR_MISSING)
        )
    {
        APP_ERROR_HANDLER(err_code);
    }
} break; // THINGY_CLIENT_EVT_HUMIDITY_NOTIFICATION

default:
    // No implementation needed.
    break;
}
}

```

- The following function is very similar to the above function except it handles events related to the Remote Control Peripheral.

In this function, we:

- In the case of a Notification event for the ON Button press, we send a Command to the Playbulb to turn ON the light
- In the case of a Notification event for the OFF Button press, we send a Command to the Playbulb to turn OFF

the light

```
/**@brief Handles events coming from the Remote Control central module.
 */
static void remote_control_c_evt_handler(remote_control_client_t * p_remote_control_c,
remote_control_client_evt_t * p_remote_control_c_evt)
{
    switch (p_remote_control_c_evt->evt_type)
    {
        case REMOTE_CONTROL_CLIENT_EVT_DISCOVERY_COMPLETE:
        {
            if (m_conn_handle_remote_control_client == BLE_CONN_HANDLE_INVALID)
            {
                ret_code_t err_code;

                m_conn_handle_remote_control_client = p_remote_control_c_evt->conn_handle;
                NRF_LOG_INFO("Remote Control Button Service discovered on conn_handle 0x%x",
m_conn_handle_remote_control_client);

                err_code = remote_control_client_handles_assign(p_remote_control_c,
                                                    m_conn_handle_remote_control_client,
                                                    &p_remote_control_c_evt->params.peer_db);
                APP_ERROR_CHECK(err_code);

                // Button service discovered. Enable notification of ON and OFF Buttons readings.
                err_code = remote_control_client_on_button_notify_enable(p_remote_control_c);
                APP_ERROR_CHECK(err_code);

                err_code = remote_control_client_off_button_notify_enable(p_remote_control_c);
                APP_ERROR_CHECK(err_code);
            }
            break; // REMOTE_CONTROL_CLIENT_EVT_DISCOVERY_COMPLETE
        }

        case REMOTE_CONTROL_EVT_ON_BUTTON_PRESS_NOTIFICATION:
        {
            ret_code_t err_code;

            NRF_LOG_INFO("ON Button = %s", p_remote_control_c_evt->params.on_button.button_pressed == 1?
"Pressed":"Released");

            // Send command to turn on Playbulb candle when ON Button is pressed
            if (p_remote_control_c_evt->params.on_button.button_pressed == 1 &&
(m_playbulb_client.conn_handle != BLE_CONN_HANDLE_INVALID))
            {
                err_code = playbulb_client_turn_on(&m_playbulb_client);
                APP_ERROR_CHECK(err_code);

                set_playbulb_light_status(1);
            }
            break; // REMOTE_CONTROL_EVT_ON_BUTTON_PRESS_NOTIFICATION
        }

        case REMOTE_CONTROL_EVT_OFF_BUTTON_PRESS_NOTIFICATION:
        {

```

```

    ret_code_t err_code;

    NRF_LOG_INFO("OFF Button = %s", p_remote_control_c_evt->params.off_button.button_pressed ==
1? "Pressed":"Released");

    // Send command to turn OFF Playbulb candle when OFF Button is pressed
    if (p_remote_control_c_evt->params.off_button.button_pressed == 1 &&
(m_playbulb_client.conn_handle != BLE_CONN_HANDLE_INVALID))
    {
        err_code = playbulb_client_turn_off(&m_playbulb_client);
        APP_ERROR_CHECK(err_code);

        set_playbulb_light_status(0);
    }
} break; // REMOTE_CONTROL_EVT_OFF_BUTTON_PRESS_NOTIFICATION

default:
    // No implementation needed.
    break;
}
}

```

- The following function handles events coming from the Playbulb Client module. Specifically, we are only interested in the event indicating that we've discovered the Playbulb Light Service.

```

/**@brief Handles events coming from the Playbulb Client module.
 */
static void playbulb_c_evt_handler(playbulb_client_t * p_playbulb_c, playbulb_client_evt_t *
p_playbulb_c_evt)
{
    switch (p_playbulb_c_evt->evt_type)
    {
        case PLAYBULB_CLIENT_EVT_DISCOVERY_COMPLETE:
        {
            if (m_conn_handle_playbulb_client == BLE_CONN_HANDLE_INVALID)
            {
                ret_code_t err_code;

                m_conn_handle_playbulb_client = p_playbulb_c_evt->conn_handle;
                NRF_LOG_INFO("Playbulb Light Service discovered on conn_handle 0x%x",
m_conn_handle_playbulb_client);

                err_code = playbulb_client_handles_assign(p_playbulb_c,
                                              m_conn_handle_playbulb_client,
                                              &p_playbulb_c_evt->params.peer_db);
                APP_ERROR_CHECK(err_code);
            }
} break; // PLAYBULB_CLIENT_EVT_DISCOVERY_COMPLETE

default:
    // No implementation needed.
    break;
}

```

```

    }
}
```

- The following function initializes all the Client modules needed for discovering the Services and Characteristics for the different Peripherals (Thingy:52, Playbulb candlelight, and Remote Control).

```

/**@brief Central Clients initialization.
 */
void central_init(void)
{
    ret_code_t                  err_code;
    thingy_client_init_t        thingy_init_obj;
    remote_control_client_init_t remote_control_init_obj;
    playbulb_client_init_t      playbulb_init_obj;
    ble_bas_c_init_t            bas_c_init_thingy_obj;
    ble_bas_c_init_t            bas_c_init_remote_control_obj;
    ble_bas_c_init_t            bas_c_init_playbulb_obj;

    thingy_init_obj.evt_handler           = thingy_c_evt_handler;
    remote_control_init_obj.evt_handler   = remote_control_c_evt_handler;
    playbulb_init_obj.evt_handler         = playbulb_c_evt_handler;
    bas_c_init_thingy_obj.evt_handler     = bas_c_thingy_evt_handler;
    bas_c_init_remote_control_obj.evt_handler = bas_c_remote_control_evt_handler;
    bas_c_init_playbulb_obj.evt_handler   = bas_c_playbulb_evt_handler;

    // Initialize the different clients:
    NRF_LOG_INFO("Starting Central role\n");

    // Initialize the Thingy Client
    err_code = thingy_client_init(&m_thingy_client, &thingy_init_obj);
    APP_ERROR_CHECK(err_code);

    // Initialize the Remote Control Client
    err_code = remote_control_client_init(&m_remote_control_client, &remote_control_init_obj);
    APP_ERROR_CHECK(err_code);

    // Initialize the Playbulb Candle Client
    err_code = playbulb_client_init(&m_playbulb_client, &playbulb_init_obj);
    APP_ERROR_CHECK(err_code);

    // Initialize the Battery Service clients
    err_code = ble_bas_c_init(&m_bas_thingy_client, &bas_c_init_thingy_obj);
    APP_ERROR_CHECK(err_code);
    err_code = ble_bas_c_init(&m_bas_remote_control_client, &bas_c_init_remote_control_obj);
    APP_ERROR_CHECK(err_code);
    err_code = ble_bas_c_init(&m_bas_playbulb_client, &bas_c_init_playbulb_obj);
    APP_ERROR_CHECK(err_code);
}
```

- The following function handles all general BLE events relevant to the Central Role.

For all events, we pass on the event to each Client module event handler so it can process the event.

In the case of a Connection event, we start the Database Discovery process. We also set the appropriate LED status based on whether:

- We are connected to all the Peripherals of interest → Turn OFF the Scanning LED
- We have to keep looking for the remaining Peripherals we haven't connected to yet → We keep the Scanning LED ON

In case we get disconnected from a Peripheral, we need to set the corresponding connection handle to `BLE_CONN_HANDLE_INVALID`.

If we still have Peripherals to search for, then we start the Scanning operation. We also update the Scanning LED and turn it ON.

If we are not connected to any Peripheral, we need to turn OFF the Connected LED.

If we receive an Advertising Report event, then we need to parse it to see if we found any of the Peripherals of interest (Thingy:52, Playbulb, or Remote Control). If we find one of the Peripherals we're interested in connecting to, we initiate the connection with the default Connection parameters.

If we receive a Client Connection Timeout event (no packets received from the Peripheral within a specified timeout), we need to disconnect from the Peripheral. We also disconnect if the Peripheral fails to respond a Request in time.

```
/**@brief  Function for handling BLE events from central applications.
*
* @details This function parses scanning reports and initiates a connection to peripherals when a
*          target UUID is found. It updates the status of LEDs used to report central applications
*          activity.
*
* @param[in]  p_ble_evt  Bluetooth stack event.
*/
void on_ble_central_evt(ble_evt_t const * p_ble_evt)
{
    ret_code_t              err_code;
    ble_gap_evt_t const * p_gap_evt = &p_ble_evt->evt.gap_evt;

    // Call the event handlers for each of the clients
    thingy_client_on_ble_evt(p_ble_evt, &m_thingy_client);
    remote_control_client_on_ble_evt(p_ble_evt, &m_remote_control_client);
    playbulb_client_on_ble_evt(p_ble_evt, &m_playbulb_client);

    switch (p_ble_evt->header.evt_id)
    {
        // Upon connection, check which peripheral has connected (Thingy, Playbulb, or Remote Control),
initiate DB
        // discovery, update LEDs status and resume scanning if necessary.
        case BLE_GAP_EVT_CONNECTED:
```

```

{
    NRF_LOG_INFO("Central connected");
    // If no Thingy is currently connected, try to find them on this peripheral.
    if ( (m_conn_handle_thingy_client == BLE_CONN_HANDLE_INVALID)
        || (m_conn_handle_remote_control_client == BLE_CONN_HANDLE_INVALID)
        || (m_conn_handle_playbulb_client == BLE_CONN_HANDLE_INVALID))
    {
        NRF_LOG_INFO("Attempt to find Thingy, Playbulb or Remote Control on conn_handle 0x%x",
p_gap_evt->conn_handle);

        err_code = ble_db_discovery_start(&m_db_discovery[0], p_gap_evt->conn_handle);
        if (err_code == NRF_ERROR_BUSY)
        {
            err_code = ble_db_discovery_start(&m_db_discovery[1], p_gap_evt->conn_handle);
            if (err_code == NRF_ERROR_BUSY)
            {
                err_code = ble_db_discovery_start(&m_db_discovery[2], p_gap_evt->conn_handle);
                APP_ERROR_CHECK(err_code);
            }
            else
            {
                APP_ERROR_CHECK(err_code);
            }
        }
        else
        {
            APP_ERROR_CHECK(err_code);
        }
    }

    // Update LEDs status, and check if we should be looking for more peripherals to connect to.
    bsp_board_led_on(CENTRAL_CONNECTED_LED);
    if (ble_conn_state_central_conn_count() == NRF_SDH_BLE_CENTRAL_LINK_COUNT)
    {
        bsp_board_led_off(CENTRAL_SCANNING_LED);
    }
    else
    {
        // Resume scanning.
        bsp_board_led_on(CENTRAL_SCANNING_LED);
        scan_start();
    }
} break; // BLE_GAP_EVT_CONNECTED

// Upon disconnection, reset the connection handle of the peer which disconnected,
// update the LEDs status and start scanning again.
case BLE_GAP_EVT_DISCONNECTED:
{
    if (p_gap_evt->conn_handle == m_conn_handle_thingy_client)
    {
        NRF_LOG_INFO("Thingy client disconnected (reason: 0x%x)",
p_gap_evt->params.disconnected.reason);

        m_conn_handle_thingy_client = BLE_CONN_HANDLE_INVALID;
    }
}

```

```

    }

    if (p_gap_evt->conn_handle == m_conn_handle_remote_control_client)
    {
        NRF_LOG_INFO("Remote Control client disconnected (reason: 0x%x)",
                     p_gap_evt->params.disconnected.reason);

        m_conn_handle_remote_control_client = BLE_CONN_HANDLE_INVALID;
    }
    if (p_gap_evt->conn_handle == m_conn_handle_playbulb_client)
    {
        NRF_LOG_INFO("Playbulb client disconnected (reason: 0x%x)",
                     p_gap_evt->params.disconnected.reason);

        m_conn_handle_playbulb_client = BLE_CONN_HANDLE_INVALID;
    }

    if ((m_conn_handle_thingy_client == BLE_CONN_HANDLE_INVALID) ||
        (m_conn_handle_remote_control_client == BLE_CONN_HANDLE_INVALID) ||
        (m_conn_handle_playbulb_client == BLE_CONN_HANDLE_INVALID))
    {
        // Start scanning
        scan_start();

        // Update LEDs status.
        bsp_board_led_on(CENTRAL_SCANNING_LED);
    }

    if (ble_conn_state_central_conn_count() == 0)
    {
        bsp_board_led_off(CENTRAL_CONNECTED_LED);
    }
} break; // BLE_GAP_EVT_DISCONNECTED

case BLE_GAP_EVT_ADV_REPORT:
{
    int8_t index;

    // Find the devices of interest: Thingy:52, Playbulb Candle, Remote Control
    if ((index = find_adv_name(&p_gap_evt->params.adv_report, m_target_periph_names,
NUMBER_OF_TARGET_PERIPHERALS)) >= 0)
    {
        NRF_LOG_INFO("We found a device named: %s", m_target_periph_names[index]);

        // Initiate connection.
        err_code = sd_ble_gap_connect(&p_gap_evt->params.adv_report.peer_addr,
                                      &m_scan_params,
                                      &m_connection_param,
                                      APP_BLE_CONN_CFG_TAG);
        if (err_code != NRF_SUCCESS)
        {
            NRF_LOG_INFO("Connection Request Failed, reason %d", err_code);
        }
        else
        {

```

```

        NRF_LOG_INFO("Connection Request SUCCEEDED");
    }
}
else
{
    err_code = sd_ble_gap_scan_start(NULL, &m_scan_buffer);
    APP_ERROR_CHECK(err_code);
}
} break; // BLE_GAP_ADV_REPORT

case BLE_GAP_EVT_TIMEOUT:
{
    // We have not specified a timeout for scanning, so only connection attempts can timeout.
    if (p_gap_evt->params.timeout.src == BLE_GAP_TIMEOUT_SRC_CONN)
    {
        NRF_LOG_INFO("Connection Request timed out.");
    }
} break;

case BLE_GAP_EVT_CONN_PARAM_UPDATE_REQUEST:
{
    // Accept parameters requested by peer.
    err_code = sd_ble_gap_conn_param_update(p_gap_evt->conn_handle,
                                             &p_gap_evt->params.conn_param_update_request.conn_params);
    APP_ERROR_CHECK(err_code);
} break;

case BLE_GAP_EVT_PHY_UPDATE_REQUEST:
{
    NRF_LOG_DEBUG("PHY update request.");
    ble_gap_phys_t const phys =
    {
        .rx_phys = BLE_GAP_PHY_AUTO,
        .tx_phys = BLE_GAP_PHY_AUTO,
    };
    err_code = sd_ble_gap_phy_update(p_ble_evt->evt.gap_evt.conn_handle, &phys);
    APP_ERROR_CHECK(err_code);
} break;

case BLE_GATTC_EVT_TIMEOUT:
    // Disconnect on GATT Client timeout event.
    NRF_LOG_DEBUG("GATT Client Timeout.");
    err_code = sd_ble_gap_disconnect(p_ble_evt->evt.gattc_evt.conn_handle,
                                     BLE_HCI_REMOTE_USER_TERMINATED_CONNECTION);
    APP_ERROR_CHECK(err_code);
break;

case BLE_GATTS_EVT_TIMEOUT:
    // Disconnect on GATT Server timeout event.
    NRF_LOG_DEBUG("GATT Server Timeout.");
    err_code = sd_ble_gap_disconnect(p_ble_evt->evt.gatts_evt.conn_handle,
                                     BLE_HCI_REMOTE_USER_TERMINATED_CONNECTION);
    APP_ERROR_CHECK(err_code);
break;

```

```

    default:
        // No implementation needed.
        break;
    }
}

```

thingy_client.h

This file serves as the header file for the Thingy Client module.

- The following code defines an enumeration for the different Client events such as Discovery of the Thingy Service and any Notifications.

```

/**@brief Thingy Client event type.*/
typedef enum
{
    THINGY_CLIENT_EVT_DISCOVERY_COMPLETE = 1,      /**< Event indicating that the Thingy Service has been
discovered at the peer. */
    THINGY_CLIENT_EVT_TEMP_NOTIFICATION,           /**< Event indicating that a notification of the Garage
Sensor Temperature characteristic has been received from the peer. */
    THINGY_CLIENT_EVT_HUMIDITY_NOTIFICATION       /**< Event indicating that a notification of the Garage
Sensor Humidity characteristic has been received from the peer. */
} thingy_client_evt_type_t;

```

- The following code defines the data structure that holds the Temperature reading value received from the Thingy:52. The different Thingy:52 services are defined by Nordic Semiconductor at [this link](#).

```

/**@brief Structure containing the temperature measurement received from the peer.*/
typedef struct
{
    int8_t  temp_integer;                                /**< Temperature (in Celsius) - Integer part.
*/
    uint8_t temp_decimal;                               /**< Temperature (in Celsius) - Decimal part.
*/
} thingy_temp_t;

```

- The following code defines the data structure that holds the Humidity level value received from the Thingy:52.

```

/**@brief Structure containing the humidity measurement received from the peer.*/
typedef struct
{
    uint8_t  humidity;                                  /**< Humidity Value. */

```

```
} thingy_humidity_t;
```

- The following data structure holds the handle values for the different Characteristics in the Thingy Environment Service that we're interested in. The CCCD handle is used for enabling/disabling Notifications.

```
/**@brief Structure containing the handles related to the Thingy found on the peer. */
typedef struct
{
    uint16_t temp_cccd_handle;      /**< Handle of the CCCD of the Temperature Measurement
characteristic. */
    uint16_t temp_handle;          /**< Handle of the Temperature Measurement characteristic as provided
by the SoftDevice. */
    uint16_t humidity_cccd_handle; /**< Handle of the CCCD of the Humidity Measurement characteristic.
*/
    uint16_t humidity_handle;      /**< Handle of the Humidity Measurement characteristic as provided by
the SoftDevice. */
} thingy_db_t;
```

- Next, we define a data structure to hold the different events.

```
/**@brief Thingy Event structure. */
typedef struct
{
    thingy_client_evt_type_t evt_type;    /**< Type of the event. */
    uint16_t conn_handle; /*< Connection handle on which the Thingy was discovered on
the peer device..*/
    union
    {
        thingy_db_t peer_db;           /**< Thingy Environment Service related handles found on the peer
device.. This will be filled if the evt_type is @ref THINGY_CLIENT_EVT_DISCOVERY_COMPLETE.*/
        thingy_temp_t temp;           /**< Temperature measurement received. This will be filled if the
evt_type is @ref THINGY_CLIENT_EVT_TEMP_NOTIFICATION.*/
        thingy_humidity_t humidity;   /**< Humidity measurement received. This will be filled if the
evt_type is @ref THINGY_CLIENT_EVT_HUMIDITY_NOTIFICATION.*/
    } params;
} thingy_client_evt_t;
```

- The following code defines the function prototype for the Thingy Client event handler. The application defines an event handler function that matches this prototype.

```
typedef struct thingy_client_s thingy_client_t;

/**@brief Event handler type.
*
* @details This is the type of the event handler that should be provided by the application
*          of this module in order to receive events.
*/
```

```
typedef void (* thingy_client_evt_handler_t) (thingy_client_t * p_thingy_client, thingy_client_evt_t * p_evt);
```

- The following code defines the main Thingy Client structure that holds context information for the Thingy Client module instance.

```
/**@brief Thingy Client structure.
 */
struct thingy_client_s
{
    uint16_t conn_handle;      /**< Connection handle as provided by the SoftDevice.*/
    thingy_db_t peer_thingy_db; /*< Handles related to Thingy Environment Service on
the peer*/
    ble_uuid_t service_uuid;   /**< Thingy Environment Service UUID */
    thingy_client_evt_handler_t evt_handler; /*< Application event handler to be called when there
is an event related to the Thingy Environment service. */
};
```

- The following code defines the Initialization data structure used by the application to define an event handler function for any Thingy Client module events.

```
/**@brief Thingy Client initialization structure.
 */
typedef struct
{
    thingy_client_evt_handler_t evt_handler; /*< Event handler to be called by the Thingy Client module
whenever there is an event related to the Thingy Service. */
} thingy_client_init_t;
```

- The following function is called to initialize the Thingy Client module.

```
/**@brief     Function for initializing the thingy client module.
 *
 * @details  This function will register with the DB Discovery module. There it
 *           registers for the Thingy Environment Service. Doing so will make the DB Discovery
 *           module look for the presence of a Thingy Environment Service instance at the peer when a
 *           discovery is started.
 *
 * @param[in] p_thingy_client     Pointer to the thingy client structure.
 * @param[in] p_thingy_client_init Pointer to the thingy initialization structure containing the
 *                               initialization information.
 *
 * @retval    NRF_SUCCESS On successful initialization. Otherwise an error code. This function
 *           propagates the error code returned by the Database Discovery module API
 *           @ref ble_db_discovery_evt_register.
 */

```

```
uint32_t thingy_client_init(thingy_client_t * p_thingy_client, thingy_client_init_t *
p_thingy_client_init);
```

- The following functions needs to be called by the application to pass on any BLE events to get processed by the Thingy Client module.

```
/**@brief      Function for handling BLE events from the SoftDevice.
*
* @details    This function will handle the BLE events received from the SoftDevice. If a BLE
*             event is relevant to the Thingy Client module, then it uses it to update
*             interval variables and, if necessary, send events to the application.
*
* @param[in]  p_ble_evt      Pointer to the BLE event.
* @param[in]  p_context      Pointer to the thingy client structure.
*/
void thingy_client_on_ble_evt(ble_evt_t const * p_ble_evt, void * p_context);
```

- The following function gets called by the application (in `central.c` to enable Notifications for the Temperature measurement Characteristic).

```
/**@brief      Function for requesting the peer to start sending notification of Temperature
*             Measurement.
*
* @details    This function will enable to notification of the Temperature Measurement at the peer
*             by writing to the CCCD of the Temperature Measurement Characteristic.
*
* @param      p_thingy_client Pointer to the thingy client structure.
*
* @retval    NRF_SUCCESS If the SoftDevice has been requested to write to the CCCD of the peer.
*             Otherwise, an error code. This function propagates the error code returned
*             by the SoftDevice API @ref sd_ble_gattc_write.
*/
uint32_t thingy_client_temp_notify_enable(thingy_client_t * p_thingy_client);
```

- The following function gets called by the application (in `central.c` to enable Notifications for the Humidity level Characteristic).

```
/**@brief      Function for requesting the peer to start sending notification of Humidity
*             Measurement.
*
* @details    This function will enable to notification of the Humidity Measurement at the peer
*             by writing to the CCCD of the Humidity Measurement Characteristic.
*
* @param      p_thingy_client Pointer to the thingy client structure.
*
* @retval    NRF_SUCCESS If the SoftDevice has been requested to write to the CCCD of the peer.
```

```

*
*           Otherwise, an error code. This function propagates the error code returned
*           by the SoftDevice API @ref sd_ble_gattc_write.
*/
uint32_t thingy_client_humidity_notify_enable(thingy_client_t * p_thingy_client);

```

- The following function handles the Database Discovery events and gets called from the application (in `central.c`) to pass on any events received from the Database Discovery module.

```

/**@brief      Function for handling events from the database discovery module.
*
* @details    Call this function when getting a callback event from the DB discovery module.
*             This function will handle an event from the database discovery module, and determine
*             if it relates to the discovery of Environment service at the peer. If so, it will
*             call the application's event handler indicating that the Temperature service has been
*             discovered at the peer. It also populates the event with the service related
*             information before providing it to the application.
*
* @param[in]  p_thingy_client Pointer to the heart rate client structure instance to associate.
* @param[in]  p_evt Pointer to the event received from the database discovery module.
*
*/
void thingy_on_db_disc_evt(thingy_client_t * p_thingy_client, const ble_db_discovery_evt_t * p_evt);

```

- The final function declared here assigns the different Handles for the Attributes (Services, Characteristics, CCCD) discovered by the Database Discovery module.

```

/**@brief      Function for assigning handles to this instance of thingy_client.
*
* @details    Call this function when a link has been established with a peer to
*             associate this link to this instance of the module. This makes it
*             possible to handle several link and associate each link to a particular
*             instance of this module. The connection handle and attribute handles will be
*             provided from the discovery event @ref BLE_HRS_C_EVT_DISCOVERY_COMPLETE.
*
* @param[in]  p_thingy_client      Pointer to the heart rate client structure instance to associate.
* @param[in]  conn_handle          Connection handle to associated with the given Thingy Client
*                                Instance.
* @param[in]  p_peer_thingy_handles Attribute handles for the Thingy server you want this Thingy client
*                                to
*                                interact with.
*/
uint32_t thingy_client_handles_assign(thingy_client_t * p_thingy_client,
                                      uint16_t conn_handle,
                                      const thingy_db_t * p_peer_thingy_handles);

#endif // THINGY_C_H__

```

thingy_client.c

This file is for the Thingy Client module that handles the different Attributes related to the Thingy Peripheral's GATT Server. It is used by the Central module to discover the Service and Characteristics of the Thingy after a connection is established with it.

- The following block of code is taken from the Nordic examples included in the nRF SDK. This is standard code that gets used in all Client modules.

```
#define TX_BUFFER_MASK          0x07           /**< TX Buffer mask, must be a mask of continuous
zeroes, followed by continuous sequence of ones: 000...111. */
#define TX_BUFFER_SIZE          (TX_BUFFER_MASK + 1)  /**< Size of send buffer, which is 1 higher than the
mask. */

#define WRITE_MESSAGE_LENGTH    BLE_CCCD_VALUE_LEN   /**< Length of the write message for CCCD. */
#define WRITE_MESSAGE_LENGTH    BLE_CCCD_VALUE_LEN   /**< Length of the write message for CCCD. */
```

- Next, we define the UUIDs for the Service and the Characteristics we're interested in (Thingy Environment Service, Humidity Level Characteristic, and Temperature reading Characteristic). Custom (vendor-specific) UUIDs are defined by using a **Base** UUID and then replacing the 3rd and 4th MSB within the Base UUID with the 2-byte UUID for the specific Attribute in question. Refer to [this Nordic article](#) for more on how this works.

```
// Thingy Services & Characteristics
// Base UUID for Enviornment service: EF68xxxx-9B35-4933-9B10-52FFA9740042
// Base UUID: 13BB0000-5884-4C5D-B75B-8768DE741149
#define BLE_UUID_ENVIRONMENT_SERVICE_BASE_UUID {0x42, 0x00, 0x74, 0xA9, 0xFF, 0x52, 0x10, 0x9B, \
                                             0x33, 0x49, 0x35, 0x9B, 0x00, 0x00, 0x68, 0xEF }

// Service & characteristics UUIDs
#define BLE_UUID_ENVIRONMENT_SERVICE_UUID     0x0200
#define BLE_UUID_TEMPERATURE_CHAR_UUID        0x0201
#define BLE_UUID_HUMIDITY_CHAR_UUID           0x0203
```

- The following block of code is also standard (taken from Nordic examples) for handling the transmission and receipt of data for an Attribute operation (Write, Read).

```
typedef enum
{
    READ_REQ,  /**< Type identifying that this tx_message is a read request. */
    WRITE_REQ  /**< Type identifying that this tx_message is a write request. */
} tx_request_t;

/**@brief Structure for writing a message to the peer, i.e. CCCD.
```

```

*/
typedef struct
{
    uint8_t                  gattc_value[WRITE_MESSAGE_LENGTH];  /**< The message to write. */
    ble_gattc_write_params_t gattc_params;                      /**< GATTC parameters for this message.
*/
} write_params_t;

/**@brief Structure for holding data to be transmitted to the connected central.
*/
typedef struct
{
    uint16_t      conn_handle;  /**< Connection handle to be used when transmitting this message. */
    tx_request_t type;        /**< Type of this message, i.e. read or write message. */
    union
    {
        uint16_t      read_handle;  /**< Read request message. */
        write_params_t write_req;   /**< Write request message. */
    } req;
} tx_message_t;

static tx_message_t m_tx_buffer[TX_BUFFER_SIZE];  /**< Transmit buffer for messages to be transmitted to
the central. */
static uint32_t      m_tx_insert_index = 0;          /**< Current index in the transmit buffer where the
next message should be inserted. */
static uint32_t      m_tx_index = 0;                 /**< Current index in the transmit buffer from where
the next message to be transmitted resides. */

/**@brief Function for passing any pending request from the buffer to the stack.
*/
static void tx_buffer_process(void)
{
    if (m_tx_index != m_tx_insert_index)
    {
        uint32_t err_code;

        if (m_tx_buffer[m_tx_index].type == READ_REQ)
        {
            err_code = sd_ble_gattc_read(m_tx_buffer[m_tx_index].conn_handle,
                                         m_tx_buffer[m_tx_index].req.read_handle,
                                         0);
        }
        else
        {
            err_code = sd_ble_gattc_write(m_tx_buffer[m_tx_index].conn_handle,
                                         &m_tx_buffer[m_tx_index].req.write_req.gattc_params);
        }
        if (err_code == NRF_SUCCESS)
        {
            m_tx_index++;
            m_tx_index &= TX_BUFFER_MASK;
        }
        else
        {

```

```

        NRF_LOG_DEBUG("SD Read/Write API returns error. This message sending will be "
                      "attempted again..");
    }
}
}

```

- The following function is responsible for handling any Write Response events that we receive.

```

/**@brief      Function for handling write response events.
 *
 * @param[in] p_thingy_client Pointer to the Thingy Client structure.
 * @param[in] p_ble_evt       Pointer to the BLE event received.
 */
static void on_write_rsp(thingy_client_t * p_thingy_client, const ble_evt_t * p_ble_evt)
{
    // Check if the event on the link for this instance
    if (p_thingy_client->conn_handle != p_ble_evt->evt.gattc_evt.conn_handle)
    {
        return;
    }
    // Check if there is any message to be sent across to the peer and send it.
    tx_buffer_process();
}

```

- The following function gets called when a Notification is received from the BLE stack (SoftDevice).

In it, we check if it's a Temperature reading Notification. If it is, then we parse the data to pass on to the application.

The Temperature reading Value includes two bytes: one for the Integer portion, and another for the Decimal portion. We store each of these values in their corresponding fields within the temperature data structure.

We also check if it's a Humidity level Notification. If it is, then we parse the data to pass on to the application. The Humidity level only contains one byte, which we store in the humidity field.

```

/**@brief      Function for handling Handle Value Notification received from the SoftDevice.
 *
 * @details    This function will use the Handle Value Notification received from the SoftDevice
 *             and checks if it is a notification of the Humidity level or Temperature reading from the
 *             peer. If
 *             it is, this function will decode value and send it to the application.
 *
 * @param[in] p_thingy_client Pointer to the Thingy Client structure.
 * @param[in] p_ble_evt       Pointer to the BLE event received.
 */
static void on_hvx(thingy_client_t * p_thingy_client, const ble_evt_t * p_ble_evt)
{
    // Check if the event is on the link for this instance
}

```

```

if (p_thingy_client->conn_handle != p_ble_evt->evt.gattc_evt.conn_handle)
{
    return;
}

// Check if this is a temperature notification.
if (p_ble_evt->evt.gattc_evt.params.hvx.handle == p_thingy_client->peer_thingy_db.temp_handle)
{
    thingy_client_evt_t thingy_client_evt;

    thingy_client_evt.evt_type      = THINGY_CLIENT_EVT_TEMP_NOTIFICATION;
    thingy_client_evt.conn_handle   = p_thingy_client->conn_handle;

    thingy_client_evt.params.temp.temp_integer = p_ble_evt->evt.gattc_evt.params.hvx.data[0];
    thingy_client_evt.params.temp.temp_decimal = p_ble_evt->evt.gattc_evt.params.hvx.data[1];

    p_thingy_client->evt_handler(p_thingy_client, &thingy_client_evt);
}
else if (p_ble_evt->evt.gattc_evt.params.hvx.handle == p_thingy_client-
>peer_thingy_db.humidity_handle)
{
    thingy_client_evt_t thingy_client_evt;

    thingy_client_evt.evt_type      = THINGY_CLIENT_EVT_HUMIDITY_NOTIFICATION;
    thingy_client_evt.conn_handle   = p_thingy_client->conn_handle;

    thingy_client_evt.params.humidity.humidity = p_ble_evt->evt.gattc_evt.params.hvx.data[0];

    p_thingy_client->evt_handler(p_thingy_client, &thingy_client_evt);
}
}

```

- The following function is used for handling disconnection events.

```

/**@brief      Function for handling Disconnected event received from the SoftDevice.
*
* @details    This function check if the disconnect event is happening on the link
*             associated with the current instance of the module, if so it will set its
*             conn_handle to invalid.
*
* @param[in]  p_thingy_client Pointer to the Thingy Client structure.
* @param[in]  p_ble_evt       Pointer to the BLE event received.
*/
static void on_disconnected(thingy_client_t * p_thingy_client, const ble_evt_t * p_ble_evt)
{
    if (p_thingy_client->conn_handle == p_ble_evt->evt.gap_evt.conn_handle)
    {
        p_thingy_client->conn_handle          = BLE_CONN_HANDLE_INVALID;
        p_thingy_client->peer_thingy_db.temp_cccd_handle = BLE_GATT_HANDLE_INVALID;
        p_thingy_client->peer_thingy_db.temp_handle   = BLE_GATT_HANDLE_INVALID;
        p_thingy_client->peer_thingy_db.humidity_cccd_handle = BLE_GATT_HANDLE_INVALID;
        p_thingy_client->peer_thingy_db.humidity_handle     = BLE_GATT_HANDLE_INVALID;
    }
}

```

```

    }
}
```

- The following function handles the Database Discovery events. We check if the Thingy Environment Service was discovered. If we've discovered the Thingy Environment Service, then we need to search for the Characteristics we're interested in: Humidity level and Temperature reading Characteristics. We check if the discovered Characteristics' UUIDs match.

If we find the Temperature Characteristic, we store the different Handles associated with it.

If we find the Humidity Characteristic, we store the different Handles associated with it.

If the connection was assigned before the completion of the Database Discovery, then we check to see if the Handles are "invalid". If they are, we store the complete Handles data structure.

```

void thingy_on_db_disc_evt(thingy_client_t * p_thingy_client, const ble_db_discovery_evt_t * p_evt)
{
    if (p_evt->evt_type == BLE_DB_DISCOVERY_COMPLETE)
    {
        // Debug log
        NRF_LOG_INFO("BLE DB Discovery complete - Thingy Client. Discovered UUID:%d, Type: %d",
                     p_evt->params.discovered_db.srv_uuid.uuid, p_evt-
>params.discovered_db.srv_uuid.type);
    }

    // Check if the Thingy Environment Service was discovered.
    if (p_evt->evt_type == BLE_DB_DISCOVERY_COMPLETE &&
        p_evt->params.discovered_db.srv_uuid.uuid == BLE_UUID_ENVIRONMENT_SERVICE_UUID &&
        p_evt->params.discovered_db.srv_uuid.type == p_thingy_client->service_uuid.type)
    {
        uint32_t i;

        thingy_client_evt_t evt;

        evt.evt_type      = THINGY_CLIENT_EVT_DISCOVERY_COMPLETE;
        evt.conn_handle = p_evt->conn_handle;

        NRF_LOG_DEBUG("Thingy Environment Service discovered.");

        // Look for the characteristics of interest
        for (i = 0; i < p_evt->params.discovered_db.char_count; i++)
        {
            if (p_evt->params.discovered_db.characteristics[i].characteristic.uuid.uuid ==
                BLE_UUID_TEMPERATURE_CHAR_UUID)
            {
                // Found Temperature characteristic. Store CCCD handle and continue.
                evt.params.peer_db.temp_cccd_handle =
                    p_evt->params.discovered_db.characteristics[i].cccd_handle;
                evt.params.peer_db.temp_handle =
                    p_evt->params.discovered_db.characteristics[i].characteristic.handle_value;
            }
        }
    }
}
```

```

        continue;
    }
    if (p_evt->params.discovered_db.characteristics[i].characteristic.uuid.uuid ==
        BLE_UUID_HUMIDITY_CHAR_UUID)
    {
        // Found Humidity characteristic. Store CCCD handle and continue.
        evt.params.peer_db.humidity_cccd_handle =
            p_evt->params.discovered_db.characteristics[i].cccd_handle;
        evt.params.peer_db.humidity_handle =
            p_evt->params.discovered_db.characteristics[i].characteristic.handle_value;
        continue;
    }
}

//If the instance has been assigned prior to db_discovery, assign the db_handles
if (p_thingy_client->conn_handle != BLE_CONN_HANDLE_INVALID)
{
    if ((p_thingy_client->peer_thingy_db.temp_cccd_handle == BLE_GATT_HANDLE_INVALID)&&
        (p_thingy_client->peer_thingy_db.temp_handle == BLE_GATT_HANDLE_INVALID)&&
        (p_thingy_client->peer_thingy_db.humidity_cccd_handle == BLE_GATT_HANDLE_INVALID)&&
        (p_thingy_client->peer_thingy_db.humidity_handle == BLE_GATT_HANDLE_INVALID))
    {
        p_thingy_client->peer_thingy_db = evt.params.peer_db;
    }
}

p_thingy_client->evt_handler(p_thingy_client, &evt);
}
}

```

- The following function is responsible for initializing the Thingy Client module. In it, we initialize the Connection Handle.

In addition to that, we need to add the Thingy Environment Service Base UUID so that we can reference its Type later. This is a Nordic-specific implementation. The Type is assigned when adding the vendor-specific UUID by calling `sd_ble_uuid_vs_add`. We store this Type so we can check to see if it matches the Type for the Service when it gets discovered by the Database Discovery module later.

Once we are done, we then need to “register” the Thingy Environment Service with the Database Discovery module to let it know to search for it during the Discovery process.

```

uint32_t thingy_client_init(thingy_client_t * p_thingy_client, thingy_client_init_t *
p_thingy_client_init)
{
    ret_code_t err_code;

    VERIFY_PARAM_NOT_NULL(p_thingy_client);
    VERIFY_PARAM_NOT_NULL(p_thingy_client_init);

    // Initialize service structure

```

```

p_thingy_client->conn_handle = BLE_CONN_HANDLE_INVALID;

// Add service UUID
ble_uuid128_t base_uuid = {BLE_UUID_ENVIRONMENT_SERVICE_BASE_UUID};
err_code = sd_ble_uuid_vs_add(&base_uuid, &p_thingy_client->service_uuid.type);
if (err_code != NRF_SUCCESS)
{
    return err_code;
}

// Set up the UUID for the service (base + service-specific)
p_thingy_client->service_uuid.uuid = BLE_UUID_ENVIRONMENT_SERVICE_UUID;

p_thingy_client->evt_handler          = p_thingy_client_init->evt_handler;
p_thingy_client->conn_handle         = BLE_CONN_HANDLE_INVALID;
p_thingy_client->peer_thingy_db.temp_cccd_handle = BLE_GATT_HANDLE_INVALID;
p_thingy_client->peer_thingy_db.temp_handle   = BLE_GATT_HANDLE_INVALID;
p_thingy_client->peer_thingy_db.humidity_cccd_handle = BLE_GATT_HANDLE_INVALID;
p_thingy_client->peer_thingy_db.humidity_handle   = BLE_GATT_HANDLE_INVALID;

return ble_db_discovery_evt_register(&p_thingy_client->service_uuid);
}

```

- This function handles the BLE events in the context of the Thingy Client module. It needs to get called from the application (`central.c`) when a BLE event is received by the application (from the SoftDevice).

```

void thingy_client_on_ble_evt(ble_evt_t const * p_ble_evt, void * p_context)
{
    thingy_client_t * p_thingy_client = (thingy_client_t *)p_context;

    if ((p_thingy_client == NULL) || (p_ble_evt == NULL))
    {
        return;
    }

    switch (p_ble_evt->header.evt_id)
    {
        case BLE_GATTC_EVT_HVX:
            on_hvx(p_thingy_client, p_ble_evt);
            break;

        case BLE_GATTC_EVT_WRITE_RSP:
            on_write_rsp(p_thingy_client, p_ble_evt);
            break;

        case BLE_GAP_EVT_DISCONNECTED:
            on_disconnected(p_thingy_client, p_ble_evt);
            break;

        default:
            break;
    }
}

```

```
}
```

- This function handles configuring the CCCD of a certain Characteristic. It allows the caller to enable/disable Notifications/Indications. (This is standard code provided by Nordic in many of their examples).

```
/**@brief Function for creating a message for writing to the CCCD.
 */
static uint32_t cccd_configure(uint16_t conn_handle, uint16_t handle_cccd, bool enable)
{
    NRF_LOG_DEBUG("Configuring CCCD. CCCD Handle = %d, Connection Handle = %d",
                  handle_cccd, conn_handle);

    tx_message_t * p_msg;
    uint16_t      cccd_val = enable ? BLE_GATT_HVX_NOTIFICATION : 0;

    p_msg           = &m_tx_buffer[m_tx_insert_index++];
    m_tx_insert_index &= TX_BUFFER_MASK;

    p_msg->req.write_req.gattc_params.handle   = handle_cccd;
    p_msg->req.write_req.gattc_params.len       = WRITE_MESSAGE_LENGTH;
    p_msg->req.write_req.gattc_params.p_value  = p_msg->req.write_req.gattc_value;
    p_msg->req.write_req.gattc_params.offset    = 0;
    p_msg->req.write_req.gattc_params.write_op = BLE_GATT_OP_WRITE_REQ;
    p_msg->req.write_req.gattc_value[0]         = LSB_16(cccd_val);
    p_msg->req.write_req.gattc_value[1]         = MSB_16(cccd_val);
    p_msg->conn_handle                         = conn_handle;
    p_msg->type                                = WRITE_REQ;

    tx_buffer_process();
    return NRF_SUCCESS;
}
```

- The following function enables Notifications on the Thingy's Temperature reading Characteristic.

```
uint32_t thingy_client_temp_notify_enable(thingy_client_t * p_thingy_client)
{
    VERIFY_PARAM_NOT_NULL(p_thingy_client);

    NRF_LOG_INFO("Enabling notifications for Temperature readings from Thingy");

    return cccd_configure(p_thingy_client->conn_handle,
                          p_thingy_client->peer_thingy_db.temp_cccd_handle,
                          true);
}
```

- The following function enables Notifications on the Thingy's Humidity level Characteristic.

```

uint32_t thingy_client_humidity_notify_enable(thingy_client_t * p_thingy_client)
{
    VERIFY_PARAM_NOT_NULL(p_thingy_client);

    NRF_LOG_INFO("Enabling notifications for Humidity level from Thingy");

    return cccd_configure(p_thingy_client->conn_handle,
                          p_thingy_client->peer_thingy_db.humidity_cccd_handle,
                          true);
}

```

- The following function assigns the different Handles including the connection handle (Nordic-specific) and the Characteristic Handles (BLE).

```

uint32_t thingy_client_handles_assign(thingy_client_t * p_thingy_client,
                                      uint16_t conn_handle,
                                      const thingy_db_t * p_peer_thingy_handles)
{
    VERIFY_PARAM_NOT_NULL(p_thingy_client);

    p_thingy_client->conn_handle = conn_handle;
    if (p_peer_thingy_handles != NULL)
    {
        p_thingy_client->peer_thingy_db = *p_peer_thingy_handles;
    }
    return NRF_SUCCESS;
}

```

Using nRF Cloud as the Internet Gateway

Here, we'll go over how to connect the Gateway device from the Main Project to a Cloud service — specifically the nRF Cloud. To make it easier to follow, I've recorded a video detailing the steps to accomplish this.

The video is titled is included with the e-book and is titled "**BONUS Video #3 - nRF Cloud connectivity (GW to Internet).mp4**". You can also watch the video here:

[BONUS Video #3: nRF Cloud connectivity \(Gateway to Internet\)](#)

Glossary of Terms

2M PHY: a mode introduced in Bluetooth 5 in which the radio transmits data at a rate of 2 Megasymbols/second.

Active Eavesdropping: a special case of man-in-the-middle (MITM) attacks, in which the attacker makes independent connections with the victims and relays messages between them to make them believe they are talking directly to each other over a private connection, when in fact the entire conversation is controlled by the attacker.

Advertising: the link layer state in which the device is sending out advertising packets for other BLE to discover it.

AES CCM: an authenticated encryption algorithm designed to provide both authentication and confidentiality.

Application: the top level of the BLE architecture which implements the particular use case of the BLE device.

ATT (Attribute Protocol): a simple protocol for devices to list attributes that allow different operations such as write and read.

Attribute: a generic term for any type of data exposed by a BLE server. It also defines the structure of this data.

Authentication: the process or action of verifying the identity of a user, device, or process.

Auxiliary Packets: this is the same as the secondary advertisement packets.

Beacon: a BLE device that broadcasts advertisement packets for the sole purpose of being discovered by other BLE devices. Usually, a beacon does not enter the connected state.

BLE: Bluetooth Low Energy.

Bluetooth Classic: a short range wireless technology focused on streaming applications such as audio streaming.

Bluetooth Lightbulb: a lightbulb that contains a BLE radio and acts as a BLE peripheral allowing other BLE devices to interact with it (defined in chapter 6: "GATT Design Exercise").

Bluetooth Low Energy: a short range wireless technology focused on low-power and low-bandwidth applications.

Bluetooth mesh: a new Bluetooth specification that builds on top of BLE and allows BLE devices to form a many-to-many network topology.

Bluetooth stack: software that refers to an implementation of the Bluetooth protocol.

Bonding: a process where the information from the pairing process is stored on the devices so that the pairing process does not have to be repeated every time the devices reconnect to each other.

BR/EDR: Basic Rate/Enhanced Data Rate (Bluetooth Classic).

Broadcaster: a BLE device that sends out advertising packets and does not allow other BLE devices to connect to it.

Central: a BLE device (usually a smartphone/tablet/PC) that listens for peripheral devices that are advertising. It is also capable of connecting to peripherals, and is responsible for managing the connection via its various parameters.

Channel Hopping: the act of rapidly switching between different frequencies, using a pseudorandom frequency-selection algorithm agreed on by the transmitter and receiver.

Characteristic: a container for a piece of user data, usually coupled with metadata describing it (such as being writeable, readable, its description, etc.).

Coded PHY: a mode introduced in Bluetooth 5 in which the radio transmits data at the original rate of 1 Megasymbols/second, while adding data redundancy (multiple symbols per bit of data) as a method to increase error detection and recovery at the receiving end.

Commissioning: the process by which a device (which is installed, is complete, or near completion) is tested to verify if it functions according to its design objectives or specifications.

Confidentiality: the state of keeping or being kept secret or private.

Configuration: the process by which a device's behavior is altered via one or more parameters.

Connection: the link layer state in which the device is connected to another BLE device, exchanging data regularly with this other device.

Controller: the lower layers of the BLE architecture. It is primarily responsible for interfacing with the radio and provides a standard interface for the host.

CSRK (Connection Signature Resolving Key): a security key used to sign transmitted data and to verify signatures on received data.

Data Length Extension: a feature that allows a BLE device to send packets with payloads of up to 251 bytes of application data, while in the connected state.

Device Address: a 48-bit number that identifies a BLE device.

DTM (Direct Test Mode): a mode for performing RF tests, used during manufacturing and for certification tests of BLE devices.

EDIV (Encrypted Diversifier): a value used along with the RAND (Random Number) in the process of creating and identifying the LTK (Long Term Key).

Embedded device: a device that contains a special-purpose computing system, usually without a (or with a minimal) user interface.

Encryption: the process of converting information or data into a code, especially to prevent unauthorized access.

Environment Sensor: a BLE peripheral device that collects data about the surrounding environment via a set of embedded sensors (defined in chapter 6: "GATT Design Exercise").

Extended Advertisements: a feature introduced in Bluetooth 5 by which a BLE device sends out advertising packets on the secondary channels.

FHSS (Frequency Hopping Spread Spectrum): a method of transmitting radio signals by rapidly switching between different frequencies, using a pseudorandom frequency-selection algorithm agreed on by the transmitter and receiver.

Filter Policy: a set of rules that defines which devices are approved or unapproved in an operation.

GAP (Generic Access Profile): the layer responsible for managing connections, advertisements, discovery and security features.

Gateway: a device that acts as both a BLE peripheral and a central connecting to multiple other peripheral devices (defined in chapter 6: "GATT Design Exercise").

GATT (Generic Attribute Profile): a basic data model that allows devices to discover, write, and read elements.

HCI (Host Controller Interface): the standard protocol that allows a controller to interface with a Host (within the same chipset or across different ones).

Hop Increment: an integer value that defines the increment applied to a channel number to indicate the next channel to be used during data transmission.

Host: a collection of the upper layers of the BLE architecture. It is responsible for providing an interface to the application layer for interacting with the controller.

Identity Tracking: the act of tracking a device by a known identifier or address of that device.

IEEE (Institute of Electrical and Electronics Engineers): an association dedicated to serving professionals involved in all aspects of the electrical, electronic, and computing fields and related areas of science and technology.

Initiating: the link layer state in which the device is listening for advertising packets from a specific device(s) and responding to these packets to initiate a connection with another device.

Integrity: internal consistency or lack of corruption in digital data.

IO (Input Output): the communication between an information processing system, such as a computer, and the outside world (possibly a human or another information processing system).

IoT (Internet of Things): the network of physical devices, vehicles, home appliances, and other items embedded with electronics, software, sensors, actuators, and connectivity, enabling these devices to connect and exchange data.

IRK (Identity Resolving Key): a security key used to resolve BLE random private addresses.

ISM band: radio bands (portions of the radio spectrum) reserved internationally for the use of radio frequency (RF) energy for industrial, scientific and medical purposes other than telecommunications.

L2CAP (Logical Link Control and Adaptation Control): the layer within the BLE architecture that acts as a protocol-multiplexing layer and handles fragmentation and recombination of packets.

LE Legacy Connections: the security features originally defined by the first BLE specification (Bluetooth version 4.0).

Legacy Advertisements: advertisement packets that exist in all versions of BLE and are sent on the primary advertising channels (channels 37, 38, and 39).

LESC (LE Secure Connections): an enhanced security feature introduced in Bluetooth version 4.2. It uses a Federal Information Processing Standards (FIPS) compliant algorithm called Elliptic Curve Diffie Hellman (ECDH) for key generation.

Link Layer: a layer within the BLE architecture that defines the different states a BLE device can operate in.

Low-bandwidth data transfer: transmission of small amounts of data that does not utilize the full capacity of a network connection.

Master: a role within the link layer that initiates a connection and controls the timings of data transmissions.

Message Integrity: the validity of a transmitted message, indicating that the message has not been tampered with or altered.

MITM (man-in-the-middle) attack: the act of secretly relaying and possibly altering the communication between two parties who believe they are directly communicating with each other.

MTU (Maximum Transmission Unit): the size of the largest protocol data unit (PDU) that can be communicated in a single network unit (packet).

Network Topology: the way in which different elements in a network are interrelated or arranged.

Observer: a BLE device that scans for advertising BLE devices, but is not capable of connecting to these devices.

One-to-many: a network topology where one device connects with multiple devices at the same time.

One-to-one: a network topology where two devices communicate directly with each other.

OOB (Out-of-Band): refers to a communication medium other than BLE, used for exchanging security keys between two BLE devices to secure the communication channel.

Packet: a formatted unit of data carried by a network. A packet consists of control information and user data, which is also known as the payload.

Pairing: the process by which two BLE devices exchange device information so that a secure link can be established.

Passive Eavesdropping: secretly or stealthily listening to the private conversation or communications of others without their consent.

Payload: the user data portion of a packet.

PDU (Protocol Data Unit): information that is transmitted as a single unit among peer entities in a network.

Periodic Advertisements: a special case of extended advertisements that allows a central to synchronize to a peripheral that is sending extended advertisements at a fixed interval.

Peripheral: a BLE device that sends out advertising packets and allows other BLE devices (specifically Centrals) to connect to it.

PHY (Physical Layer): the layer that represents the physical circuitry responsible for transmitting and receiving radio packets.

Primary Advertisements: advertising packets that are sent on one of the primary advertising channels in BLE (channels 37, 38, and 39).

Primary Service: a BLE service that provides the primary functionality of a device.

Privacy: the state or condition of being free from being observed or disturbed by other entities.

Public Address: a factory-programmed device address that does not change and must be registered with the IEEE.

Publish-Subscribe: a messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers, but instead categorize published messages into classes without knowledge of which subscribers, if any, there may be.

Rand (Random Number): a random number used along with the EDIV (Encrypted Diversifier) in the process of creating and identifying the LTK (Long Term Key).

Random Address: a device address that is programmed on the device or generated at runtime.

Remote Control: a BLE peripheral device that's used to control a Bluetooth lightbulb via button presses (defined in chapter 6: "GATT Design Exercise").

Replay attack: an attack that occurs when one or more messages are stored and replayed later by a malicious device.

Scanning: the link layer state in which the device is looking for BLE devices that are in the advertising state.

Secondary Advertisements: advertising packets that are sent on one of the secondary advertising channels in BLE (channels 1-36).

Secondary Service: a BLE service that adds auxiliary functionality to a device and is referenced from at least one primary service.

Security Key: the alphanumeric key that's used to decrypt or encrypt data which could be exchanged between two BLE devices.

Self-healing: a network in which devices can still communicate with each other if one of the nodes drops off the network.

Service: a collection of one or more characteristics and the relationships to other services, representing specific functionality of a device.

Slave: a role within the link layer that accepts a connection from a master and abides to its timing requirements.

SM (Security Manager): the layer within the BLE architecture that defines the methods of pairing and key distribution between two BLE devices.

Standby: the link layer state in which the device is not sending any BLE packets (not advertising, scanning, initiating, or connected to another BLE device).

STK (Short Term Key): a security key used in LE Legacy Connections during the pairing phase to encrypt the communication between two BLE devices.

TK (Temporary Key): a security key used in LE Legacy Connections during the pairing phase to create the Short Term Key (STK).

UUID (Universally Unique Identifier): a 128-bit number used to uniquely identify a device in a network, or entity within a device.

White List: a list containing information about devices that are viewed with *approval* within a network.

References and Resources

- [Official Bluetooth website](#)
- [Official Bluetooth Core Specification documents](#)
- [Nordic InfoCenter](#)
- [Nordic DevZone](#)
- [Nordic S140 SoftDevice Specification document](#)
- [iOS Bluetooth Guidelines](#)
- [Blog post: Maximizing BLE Throughput on iOS and Android](#)
- [Blog post: Maximizing BLE Throughput Part 2: Use Larger ATT MTU](#)
- [Book: "Intro to Bluetooth Low Energy" by Mohammad Afaneh. Copyright 2018 Novel Bits, LLC.](#)
- [Book: "Getting Started with Bluetooth Low Energy by Kevin Townsend, Carles Cufi, Akiba, and Robert Davidson \(O'Reilly\). Copyright 2014 Kevin Townsend, Carles Cufi, Akiba, and Robert Davidson, 978-1-491-94951-1"](#)
- [Book: "Bluetooth low energy: the developer's handbook by Robin Heydon. Copyright 2013 Pearson Education, 978-0-13-288836-3"](#)
- [Six hidden costs in a 99 cent Wireless SoC](#)

