Creating *Bluetooth*® Low Energy Applications Using nRF51822

nAN-36

Application Note v1.1

This application note is intended for anyone who would like to begin programming *Bluetooth* low energy (BLE) applications on nRF51822. It consists of a general overview of BLE functionality and is followed by a description of a simple example program that implements a custom service, the LED Button service.

1 Introduction

The purpose of this application note is to show you the steps necessary for creating your own BLE application, including a custom service, using the nRF51822 chip.

1.1 Minimum requirements

Experience in embedded C programming is needed to fully understand this application note.

1.1.1 Required tools

The nRF51822 Evaluation Kit is needed for this application note. Additionally, the following tools should be downloaded and installed:

- S110 SoftDevice
- nRFgo Studio
- nRF51 SDK
- Keil MDK-ARM
- SEGGER's J-Link tools

Program the S110 SoftDevice onto the chip using the instructions in the *nRF51822 Evaluation Kit User Guide*.

Note: nRF51 SDK version 5.2.0 and S110 SoftDevice version 6.0.0 were the latest versions and the ones used as reference when this document was written.

1.2 Documentation

The following documentation is important reference material:

Document	Description
nRF51822 Evaluation Kit User Guide	Contains information about setting up and using the Evaluation Kit, including setting up Keil and the SoftDevice.
nRF51 SDK documentation	This is available in the Documentation subfolder of the SDK installation folder, contains API documentation for all functions in the SDK and also from the Nordic Developer Zone.
S110 nRF51822 SoftDevice Specification	Contains information about the S110 SoftDevice, including resource usage and high-level functionality.
nRF51822 Product Specification	Contains a description of the hardware, modules, and electrical specifications specific to the nRF51822 chip.
nRF51 Series Reference Manual	Contains a functional description of all the modules and peripherals supported for all the chips in the nRF51 series.
nAN-15: Creating Applications with the Keil C51 Compiler	This application note contains information about using Keil µVision. It was originally written for the nRF24LE1 chip, but Section 3.3 "Including files" and Section 3.4 "Debug your project" are also relevant for the nRF51822 chip.
Bluetooth Core Specification, version 4.0, Volumes 1, 3, 4, and 6	Provided by the *Bluetooth* SIG, this document contains information relating to *Bluetooth* services and profiles.

1.3 *Bluetooth* resources

All *Bluetooth* SIG services, characteristics, and descriptors are defined on the *Bluetooth* Developer Portal which can be used as a reference for finding UUIDs or data formats used by different parts of the specification.

1.4 nRF51822 and the S110 SoftDevice

The S110 SoftDevice is a BLE Peripheral protocol stack solution. It integrates a low energy controller and host, and provides a full and flexible API for building *Bluetooth* low energy System on Chip (SoC) solutions. The S110 SoftDevice is provided as a precompiled HEX file that must be programmed onto the chip before your application is loaded.

The SoftDevice uses a portion of the chip's flash and RAM, but they are protected from your code, so that you can't accidentally write to these areas. The SoftDevice also needs exclusive access to some peripherals and registers.

For information on how to program the SoftDevice into the nRF51822 chip, see the *nRF51822 Evaluation Kit User Guide*. For information on which resources the SoftDevice uses, see the *nRF51822 S110 SoftDevice Specification*.

1.5 Application overview

The LED Button application example was created in order to give you an environment where you can learn how to use *Bluetooth* low energy on the nRF51822 chip. It is a simple BLE application that demonstrates bi-directional communication over BLE. When it is running, you will be able to toggle an LED output on the nRF51822 chip from the Central (see ***Section 2.1.1 "Roles"*** on page 5 for a definition of the Central), and receive a notification when button input on nRF51822 is pressed.

The application is implemented in one service (see ***Section 2.2.2 "GATT hierarchy"*** on page 7 for a description of a service and characteristics), with two characteristics: the LED characteristic, which can be used to control the LED remotely through the write without response operation, and the Button characteristic, which sends a notification when the button is pressed.

2 Introduction to *Bluetooth* low energy

This chapter describes the different layers of the BLE protocol, the components within these layers, and their concepts.

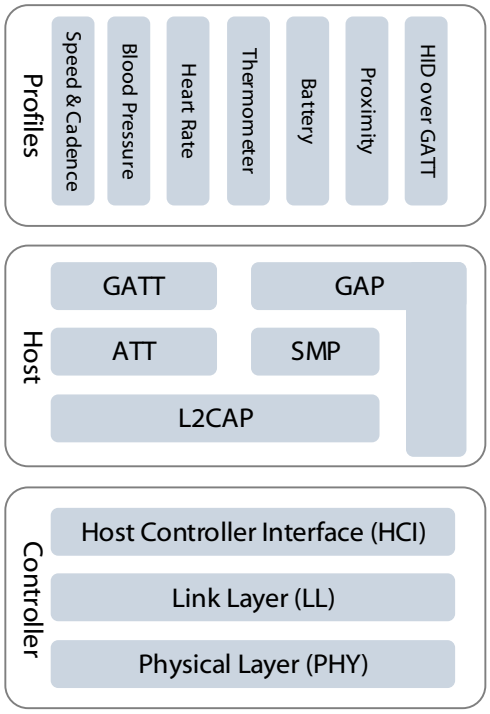


Figure 1 *Protocol stack components and layers*

2.1 Generic Access Profile (GAP)

GAP is the lowest layer of the *Bluetooth* stack that an application interfaces with. It includes parameters that govern advertising and connection among other things.

Note: GAP is covered in more detail in Volume 3, Part C of the *Bluetooth Core Specification*.

2.1.1 Roles

In creating and maintaining a BLE link, certain roles are involved. A BLE device is either a Central or a Peripheral, with the definition depending on the initiator of the link. The Central is always the device that initiates the connection, while the Peripheral is the device that is connected to. The terms Master and Slave are the Link Layer roles equivalent to Central and Peripheral.

In the LED Button application, the nRF51822 chip programmed with the S110 SoftDevice will be the Peripheral, and either a computer or phone will be the Central.

In addition to the Central and Peripheral roles, the *Bluetooth Core Specification* defines Observer and Broadcaster roles. Observers listen to what's happening on the air and Broadcasters send but don't receive

information. Both the Observer and Broadcaster roles only use advertising and never establish a connection. These are not applicable for our use case.

Note: The device on the other end of a link is often called a peer device, even if the device in question is a Central or Peripheral.

2.1.2 Advertising

For a Central to be able to connect to a Peripheral, the Peripheral must be advertising. It sends advertising packets with a time interval, known as the advertising interval, that ranges between 20 ms and 10.24 s. The advertising interval affects how long it takes to initiate a connection.

The Central must receive an advertising packet before it can send a connection request to initiate a connection. The Peripheral only listens for connection requests for a short while after sending an advertising packet.

An advertising packet can contain up to 31 bytes of data. It usually contains a user readable name, information about the device sending packets, some flags used to know whether the device is discoverable or not, and similar.

When a Central receives an advertising packet, it may send a request for more advertising data, called a Scan Request, if it is configured as an Active Scanner. A Peripheral responds to the request by sending a Scan Response that can contain an additional 31 bytes.

Advertising, including scan requests and responses, occurs on three frequencies spread over the 2.4 GHz band to avoid WLAN interference.

2.1.3 Scanning

Scanning is used by the Central to listen for advertising packets and to send scan requests. There are two timing parameters you need to be aware of in this context: scan window and scan interval.

For each scan interval, the Central scans for a time equal to the scan window, meaning that if the scan window is equal to the scan interval, the Central will do a continuous scan. The scan window divided by the scan interval is known as the scan duty cycle.

2.1.4 Initiating

When the Central wants to enter a connection, it will use the same procedure as when scanning to listen for advertising packets. When initiating, the Central will send a connection request to the Peripheral when it receives an advertising packet.

2.1.5 Connection

By definition the Central and Peripheral are in a connection from the first data exchange. When in a connection, the Central will request data from the Peripheral at specifically defined intervals. This interval is called the connection interval. It is decided and applied to the link by the Central, but a Peripheral can send Connection Parameter Update Requests to the Central. The connection interval must be between 7.5 ms and 4 s according to the *Bluetooth Core Specification*.

If the Peripheral doesn't respond to packages from the Central within the time frame, called connection supervision timeout, the link is considered lost.

It is possible to achieve higher data throughput by transmitting multiple packets in each connection interval. Each packet transferred can contain up to 20 bytes of usable application data. However, if current consumption is important and a Peripheral has no data to send, it can choose to ignore a certain number of intervals. The number of ignored intervals is called the slave latency.

While in a connection, the devices will hop through all the channels in the frequency band, except for the advertising channels, in a way that is completely transparent to the application.

2.2 Generic Attribute profile (GATT)

GATT is the layer where the data values are actually transferred.

2.2.1 Roles

In addition to the roles in GAP, BLE also defines two roles, a GATT Server and a GATT Client, that are completely independent of the GAP roles. The device holding data is the GATT Server, while the device accessing it is the GATT Client.

> For the LED Button example, the Peripheral device (with its LED and button) is in the role of Server and the Central is in the role of Client.

Note: It is possible for a device to be both a GATT Server and GATT Client simultaneously.

2.2.2 GATT hierarchy

A GATT Server organizes data in what is called an attribute table and it is the attributes that contain the actual data.

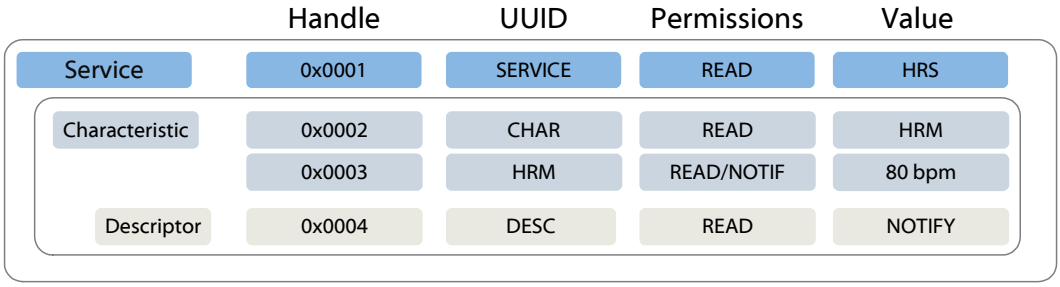


Figure 2 *GATT overview*

2.2.2.1 Attribute

An attribute has a handle, a UUID, and a value. A handle is the index in the GATT table for the attribute and is unique for each attribute in a device. The UUID contains information on the type of data within the attribute, which is key information for understanding the bytes that are contained in the value of the attribute. There may be many attributes in a GATT table with the same UUID.

2.2.2.2 Characteristic

A characteristic consists of at least two attributes: a characteristic declaration and an attribute that holds the value for the characteristic.

All data that will be transferred through a GATT Service must be mapped to a set of characteristics. It is a good idea to consider bundling the data up so that each characteristic is a self-contained, single instance data point. For example, if some pieces of data always change together, it will often make sense to collect them in one characteristic.

> In the LED Button service there is no relation between the LED and the button, and they can change independently from each other. Therefore, it makes sense to keep them as separate characteristics, so that we use one characteristic for the current button state and one characteristic for the current LED state.

2.2.2.3 Descriptors

Any attribute within a characteristic definition that is not the characteristic value, is by definition a descriptor. A descriptor is an additional attribute that provides more information about a characteristic, for instance a human-readable description of the characteristic.

However, there is one special descriptor that is worth mentioning in particular: the Client Characteristic Configuration Descriptor (CCCD). This descriptor is added for any characteristic that supports the Notify or Indicate properties, see ***Section 2.2.5 "On-air operations and properties"*** on page 10.

Writing a '1' to the CCCD enables notifications, while writing a '2' enables indications. Writing a '0' disables both notifications and indications.

For the S110 SoftDevice, this descriptor is added automatically for any characteristic where the Notify or Indicate properties are set.

2.2.2.4 Service

A service consists of one or more characteristics, and is a logical collection of related characteristics.

GATT services typically include pieces of related functionality - such as a particular sensor's readings and settings or the inputs and outputs of a Human Interface Device. Organizing related characteristics into services is both useful and practical, since it promotes a clearer separation based on logical and use-case defined boundaries and helps code reuse across different applications. The GATT-based SIG profiles and services make good use of this approach, and it is recommended to follow their strategy for user-defined profiles.

> For the LED Button example, there isn't any reuse concerns, so both the LED and the button characteristics are grouped into one service.

2.2.2.5 Profile

A profile can be defined to collect one or more services into a use case description. A profile document includes information on services that are required or optional for that particular profile as well as how the peers will interact with each other. This includes both which GAP and GATT Roles the devices will be in during data exchange. Therefore, this document will often contain information on what kind of advertising and connection intervals should be used, whether security is required, and similar.

It should be noted that a profile does not have an attribute in the attribute table.

> For the LED Button example, a profile is not formally described.

2.2.3 Standard versus custom services and characteristics

The *Bluetooth* SIG has defined a number of profiles, services, characteristics, and attributes based on the GATT layer of the stack. However, with *Bluetooth* low energy all service implementations are part of the application and not the stack, meaning it is possible for an application to support whichever profiles or services it wants to, as long as the stack supports GATT.

Because support for profiles and services is in the application, it is possible to create custom services in the application.

> For the LED Button example, there is no *Bluetooth* SIG service that covers this use case, so it will be implemented as a custom service, with two custom characteristics.

2.2.4 UUIDs

As stated in ***Section 2.2.2 "GATT hierarchy"*** on page 7, all attributes have a UUID. A UUID is a 128-bit number that is globally unique and is used to identify an attribute type.

2.2.4.1 Bluetooth SIG UUIDs

The *Bluetooth Core Specification* makes a distinction between a base UUID and a 16-bit UUID, which completes the base UUID.

All *Bluetooth* SIG defined UUIDs use a common base UUID, and more specifically the following one:

0x0000xxxx-0000-1000-8000-00805F9B34FB

To further refine this base UUID, each *Bluetooth* SIG defined attribute has a unique 16-bit UUID that replaces the x's in the common base UUID described above. The Heart Rate Measurement Characteristic, for example, has 0x2A37 as a 16-bit UUID, and therefore the full 128-bit UUID for the Heart Rate Measurement characteristic is:

0x00002A37-0000-1000-8000-00805F9B34FB

Since all *Bluetooth* SIG UUIDs use the same base UUID, the 16-bit UUID is sufficient to uniquely identify a *Bluetooth* SIG defined attribute.

The *Bluetooth* SIG base UUID cannot be used for any custom attributes, services, or characteristics. For all custom attributes, a full 128-bit UUID should be used.

2.2.4.2 Vendor specific UUIDs

The SoftDevice organizes UUIDs in a similar way to how the *Bluetooth Core Specification* defines the Bluetooth SIG UUIDS, that is, you add a custom base UUID and then define 16-bit numbers, similar to the aliases, to be used on top of this base. It's easiest to use one base UUID for all custom attributes, at least within the same service.

A base UUID is easily generated using nRFgo Studio to create new UUIDs, see ***Section 4.4.3 "Service Initialization"*** on page 21.

> For the LED Button example, 0x0000xxxx-1212-EFDE-1523-785FEABCD123 will be used as the base.

The *Bluetooth Core Specification* does not include any rules or recommendations for how to assign the unique 16-bit UUIDs that are added to the base UUID, so you can use any scheme you want for this.

> For the LED Button example, 0x1523 is used for the service, 0x1524 for the LED characteristic, and 0x1525 for the button state characteristic.

2.2.5 On-air operations and properties

Most on-air operations happen by using the handle, since this uniquely identifies each attribute.

Use of the characteristic is dependent on its properties. Characteristic properties include:

- Write
- Write without response
- Read
- Notify
- Indicate

More properties are defined in the *Bluetooth* specification, but these are the most commonly used.

2.2.5.1 Write and Write without response

Write and Write without response allow the GATT Client to write a value to a characteristic in a GATT Server. The difference between them is that Write without response happens without any application level acknowledgment or response.

2.2.5.2 Read

The Read property makes it possible for the GATT Client to read the value of a characteristic in a GATT Server.

2.2.5.3 Notify and Indicate

Notify and Indicate allow a GATT Server to make the GATT Client aware of changes to a characteristic. The difference between Notify and Indicate is that Indicate has application level acknowledgment, while Notify does not.

For the LED Button example, the characteristic to control the LED and the characteristic for the current button state are the two custom characteristics used in the LED Button service.

For the LED characteristic, the Central needs to be able to set its value, and possibly read it back. Since the application level acknowledgments aren't needed, you can use the Write without response and Read properties.

For the button characteristic, the Client needs to be notified when the button changes state, but application level acknowledgment isn't needed. Only the Notify property is needed for this.

Note: GATT and its underlying ATT protocol are described in the *Bluetooth Core Specification*, Volume 3, parts F and G.

3 Minimal BLE application overview

This chapter provides an overview of the minimum requirements for a BLE application on the nRF51822 device using the S110 SoftDevice.

3.1 Overview of initialization

There are several initialization calls that are commonly performed as part of a BLE application. A list of initialization calls is provided in the tables below with some explained in more detail later in the document.

Initialization call	Ways to achieve this	In LED Button demo app
Enable	• Use SDK wrapper macro `SOFTDEVICE_HANDLER_INIT` from `softdevice_handler.h` • `sd_softdevice_enable()` in `nrf_sdm.h`	`ble_stack_init()` in `main.c`
Add event handler	• Function passed to `SOFTDEVICE_HANDLER_INIT`	`ble_evt_dispatch()` in `main.c`

***Table 1** SoftDevice initialization calls*

Initialization call	Ways to achieve this	In LED Button demo app
Set device name	• `sd_ble_gap_device_name_set()` in `ble_gap.h`	`gap_params_init()` in `main.c`
Set up advertising data	• `ble_advdata_set()` in `ble_advdata.h` • `sd_ble_gap_adv_data_set` in `ble_gap.h`	`advertising_init()` in `main.c`
Connection parameters	• `ble_conn_params_init()` in `ble_conn_params.h` • `sd_ble_gap_ppcp_set()` in `ble_gap.h`	`conn_params_init()` in `main.c`

***Table 2** GAP initialization calls*

Initialization call	Ways to achieve this	In LED Button demo app
Add one or more services	• `sd_ble_gatts_service_add()` in `ble_gatts.h` (Most often not done by application itself, only by services.)	`ble_lbs_init()` in `ble_lbs.c.`
Add one or more characteristics	• `sd_ble_gatts_characteristic_add()` in `ble_gatts.h` (Most often not done by application itself, only by services.)	`ble_lbs_init()` in `ble_lbs.c.`

***Table 3** GATT initialization calls*

Initialization call	Ways to achieve this	In LED Button demo app
Start advertising	• `sd_ble_gap_adv_start()` in `ble_gap.h`	`advertising_start()` in `main.c`

***Table 4** App initialization calls*

Most of the methods described above use structures as input parameters. The structures specify a set of configurations or options and the comments in the code provide a better understanding of them.

You can enter the main loop after advertising has started.

3.2 S110 SoftDevice

You must enable the S110 SoftDevice in order to use it, which gives exclusive access to the radio peripheral. See the *S110 nRF51822 SoftDevice Specification* for details on hardware resource use.

3.3 Advertising

The structure types used for advertising are `ble_gap_conn_sec_mode_t` from the `ble_gap.h` header file, and `ble_advdata_t` from `ble_advdata.h`.

```
err_code = sd_ble_gap_device_name_set(&device_name_sec_mode, DEVICE_NAME
strlen(DEVICE_NAME));
err_code = sd_ble_gap_appearance_set(BLE_APPEARANCE_UNKNOWN);

err_code = ble_advdata_set(&advdata);
```

Note: The security mode passed to `sd_ble_gap_device_name_set()` applies only to the device name itself.

Advertising parameters (`ble_gap_adv_params_t`) need to be passed to `sd_ble_gap_adv_start()`:

```
err_code = sd_ble_gap_adv_start(&m_adv_params);
```

3.4 Connection parameters

The SDK comes with a module called `ble_conn_params` that manages connection parameter updates as needed. It wraps the SoftDevice APIs for handling this, including handling the timing of requests and sending new request if the first ones are rejected.

In the initialization structure `ble_conn_params_init_t`, different parameters for the update procedure can be set. For example, whether it should start on connection, start when writing to a specific CCCD, which connection parameters should be used, delays before requests are sent, and more.

The `ble_conn_params_init()` function then takes a `ble_conn_params_init_t` structure which wraps the initial connection parameters (`ble_gap_conn_params_t`) to be requested.

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

The `ble_conn_params` SDK module ensures that the connection parameters decided by the Master are acceptable. If they are not, it asks to change them. After a set number of attempts to change them without success, it disconnects or gives an event back to the application depending on the configuration.

3.5 Services

Services can be added with `sd_ble_gatts_service_add()`. You should not do this in application code but instead build services as separate files. A service can either be primary or secondary, but in practice mostly primary services are used in common applications. The `service_uuid` variable is the UUID you want to use for the service. The `service_handle` variable is an output variable, which will be filled with a unique handle for the service that is created. The handle can be used later to identify the service.

```
err_code = sd_ble_gatts_service_add(BLE_GATTS_SVC_TYPE_PRIMARY,
                        &p_lbs->service_uuid,
                        &p_lbs->service_handle );
```

3.6 Characteristics

Characteristics are added with `sd_ble_gatts_characteristic_add()`, which takes four arguments. For code clarity, this call should happen only in service files, not in the application itself.

The first argument is the handle of the service the characteristic should be added to. The second argument is a metadata structure for the characteristic, which has information about what properties are available (read, write, notification, and so on). The third argument is a description of the value attribute, which contains its UUID, length, and initial value. The final argument is filled with a unique set of handles for the characteristic and potential descriptors. The handles can be used later to identify the characteristic, for example, to identify which characteristic was written in a write event.

```
err_code = sd_ble_gatts_characteristic_add(p_lbs->service_handle, &char_md,
                            &attr_char_value,
                            &p_lbs->led_char_handles);
```

4 LED Button application example

The LED Button application example was created in order to give you an environment where you can learn how to use *Bluetooth* low energy on the nRF51822 chip. It is a simple BLE application that demonstrates bi-directional communication over BLE. When it is running, you will be able to toggle an LED output on the nRF51822 chip from the Central, and receive a notification when button input on nRF51822 is pressed.

The application is implemented in one service, with two characteristics—the LED characteristic, which can be used to control the LED remotely through the write without response operation, and the Button characteristic, which sends a notification when the button is pressed or released.

4.1 Code overview

An overview of how the application works is provided in the following sections to help you understand and use the code.

4.1.1 Code separation

The example code is divided into three files:

- `main.c`
- `ble_lbs.c`
- `ble_lbs.h`

The structure is the same as in other SDK examples, with `main.c` implementing application behavior and the separate service files implementing the service and its behavior. All I/O handling is left to the application.

An application running on nRF51822 can interface with the following parts:

- **Hardware registers**
  Typically done through writes to registers defined in nrf51.h with values contained in nrf51_bitfields.h. There are no examples of this in the LED Button application.

- **SDK modules**
  SDK modules can be anything, from basic header file wrappers around the hardware registers, like nrf_gpio.h, to more complex modules that give useful functionality to the application, like app_timer or ble_conn_params.

- **Softdevice functions**
  These are used to configure or trigger actions in the softdevice. All softdevice function calls are prefixed with sd_.

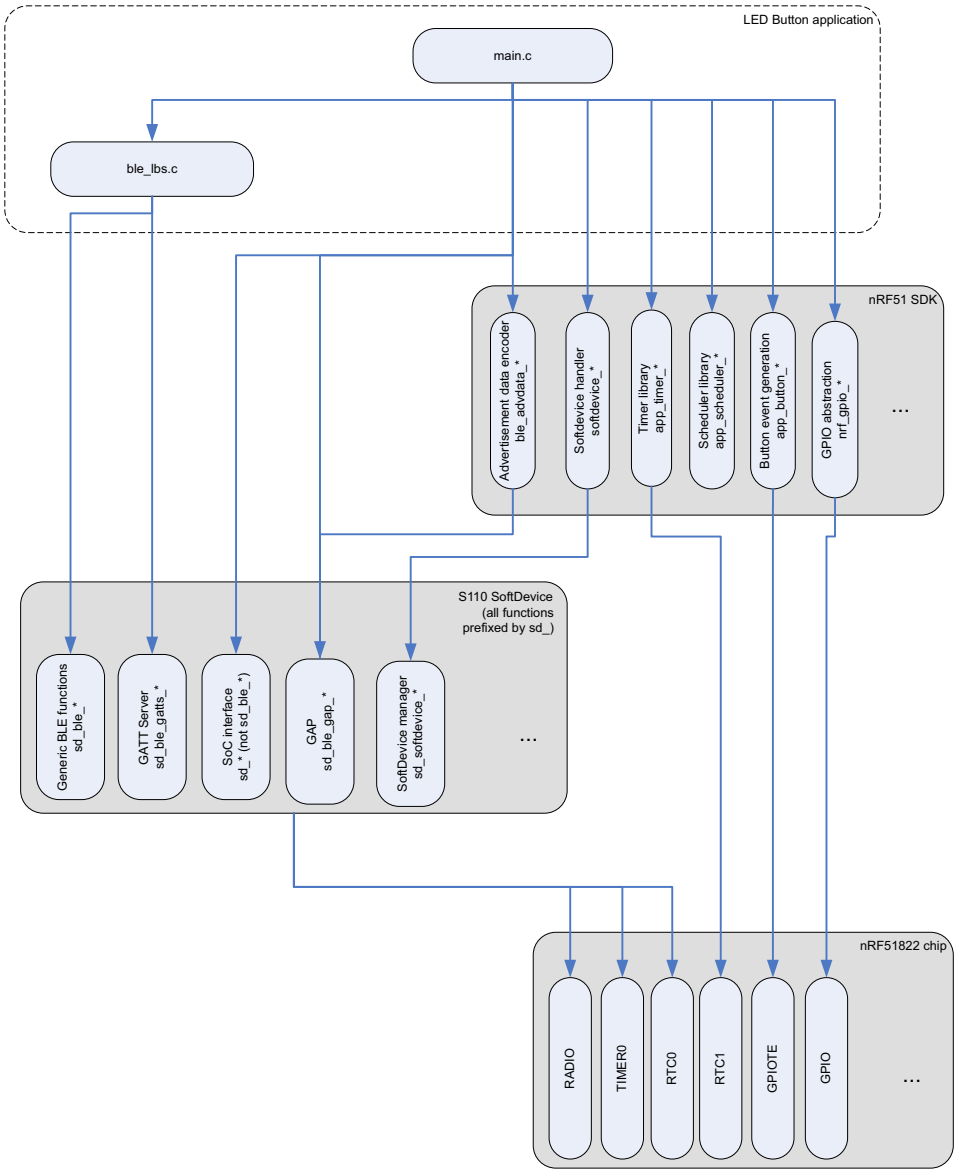


Figure 3 *LED Button application interfaces with the nRF51 SDK, S110 SoftDevice, and nRF51822 chip*

4.1.2 Code flow

The basic flow of a typical BLE application running on the nRF51822 chip is to initialize all needed parts, start advertising, and possibly enter a power-saving mode and wait for a BLE event. When an event is received, it is passed to all BLE services and modules. An event can be, but is not restricted to, one of the following:

- When a peer device connects to nRF51822.
- When a peer device writes to a characteristic.
- An indication that advertising has timed out.

This flow makes the application very modular and a service can usually be added to an application by initializing it and ensuring its event handler is called when an event comes in.

4.1.3 Inspecting and navigating in the Keil project

We recommend that you compile the example code before you start looking through it. After it is compiled you can right-click on any function, variable, type, or define and go to its point of definition by selecting **Go To Definition Of** or **Go To Reference To**.

Go To Definition Of brings you to the actual implementation of the method (that is, the source code file), while **Go To Reference To** brings you to its header file declaration. This means you can never jump to the definition of functions in the SoftDevice API, since these are not available as source code. You can however, jump to their reference, which will also show you the documentation available for the method in question. This is a powerful tool you can use to become familiar with the API and the example projects.

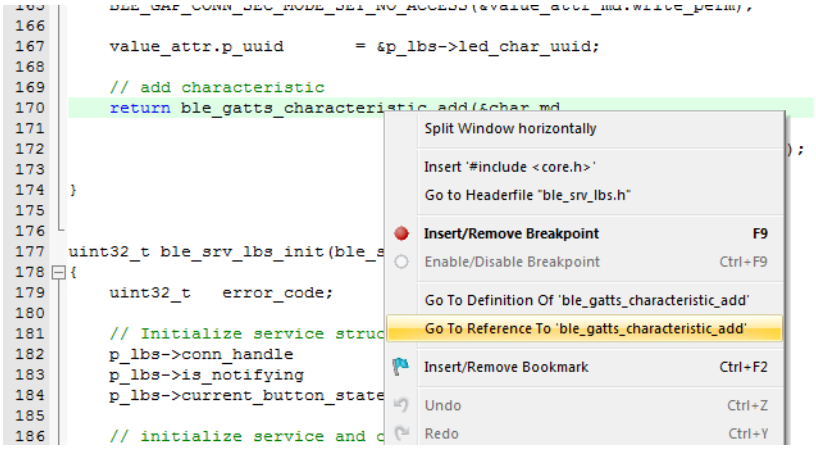


Figure 4 *Finding definitions and references for methods and variables in Keil.*

4.2 Code delivery

This application note explains all the steps needed to build the LED Button application. Additionally, a complete example is provided and available on GitHub as a Git repository. This enables you to inspect the history of the code and see how it was developed. Each section has its own tag, showing how the code should look at the end of the section.

The project can be found on GitHub at: https://github.com/NordicSemiconductor/nrf51-ble-app-lbs

4.3 Set up

The nRF51822 Evaluation Kit is needed for this application example. However, it is possible to modify the project to also work with the Development Kit.

4.3.1 Setting up the Evaluation board

Since the nRF51822 Evaluation Kit has an onboard SEGGER chip, you can connect the Evaluation board through a USB cable and immediately start working on it.

4.3.2 Setting up the application

A lot of boilerplate code is needed to get started creating an application and a service, so the first step is to copy code from the SDK:

1. Go to **Board\nrf6310\s110\ble_app_template** folder.
2. Copy this folder to **Board\pca10001\s110** and rename it to **ble_app_lbs**.
3. Inside the **arm** subfolder of **ble_app_lbs** change the name of the project files from ble_app_template to **ble_app_lbs**.

4.3.3 Setting up the service

The SDK doesn't have a template service. But, it has an implementation of the battery service, that is the simplest of the pre-implemented services, and is a good starting point for custom services. To get started follow these steps:

1. Copy `ble_bas.c` from **Source/ble/ble_services** to **Board/pca10001/ble/ble_app_lbs/**.
2. Copy `ble_bas.h` from **Include/ble/ble_services** to **Board/pca10001/ble/ble_app_lbs/**.
3. Rename `ble_bas.c` to `ble_lbs.c` and `ble_bas.h` to `ble_lbs.h`.
4. Double-click the Services folder in the Project pane to the left and select the newly created `ble_lbs.c`. This adds the `ble_lbs.c` file to your Keil project.

Since this is an application specific service, it is better that it is placed in the application folder instead of the SDK's service folder.

4.4 Implementing the service

The service is implemented generically so that it is easier to reuse for other applications. The goal is to enable the application to use the service by initializing it, handling events, and providing the I/O implementation. This is similar to the way predefined services are implemented.

4.4.1 Designing the API

The ble_lbs.h header file implements various structures, an event handler that the application needs to implement, and three API methods:

```
uint32_t ble_bas_init(ble_bas_t * p_bas, const ble_bas_init_t * p_bas_init);
void ble_bas_on_ble_evt(ble_bas_t * p_bas, ble_evt_t * p_ble_evt);
uint32_t ble_bas_battery_level_update(ble_bas_t * p_bas, uint8_t battery_level);
```

Note: Comments have been removed from the code snippets in this document.

In the code above, `ble_bas_t` is used for referencing this instance of the service while `ble_bas_init_t` includes initialization parameters that are not useful later. All API methods take a pointer to the service instance as their first parameter.

To design a similar API for the LED Button service, do the following:

1. Perform a Find and Replace All to replace all occurrences of `ble_bas` with `ble_lbs`, all occurrences of `BLE_BAS` with `BLE_LBS`, and all occurrences of `p_bas` with `p_lbs`, both in the header file and the source file.
2. Remove the `ble_lbs_battery_level_update()` function from both the header file and source file.
3. Comment out all other methods in the `ble_lbs.c` file by using an #if 0 before the first method and an #endif immediately above the init function. Do not completely remove them, because some of them will be built on later.
4. Remove the `battery_level_update` function, from both header file and source code.
5. Remove the call to `battery_level_char_add()` from the end of the init function, and instead just return NRF_SUCCESS.
6. Consider what the service needs from the application. The LED Button service needs to know when the button state has changed, which is sent to the central device. So, you need to add a method that the application can call when the button state changes:

```
uint32_t ble_lbs_init(ble_lbs_t * p_lbs, const ble_lbs_init_t * p_lbs_init);
void ble_lbs_on_ble_evt(ble_lbs_t * p_lbs, ble_evt_t * p_ble_evt);
uint32_t ble_lbs_on_button_change(ble_lbs_t * p_lbs, uint8_t button_state);
```

There are also two data structures there that need to be implemented here: `ble_lbs_t` and `ble_lbs_init_t`.

4.4.2 Implementing data structures

In the API from ***Section 4.4.1 "Designing the API"*** on page 18, some data structures that have not been implemented yet are used: `ble_lbs_t` and `ble_lbs_init_t`. We can base these on similar structures from the battery service, which look like the following:

```
typedef struct
{
    ble_bas_evt_handler_t          evt_handler;
    bool                           support_notification;
    ble_report_ref_t *             p_report_ref;
    uint8_t                        initial_batt_level;
    ble_cccd_security_mode_t       battery_level_char_attr_md;
    ble_gap_conn_sec_mode_t        battery_level_report_read_perm;
} ble_bas_init_t;

typedef struct ble_bas_s
{
    ble_bas_evt_handler_t          evt_handler;
    uint16_t                       service_handle;
    ble_gatts_char_handles_t       battery_level_handles;
    uint16_t                       report_ref_handle;
    uint8_t                        battery_level_last;
    uint16_t                       conn_handle;
    bool                           is_notification_supported;
} ble_bas_t;
```

The initialization structure from the code above contains an event handler, some optional parameters, initial value, and security modes for the information contained in the service. The service structure on the other hand contains the state of the service, such as handles, current battery level, whether notifications are enabled, and similar.

The battery service uses a general event handler to let the application know when to start and stop periodic battery level readings. The LED Button service doesn't rely on anything starting or stopping, so a single method being called when the LED characteristic is written to is enough.

This handler is the only thing that is applicable for initialization, and it becomes the lone member of the initialization structure:

```
typedef struct
{
    ble_lbs_led_write_handler_t   led_write_handler;
} ble_lbs_init_t;
```

The signature used for this function is defined like this (and must be added to the header file above the declaration of the ble_lbs_init_t, replacing the existing event handler definition):

```
typedef void (*ble_lbs_led_write_handler_t) (ble_lbs_t * p_lbs, uint8_t new_state);
```

However, the following parameters are needed to keep track of the state:

- The handle for the service.
- Characteristics handles.
- Connection handles.
- The UUID type.
- The handler for LED writes.

These parameters give the following service structure:

```
typedef struct ble_lbs_s
{
      uint16_t                     service_handle;
      ble_gatts_char_handles_t     led_char_handles;
      ble_gatts_char_handles_t     button_char_handles;
      uint8_t                      uuid_type;
      uint16_t                     conn_handle;
      ble_lbs_led_write_handler_t  led_write_handler;
} ble_lbs_t;
```

The battery service event declarations on top of `ble_lbs.h` can be removed.

4.4.3 Service Initialization

Start by looking at the init function, which is now called `ble_lbs_init`. The parameters you don't have need to be removed:

1. Remove `evt_handler`, `is_notification_supported`, and `battery_level_last`.
2. Rename the evt_handler field to `led_write_handler`, both in the init structure and the service structure:

```
p_lbs->led_write_handler          = p_lbs_init->led_write_handler;
```

The UUID handling needs to be reworked because the service being implemented uses a custom UUID instead of a *Bluetooth* SIG defined UUID.

First, define a custom Base UUID. One way to do this is in nRFgo Studio:

1. Open nRFgo Studio.
2. In the nRF8001 Setup menu, select **Edit 128-bit UUIDs** and click **Add new**.

This creates a new, random UUID that you can use for custom services.

The newly created Base UUID must be included in the source code as an array of bytes, but is only needed in one place:

1. For readability, include it as a macro definition in the header file `ble_lbs.h`, along with the shortened 16-bit versions used for the service and characteristics:

```
#define LBS_UUID_BASE {0x23, 0xD1, 0xBC, 0xEA, 0x5F, 0x78, 0x23, 0x15, 0xDE, 0xEF,
0x12, 0x12, 0x00, 0x00, 0x00, 0x00}
#define LBS_UUID_SERVICE 0x1523
#define LBS_UUID_LED_CHAR 0x1525
#define LBS_UUID_BUTTON_CHAR 0x1524
```

In the service initialization:

2. Add the following base UUID to the stack's list, and then set up the service to use it. Here it is added first, in ble_lbs_init():

```
    ble_uuid128_t base_uuid = LBS_UUID_BASE;
    err_code = sd_ble_uuid_vs_add(&base_uuid, &p_lbs->uuid_type);
    if (err_code != NRF_SUCCESS)
    {
      return err_code;
    }
```

The above code snippet will add the custom Base UUID to the stack, and store the type returned by `sd_ble_uuid_vs_add()` in the service structure:

3. Use this type when setting up the UUID for the LED Button service, still in `ble_lbs:init()`:

```
      ble_uuid.type = p_lbs->uuid_type;
      ble_uuid.uuid = LBS_UUID_SERVICE;

      err_code = sd_ble_gatts_service_add(BLE_GATTS_SRVC_TYPE_PRIMARY, &ble_uuid,
&p_lbs->service_handle);
      if (err_code != NRF_SUCCESS)
      {
            return err_code;
      }
```

The above code only adds an empty service, so the characteristics need to be added. The following section explains how to add the characteristics.

4.4.3.1 Implementing the button characteristic

The service will have two characteristics, one that controls the LED state and one that reflects the button state. Two static methods need to be created to add the characteristics to `ble_lbs.c` starting with the button state.

The button characteristic notifies on button state change, but also allows the peer device to read the button state. This is very similar to the behavior of the battery level characteristic in the Battery Service, so you can base your implementation on this:

1. Find the method called `battery_level_char_add` and rename it to `button_char_add`. If Find and Replace worked properly earlier, the parameter names should be correct as is.

The button_char_add method expects to find a parameter stating whether or not notifications are supported, but in this case only notification support is wanted.

2. Remove the if-sentence checking the `is_notification_supported` parameter, but leave the code inside.
3. Make sure that the char_md fields are set to support notifications.
4. Remove the entire code path for report reference, the `p_report_ref` parameter (everything from the if-sentence that checks if `p_report_ref` is set).

There is a flag in the initialization of the battery service that sets the security mode for the CCCD, and is stored as a `ble_gap_conn_sec_mode_t` structure. Such structures are most easily set by using one of the macros that start with `BLE_GAP_CONN_SEC_MODE` from `ble_gap.h`. There are different macros to use depending on the security level you want to require for this attribute.

1. Use BLE_GAP_CONN_SEC_MODE_SET_OPEN to make the CCCD readable and writable over any link, encrypted or not.
2. For the button state characteristic we want reading to be possible over any link, but not writing. For this, BLE_GAP_CONN_SEC_MODE_SET_NO_ACCESS can be used. Replace

BLE_GAP_CONN_SEC_MODE_SET_OPEN with BLE_GAP_CONN_SEC_MODE_SET_NO_ACCESS for the last write perm.

```
BLE_GAP_CONN_SEC_MODE_SET_OPEN(&cccd_md.read_perm);
BLE_GAP_CONN_SEC_MODE_SET_OPEN(&cccd_md.write_perm);
...
memset(&attr_md, 0, sizeof(attr_md));
BLE_GAP_CONN_SEC_MODE_SET_OPEN(&attr_md.read_perm);
BLE_GAP_CONN_SEC_MODE_SET_NO_ACCESS(&attr_md.write_perm);
```

3. Make sure you don't remove the setting of the vloc field, which decides whether the variable is placed in stack memory or in the application memory.
4. Set the type and value for the UUID.

```
ble_uuid.type = p_lbs->uuid_type;
ble_uuid.uuid = LBS_UUID_BUTTON_CHAR;
```

5. The initial value is not important, so you can pass a NULL for the p_initial_value.
6. Make sure to store the handles to the characteristic in the correct place, so modify the final call of the method to look like this:

```
return ble_gatts_characteristic_add(p_lbs->service_handle, &char_md,
                                    &attr_char_value,
                                    &p_lbs->button_char_handles);
```

By moving the #endif above this method you can compile the application and remove any unused variables shown as compilation warnings (initial_battery_level, encoded_report_ref, init_len, err_code).

4.4.3.2 Implementing the LED characteristic

The LED state characteristic needs to be writable and readable, without any notification:

1. Copy the method to add the button characteristic, renaming it to led_char_add.
2. Remove the references to the cccd_md.
3. Add the write property instead of the notify property (to enable writing to this characteristic).

```
char_md.char_props.write = 1;
```

4. Change the UUID to LBS_UUID_LED_CHAR.

```
ble_uuid.type = p_lbs->uuid_type;
ble_uuid.uuid = LBS_UUID_LED_CHAR;
```

The handle should be stored in the variable `led_char_handles` instead of in the `button_char_handles`, where the final call in the method looks like this:

```
return ble_gatts_characteristic_add(p_lbs->service_handle, &char_md,
                                    &attr_char_value,
                                    &p_lbs->led_char_handles);
```

After compilation, you can see which parameters are now unused, and remove them.

4.4.3.3 Adding the characteristics

After creating the methods to add to the characteristics, you need to call them at the end of the services initialization method. Here is an example of how to do this:

```
// Add characteristics
err_code = button_char_add(p_lbs, p_lbs_init);
if (err_code != NRF_SUCCESS)
{
       return err_code;
}
err_code = led_char_add(p_lbs, p_lbs_init);
if (err_code != NRF_SUCCESS)
{
       return err_code;
}

return NRF_SUCCESS;
```

Because any errors would have forced the function to return early, you can assume success if you reach the end of this function.

If you want to test this now, you can jump to ***Section 4.5.1 "Modifying the template for the evaluation kit"*** on page 26 and ***Section 4.5.3 "Including the service"*** on page 28, and finish them before testing as explained in ***Chapter 5 "Testing the Application"*** on page 32. After testing you should be able to connect to the device and discover all the services, but any further action will not work. Handling stack events and button presses still needs to be implemented in the service.

4.4.4 Handling stack events

A stack event occurs whenever the stack needs to notify the application of something that is relevant to it, such as when writing to a characteristic or descriptor. For this application, the write to the LED characteristic is what you will need. For notifications to work properly, you also need to store the connection handle, which you can do on the connect and disconnect events.

As part of the API, you defined a method called ble_lbs_on_ble_evt to be used to process stack events. This is split into different methods to handle the different events based on a simple switch-case.

4.4.4.1 Storing the connection handle

The battery service already stored the connection handle, and the Find and Replace from earlier should have taken care of the rest without any further changes.

4.4.4.2 Removing handling of CCCD writes

The existing event handler listens for write operations to the CCCD and passes them on to the application's battery service event handler. However, in this application, this is not required.

The documentation for the method to send notifications, `sd_ble_gatts_hvx()` states that it won't allow notifications to be sent if the CCCD is not enabled, so you don't need to check this in the application in addition to the check done inside the SoftDevice.

You can remove the entire current content of the on_write method.

4.4.4.3 Handling LED characteristic writes

The function pointer you added to the data structures enables the application to be notified when the LED characteristic is written to. This should be handled in the on_write method.

The basic tasks when a write event is received is to verify that the write happened on the correct characteristic, verify that it was the correct size, and that a handler is set. If all of this is correct, the handler can be called with whatever value was written. Therefore, the content of on_write becomes like this:

```
ble_gatts_evt_write_t * p_evt_write = &p_ble_evt->evt.gatts_evt.params.write;

   if ((p_evt_write->handle == p_lbs->led_char_handles.value_handle) &&
       (p_evt_write->len == 1) &&
       (p_lbs->led_write_handler != NULL))
   {
       p_lbs->led_write_handler(p_lbs, p_evt_write->data[0]);
   }
```

The actual toggling of the LED is left to the application, making the service easy to reuse as-is, even if a different pinout is used for LEDs and buttons.

4.4.5 Handling button press

You have already added an API method that lets the service know when a button is pressed. This has not been implemented yet, so you need to add it by copying the definition from the header file. When handling a button press, you want to send a notification to the peer device with the new button state. The SoftDevice API method for doing this is called sd_ble_gatts_hvx, and takes a connection handle and a ble_gatts_hvx_params_t structure as parameters. It then manages the process when a value is to be notified.

In the ble_gatts_hvx_params_t structure, you set whether you want a notification or indication, which attribute handle to be notified, the new data, and the new data length. The method will look as follows:

```
uint32_t ble_lbs_on_button_change(ble_lbs_t * p_lbs, uint8_t button_state)
{
    ble_gatts_hvx_params_t params;
    uint16_t len = sizeof(button_state);

    memset(&params, 0, sizeof(params));
    params.type = BLE_GATT_HVX_NOTIFICATION;
    params.handle = p_lbs->button_char_handles.value_handle;
    params.p_data = &button_state;
    params.p_len = &len;

    return sd_ble_gatts_hvx(p_lbs->conn_handle, &params);
}
```

It would also be possible to first set the value for the characteristic by using `sd_ble_gatts_value_set()`, and then notify by calling `sd_ble_gatts_hvx()` without setting a value or length. However, using `sd_ble_gatts_hvx()` does everything that's necessary, and is a cleaner solution. Use the method `sd_ble_gatts_value_set()` to update a readable (but not notifiable) value, since this function will not send any packets over the air.

This concludes the service, and the rest of the example should be implemented in the application.

4.5 Application implementation

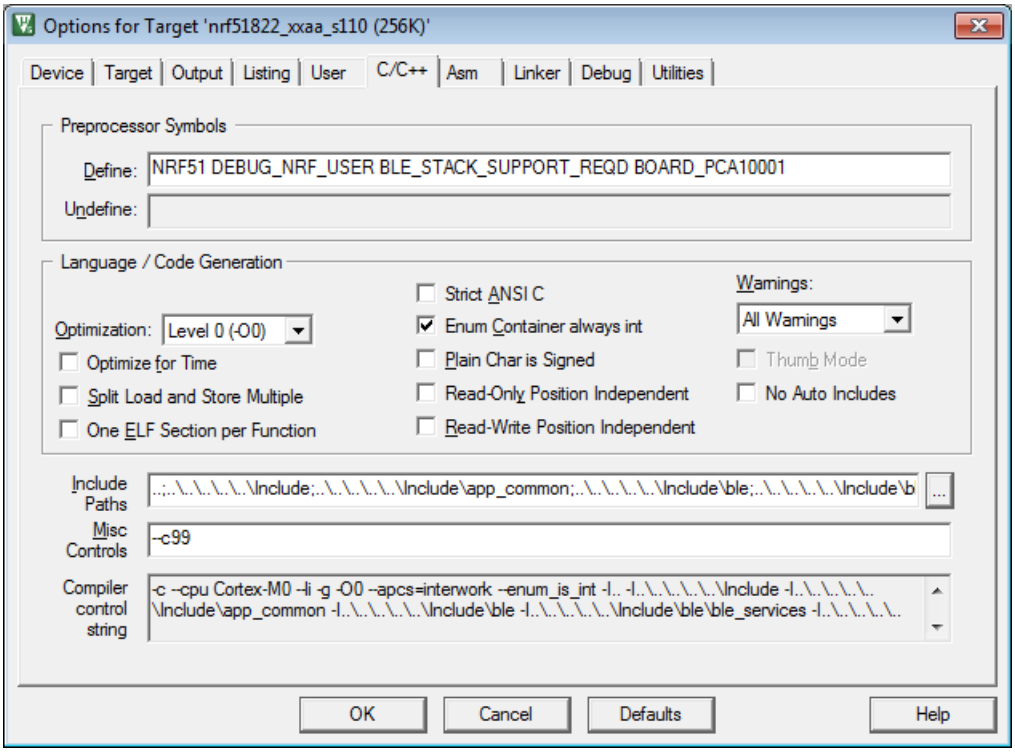
4.5.1 Modifying the template for the evaluation kit

Some modifications are needed to use the template application with the evaluation kit instead of the Development Kit.

First, you need to change the board definition macros:

1. Open the project folder and go to **Target options** and the C/C++ tab.
2. Replace BOARD_NRF6310 with BOARD_PCA10001, as shown in the figure below.

Note: If you want to use the development kit instead of the evaluation kit, skip steps 4 and 5 above, and copy ble_app template into the same folder so that you get **Board\nrf6310\s110\ble_app_lbs** project.



You also need to remove a few pin definitions from main.c. On the evaluation kit you cannot have a separate LED to show ASSERTs.

1. Remove all references to ASSERT_LED_PIN_NO from main.c (macro definitions, use in app_error_handler() and setting as output in leds_init()).
2. Instead of the assert LED, you need an LED to use for the LED Button service, so add a define for the LED pin at the top of main.c:

```
#define LEDBUTTON_LED_PIN_NO              LED_0
```

3. In leds_init(), configure it as an LED:

```
nrf_gpio_cfg_output(LEDBUTTON_LED_PIN_NO);
```

4. In SDK 4.1.0 and later, the default application error handler does a reset when an error happens, but for development, it is much more useful to use the debug module provided. You should make sure to uncomment the debug assert handler, and comment out the reset:

```
void app_error_handler(uint32_t error_code, uint32_t line_num, const uint8_t *
p_file_name)
{
    // [Comment removed from snippet for brevity]
    ble_debug_assert_handler(error_code, line_num, p_file_name);

    // On assert, the system can only recover with a reset.
    //NVIC_SystemReset();
}
```

This enables you to run the application with the debugger, and if an error occurs you can halt the chip and find out on what line file the error occurred. This is one of the reasons you should always check error codes returned by the SoftDevice with APP_ERROR_CHECK() macro. See ***Figure 5*** for an example.

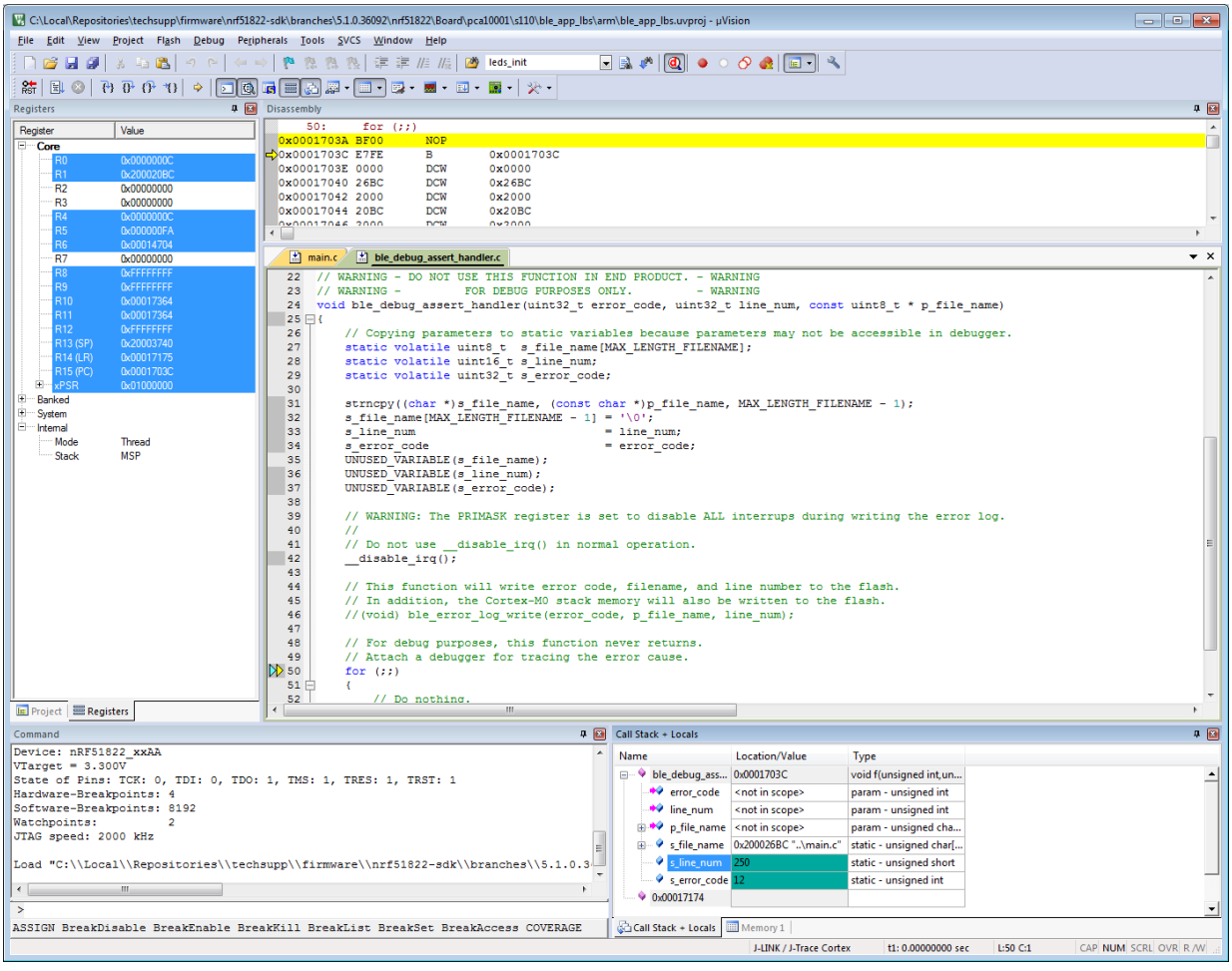


Figure 5 *Application is halted with the debugger in the assert handler, showing an error with error code 12*

In actual production, a reset is usually the only recovery option, but for development, the logger is more useful.

It is also recommended to change the *Bluetooth* name used by the project, by changing the DEVICE_NAME macro definition in main.c to something like "LedButtonDemo".

4.5.2 Using the scheduler

The SDK contains a scheduler module which provides a mechanism for moving the handling of events and interrupts from the interrupt handlers to main context. This ensures that all interrupt handlers execute quickly.

In the template application we started with, the scheduler is enabled by default. If you do not want to use it, you can remove its initialization and the main loop call to it (scheduler_init(), app_sched_execute()), and set the last argument to most of the SDK modules' init functions to false (softdevice_handler, app_timer, app_button)."

For more details on the scheduler, please see the nRF51 SDK documentation.

4.5.3 Including the service

To use the service you've created, you need to add some code to the template application.

In the main.c file, there is a method called services_init where the initialization of the LED Button service must occur:

1. Add the inclusion of the ble_lbs.h file at top of main.c as the last include:

```
#include "ble_lbs.h"
```

2. If not already done, add the source file of the service to the project. Right-click the **Services** folder in the Project explorer to the left, click **Add file** and select the **ble_lbs.c** file.
3. Add the data structure for the service as a static global variable in main.c:

```
static ble_lbs_t                        m_lbs;
```

Note: Even though this is stored as a static variable in the main.c file, m_lbs, it will often appear as a pointer to this variable, and therefore be called p_lbs.

4. Now you are ready to initialize your service:

```
static void services_init(void)
{
    uint32_t err_code;
    ble_lbs_init_t init;

    init.led_write_handler = led_write_handler;

    err_code = ble_lbs_init(&m_lbs, &init);
    APP_ERROR_CHECK(err_code);
}
```

In ***Section 4.4.4.3 "Handling LED characteristic writes"*** on page 25, we made the service call the led_write_handler in the service structure when the LED characteristic is written. The function we pass through the init structure above is the one that will be called, but this function has not yet been defined in the application.

5. Add a function called led_write_handler with the signature shown in the code below immediately above services_init.
6. Set the LED output to the state given as a function parameter, like this:

```
Static void led_write_handler(ble_lbs_t * p_lbs, uint8_t led_state)
{
    if (led_state)
    {
        nrf_gpio_pin_set(LEDBUTTON_LED_PIN_NO);
    }
    else
    {
        nrf_gpio_pin_clear(LEDBUTTON_LED_PIN_NO);
    }
}
```

7. Finally, add the service's event handler to the application's event dispatcher:

```
static void ble_evt_dispatch(ble_evt_t * p_ble_evt)
{
    on_ble_evt(p_ble_evt);
    ble_conn_params_on_ble_evt(p_ble_evt);
    ble_lbs_on_ble_evt(&m_lbs, p_ble_evt);
}
```

4.5.4 Test it

Now is a good time to run a quick test, as shown in ***Chapter 5 "Testing the Application"*** on page 32. By using the Master Control Panel, you should be able to turn on the LED by writing a '1' to the LED characteristic.

4.5.5 Button handling

To finish the application, you need to define how to handle button presses. Use the app_button module that is part of the SDK for this. This module will give a callback on both the press and the release of the button.

In buttons_init, set up the button you want to use, in this case button 1 on the evaluation kit.

1. Add a new macro definition, just to make the code more readable:

```
#define LEDBUTTON_BUTTON_PIN_NO          BUTTON_1
```

2. To set up the pin in buttons_init, add the button configuration array like this:

```
    static app_button_cfg_t buttons[] =
    {
        {WAKEUP_BUTTON_PIN,       false, NRF_GPIO_PIN_PULLUP, NULL},
        {LEDBUTTON_BUTTON_PIN_NO, false, NRF_GPIO_PIN_PULLUP, button_event_handler},
    };

    APP_BUTTON_INIT(buttons, sizeof(buttons) / sizeof(buttons[0]),
BUTTON_DETECTION_DELAY, true);
```

The buttons on the evaluation kit are active low, which is the reason for the false, but they don't have an external pull-up. You will have to turn on the internal pull-up by using NRF_GPIO_PIN_PULLUP. The wakeup-button is of no significance, so the handler is set to NULL. After this, you will initialize the module.

3. Uncomment the function button_event_handler. The app_button module will pass a parameter showing the current state of the button, so we can just pass this directly on to the LED Button service API method:

```
static void button_event_handler(uint8_t pin_no, uint8_t button action)
{
    uint32_t err_code;

    switch(pin_no)
    {
        case LEDBUTTON_BUTTON_PIN_NO:
            err_code = ble_lbs_on_button_change(&m_lbs, button_action);
            if (err_code != NRF_SUCCESS &&
                err_code != BLE_ERROR_INVALID_CONN_HANDLE &&
                err_code != NRF_ERROR_INVALID_STATE)
            {
                APP_ERROR_CHECK(err_code);
            }
            break;

       default:
                APP_ERROR_HANDLER(pin_no);
           break;
    }
}
```

In the above code we ignore any errors that may come if the CCCD has not yet been written to by the client, or if we are not currently in a connection.

In addition to defining the method, you need to make sure the button module is enabled. By default, the template application suggests the connect and disconnect events for this, which is OK for the use case here as well. Uncomment the calls to app_button_enable() and app_button_disable():

```
    switch (p_ble_evt->header.evt_id)
    {
        case BLE_GAP_EVT_CONNECTED:
            …
            /* … */
            err_code = app_button_enable();
            break;

        case BLE_GAP_EVT_DISCONNECTED:
            …
            /* … */
            err_code = app_button_disable();
            if (err_code == NRF_SUCCESS)
            {
                advertising_start();
            }
            break;
```

You are now ready to test the required functionality application, as described in ***Chapter 5 "Testing the Application"*** on page 32. However, to be able to easily distinguish devices when Centrals are scanning, it's useful to add the service UUID to the advertising packet.

4.5.6 Adding the service UUID to the advertising packet

Having the service UUID in the advertising packet enables a Central to use this information to decide whether it will connect. As mentioned in ***Section 2.1.2 "Advertising"*** on page 6, an advertising packet can contain 31 bytes, but if additional data is needed, a scan response can be sent.

You will need to add the custom UUID, which is 16 bytes, to the scan response packet because there is no room for it in the advertising packet.

Advertising data is set up in the advertising_init() function in main.c, which sets up data structures and calls ble_advdata_set(). This method then takes two parameters of the same type, one for the advertising packet and one for the scan response. You must add a structure to pass as the scan response parameter.

The UUID can then be set to LBS_UUID_SERVICE, and type as the uuid_type field in the ble_lbs_t structure. The advertising_init() should like this:

```
static void advertising_init(void)
{
    uint32_t      err_code;
    ble_advdata_t advdata;
    ble_advdata_t scanrsp;
    uint8_t       flags = BLE_GAP_ADV_FLAGS_LE_ONLY_LIMITED_DISC_MODE;

    // YOUR_JOB: Use UUIDs for service(s) used in your application.
    ble_uuid_t adv_uuids[] = {{LBS_UUID_SERVICE, m_lbs.uuid_type}};

    // Build and set advertising data
    memset(&advdata, 0, sizeof(advdata));

    advdata.name_type               = BLE_ADVDATA_FULL_NAME;
    advdata.include_appearance      = true;
    advdata.flags.size              = sizeof(flags);
    advdata.flags.p_data            = &flags;

    memset(&scanrsp, 0, sizeof(scanrsp));
    scanrsp.uuids_complete.uuid_cnt = sizeof(adv_uuids) / sizeof(adv_uuids[0]);
    scanrsp.uuids_complete.p_uuids  = adv_uuids;

    err_code = ble_advdata_set(&advdata, &scanrsp);
    APP_ERROR_CHECK(err_code);
}
```

Since the uuid_type of the m_lbs structure is used here, make sure that services_init(), in which this is set, is called before advertising_init() in main:

```
int main(void)
{
    …
    services_init();
    advertising_init();
    …
```

Now the application is fully complete.

5 Testing the Application

A USB dongle is supplied with the evaluation kit that, when combined with the Master Control Panel software, can be used to test BLE applications. The initial set up for this is described in the Quick Start section of the *nRF51822 Evaluation Kit User Guide*. Once you have opened the Master Control Panel, you can test the LED Button application by following these steps:

1. Open the Master Control Panel.

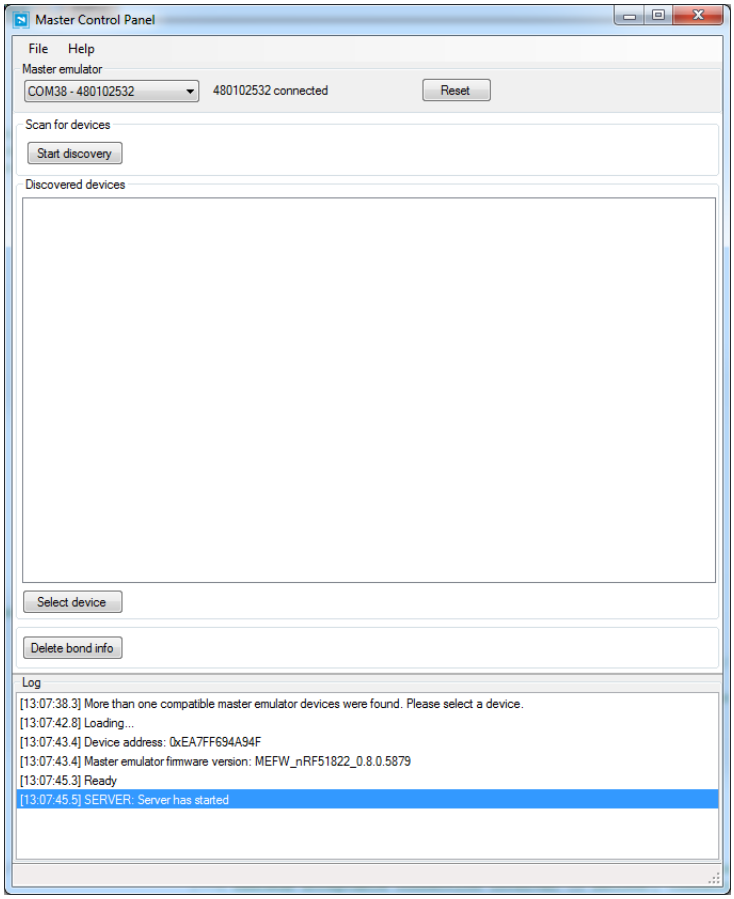


Figure 6 *Master Control Panel*

2. When the Master Control Panel has started, click **Start Discovery**. The LED Button device should appear shortly in the Discovered devices window. If it doesn't, it probably means it has hit the advertising timeout; press button 0 on the Evaluation Kit board or reset the device to start advertising again.
3. When the device appears, select it and click **Select Device**.

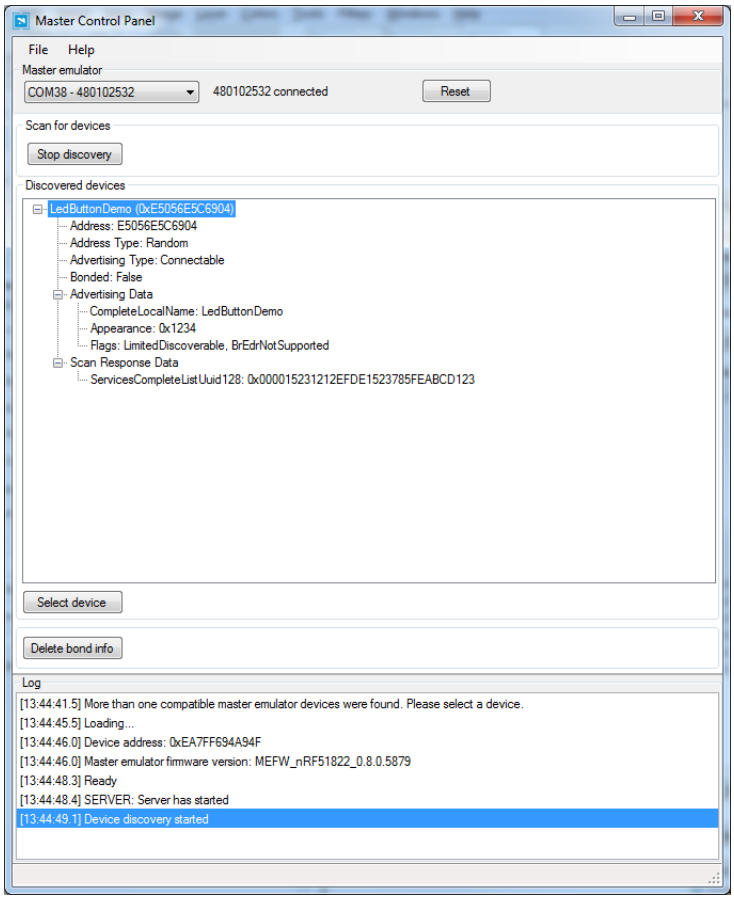


Figure 7 *Master Control Panel after the device has been discovered, showing the scan response data.*

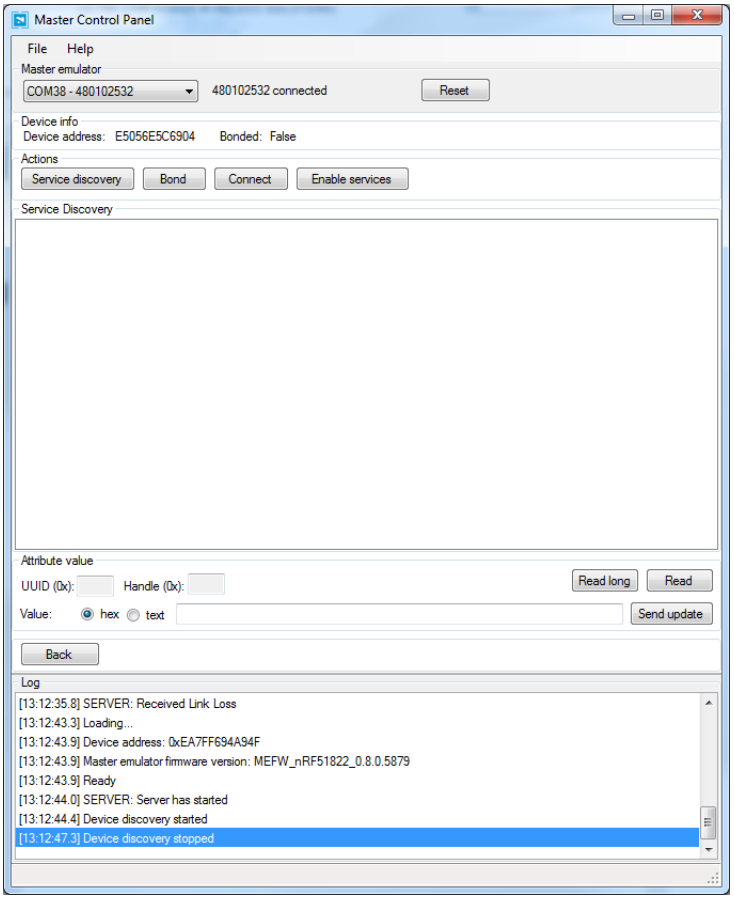


Figure 8 *Master Control Panel when the LED Button device has been selected.*

4. Click **Service discovery**. This will first connect to the device, and then do a service discovery.

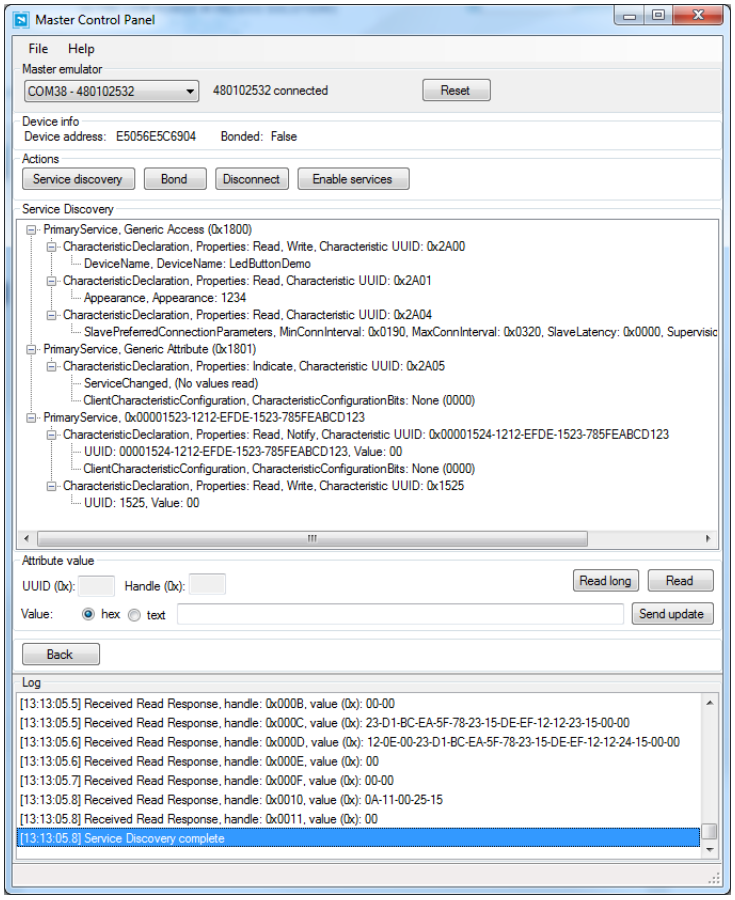


Figure 9 *Master Control Panel after service discovery*

You will see that the device has three services, even though only one was added. You can find the LED Button service at the bottom. The other two are the GAP Service (0x1800), which contains GAP data, including some of the parameters set earlier; and GATT Service (0x1801), which can contain a characteristic to be used if any services are changed after initialization. All BLE devices must have these services, and the SoftDevice adds them automatically.

We can now turn on notifications and see if button presses show up.

5. Click **Enable services** to turn on notifications, and press Button 1 on the Evaluation Kit.

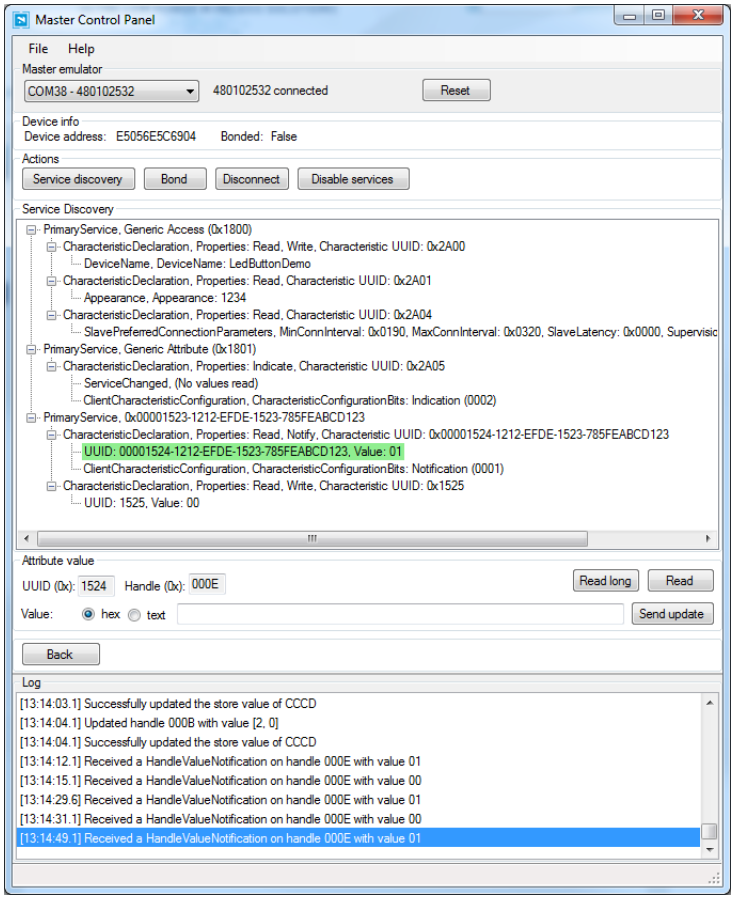


Figure 10 *Master Control Panel. Notifications have been turned on and a button has been pressed.*

You will see that the notification bit of the CCCD has been set to 1, and the value of the characteristic is updated when the button is pressed.

6. To test that the LED lights up, click the value of the LED characteristic. Under Attribute value, set the Value to **hex**, type 01 into the field, and click **Send update**. This will send a write operation over the air to the device, which will light the LED.

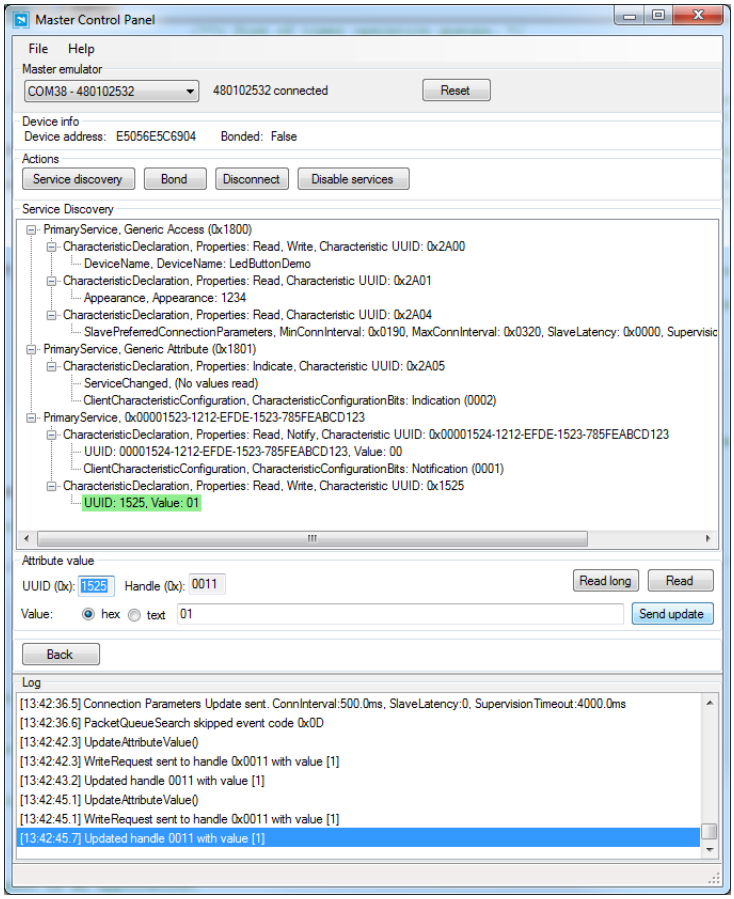


Figure 11 *Master Control Panel. The value of the LED characteristic has been updated to 01.*

Liability disclaimer

Nordic Semiconductor ASA reserves the right to make changes without further notice to the product to improve reliability, function or design. Nordic Semiconductor ASA does not assume any liability arising out of the application or use of any product or circuits described herein.

Life support applications

Nordic Semiconductor's products are not designed for use in life support appliances, devices, or systems where malfunction of these products can reasonably be expected to result in personal injury. Nordic Semiconductor ASA customers using or selling these products for use in such applications do so at their own risk and agree to fully indemnify Nordic Semiconductor ASA for any damages resulting from such improper use or sale.

Contact details

For your nearest distributor, please visit http://www.nordicsemi.com.
Information regarding product updates, downloads, and technical support can be accessed through your My Page account on our homepage.

Main office: Otto Nielsens veg 12
7052 Trondheim
Norway
Phone: +47 72 89 89 00
Fax: +47 72 89 89 89

Mailing address: Nordic Semiconductor
P.O. Box 2336
7004 Trondheim
Norway

Revision History

Date	Version	Description
June 2014	1.1	**Added content:** • ***Section 4.2 "Code delivery"*** on page 17. **Updated content:** • ***Section 2.2.4 "UUIDs"*** on page 9.
October 2013	1.0	First release.