

Before we begin

Scope

Topics that will be covered include:

- 1. Before we begin
 - Necessary equipment and software
 - Necessary prior knowledge
 2. Some basic theory
 - The Generic Attribute Profile (GATT)
 - Services
 - Characteristics
 - Universally Unique ID (UUID)
 3. The example
 - First thing first
 - Our first service
 - Advertising
 4. Summary

Change Log

2016.03.14: Updated tutorial to use SDK V11.0.0 and the possibility to use both nRF51 and nRF52 kits.

Equipment and software

To complete this tutorial you will need the following:

- - [nRF51 DK or nRF52 DK Development Kit](<https://www.nordicsemi.com/eng/Products/nRF51-DK>)
 - [nRF51 Dongle](<https://www.nordicsemi.com/eng/Products/nRF51-Dongle>)
 - Keil V5.xx.
 - [Master Control Panel (MCP)](<https://www.nordicsemi.com/eng/Products/Bluetooth-Smart-Bluetooth-low-energy/nRF51822#Downloads>). (You can also use the Master Control Panel found in the nRF Toolbox app for Android or the LightBlue app for iPhone)
 - [nRFgo Studio](<https://www.nordicsemi.com/eng/Products/Bluetooth-Smart-Bluetooth-low-energy/nRF51822#Downloads>)
 - [SDK V11.0.0](http://developer.nordicsemi.com/nRF51_SDK/) ****NOTE!** This tutorial is written for SDK V11.0.0. Example files for SDK V9.0.0 can also be found on github, but the APIs are slightly different.**
 - For nRF51 DK use SoftDevice [S130 V2.0.0](<https://www.nordicsemi.com/eng/nordic/Products/nRF51822/S130-SD-v2/53724>) for nRF52 DK use [S132

V2.0.0](<https://www.nordicsemi.com/eng/nordic/Products/nRF52832/S132-SD-v2/51479>).

- Example files found on [github](<https://github.com/NordicSemiconductor/nrf51-ble-tutorial-service>).

Other kits, dongles and software versions might work as well, but this is what I have used. This tutorial will not cover how to install and setup the software. Please search the forum if you run into trouble before posting questions here.

Necessary prior knowledge

This tutorial is intended to be a natural continuation of the tutorial "[BLE Advertising, a beginner's tutorial](#)". It is recommended, but not necessary, to go through this tutorial first. It is expected that you have basic knowledge of how to use Keil, MCP and nRFgo Studio and how to download your application to your kit. Read the tutorial [Setting up an example project on the nRF51 DK](#) to learn how to use your equipment and compile your first BLE application.

If you run into troubles please browse devzone, look for documentation on our [Infocenter](#), and read the user guides for your kits. I urge you to post any questions you might have on the forum and not below the tutorial. This makes it easier for other users and Nordic employees to see and search for your question and you will actually most likely get a faster response(!).

Some basic theory

The Generic Attribute Profile (GATT)

[The Bluetooth Core Specification](<https://www.bluetooth.org/en-us/specification/adopted-specifications>) defines the GATT like this:

“The GATT Profile specifies the structure in which profile data is exchanged. This structure defines basic elements such as services and characteristics, used in a profile.”

In other words, it is a set of rules describing how to bundle, present and transfer data using BLE. Read the [Bluetooth Core Specification v4.2](#), Vol. 3, Part G for more information. It might be a little heavy reading, but it will certainly pay off in the end.

Services

The Bluetooth Core Specification defines a service like this:

“A service is a collection of data and associated behaviors to accomplish a particular function or feature. [...] A service definition may contain [...] mandatory characteristics and optional characteristics.”

In other words, a service is a collection of information, like e.g. values of sensors. [Bluetooth Special Interest Group](#) (Bluetooth SIG) has [predefined certain services](#). For example they have defined a service called Heart Rate service. The reason why they have done this is to make it easier for developers to make apps and firmware compatible with the standard Heart

Rate service. However, this does not mean that you can't make your own heart rate sensor based on your own ideas and service structures. Sometimes people mistakenly assumes that since Bluetooth SIG has predefined some services they can only make applications abiding by these definitions. This is not the case. It is no problem to make custom services for your custom applications.

Characteristics

The Bluetooth Core Specification defines a characteristic like this:

“A characteristic is a value used in a service along with properties and configuration information about how the value is accessed and information about how the value is displayed or represented.”

In other words, the characteristic is where the actual values and information is presented. Security parameters, units and other metadata concerning the information are also encapsulated in the characteristics.

An analogy might be a storage room filled with filing cabinets and each filing cabinet has a number of drawers. The GATT profile in this analogy is the storage room. The cabinets are the services, and the drawers are characteristics holding various information. Some of the drawers might also have locks on them restricting the access to its information.

Imagine a heart rate monitor watch for example. Watches like this typically use at least two services:

- 1. [A Heart rate service](https://developer.bluetooth.org/gatt/services/Pages/ServiceViewer.aspx?u=org.bluetooth.service.heart_rate.xml). It encapsulates three characteristics:
 1. A mandatory [Heart Rate Measurement characteristic](https://developer.bluetooth.org/gatt/characteristics/Pages/CharacteristicViewer.aspx?u=org.bluetooth.characteristic.heart_rate_measurement.xml) holding the heart rate value.
 2. An optional [Body Sensor Location characteristic](https://developer.bluetooth.org/gatt/characteristics/Pages/CharacteristicViewer.aspx?u=org.bluetooth.characteristic.body_sensor_location.xml).
 3. A conditional [Heart Rate Control Point characteristic](https://developer.bluetooth.org/gatt/characteristics/Pages/CharacteristicViewer.aspx?u=org.bluetooth.characteristic.heart_rate_control_point.xml).
 2. [A Battery service](https://developer.bluetooth.org/gatt/services/Pages/ServiceViewer.aspx?u=org.bluetooth.service.battery_service.xml):
 1. Mandatory [Battery level characteristic](https://developer.bluetooth.org/gatt/characteristics/Pages/CharacteristicViewer.aspx?u=org.bluetooth.characteristic.battery_level.xml).

Now why bother with this? Why not just send whatever data you need directly without the fuzz of bundling it in characteristics and services? The reasons are flexibility, efficiency, cross platform compatibilities and ease of implementation. When iPhones, Android tablets or Windows laptops discover a device advertising a heart rate service they can be 100% sure to find at least the heart rate measurement characteristic and the characteristic is guaranteed to be presented in a standardized way. If a device contains more than one service you are free to pick and choose the services and characteristics you like. By bundling information this way devices can quickly discover what information is available and communicate only what is strictly needed and thereby save precious time and energy. Remember that BLE is all about low energy.

To continue the analogy: the storage room is located in a small business office and has two filing cabinets. The first cabinet is used by the accountants. The drawers contain files with financial details of the business, sorted by date. The drawers are locked and only the accountants and the upper management have access to them. The second cabinet is used by Human Resources and contains records over the employees, sorted in alphabetical order. These drawers are also locked and only HR and upper management have access to them. Everyone in the business knows where the storage room is and what it is for, but only some people have access to it and use it. It ensures efficiency, security and order.

Universally Unique ID (UUID)

A UUID is an abbreviation you will see a lot in the BLE world. It is a unique number used to identify services, characteristics and descriptors, also known as attributes. These IDs are transmitted over the air so that e.g. a peripheral can inform a central what services it provides. To save transmitting air time and memory space in your nRF51 there are two kinds of UUIDs:

The first type is a short 16-bit UUID. The predefined [Heart rate service](#), e.g., has the UUID 0x180D and one of its enclosed characteristics, the [Heart Rate Measurement characteristic](#), has the UUID 0x2A37. The 16-bit UUID is energy and memory efficient, but since it only provides a relatively limited number of unique IDs there is a rule; you can only transmit the predefined Bluetooth SIG UUIDs directly over the air. Hence there is a need for a second type of UUID so you can transmit your own custom UUIDs as well.

The second type is a 128-bit UUID, sometimes referred to as a vendor specific UUID. This is the type of UUID you need to use when you are making your own custom services and characteristics. It looks something like this: 4A98xxxx-1CC4-E7C1-C757-F1267DD021E8 and is called the “base UUID”. The four x’s represent a field where you will insert your own 16-bit IDs for your custom services and characteristics and use them just like a predefined UUID. This way you can store the base UUID once in memory, forget about it, and work with 16-bit IDs as normal. You can generate base UUIDs using nRFgo Studio. It is very easy and you can look in the Help menu to learn how.

A little fun fact about UUIDs: There is no database ensuring that no one in the world is sharing the same UUID, but if you generate two random 128-bit UUIDs there is only a $\sim 3e-39$ chance that you will end up with two identical IDs (that is $\sim 1/340,000,000,000,000,000,000,000,000,000,000,000$).

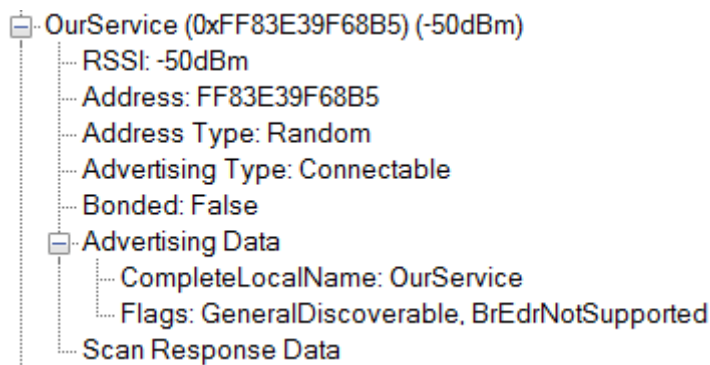
The example

First thing first

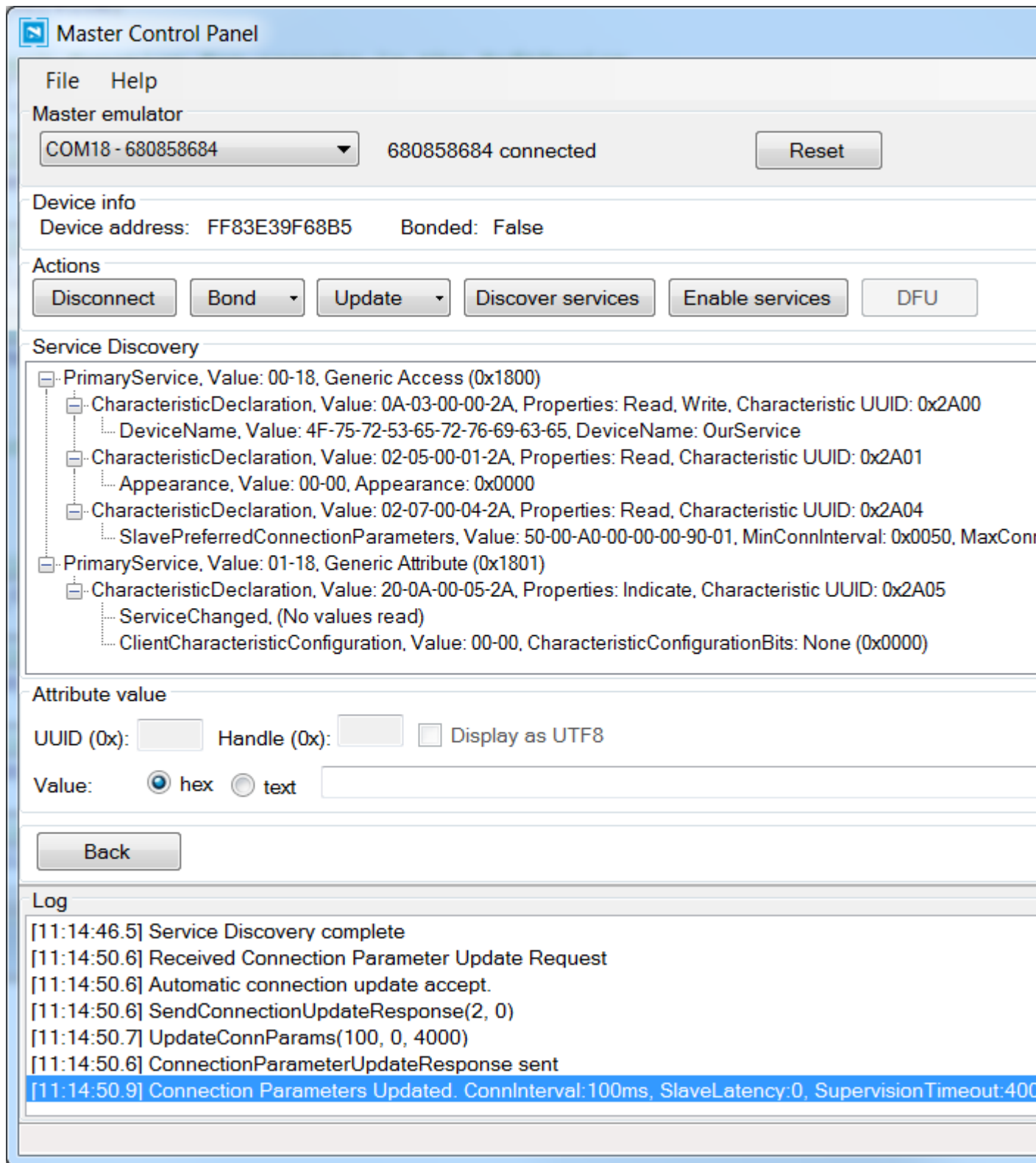
Download the example code from github. It is based on the template example found in the SDK, but stripped of all code that is not strictly necessary for our purpose. To compile it download the project files and copy the folder "nrf5x-ble-tutorial-service" to "your_SDK_folder\examples\ble_peripheral". If you need help with this please have a look at [this thread on devzone](https://devzone.nordicsemi.com/question/35068/compiling-github-projects/).

Open the project file "nrf51-ble-tutorial-service.uvprojx". As you can see I have implemented the SEGGER Real-Time Terminal (RTT) to our project so that we can easily see what is happening. Read this tutorial to learn how to use the RTT: [Debugging with Real Time Terminal](#).

The example should compile without any errors, but there might be some warnings about unused variables. So hit build and then download the code to your board. Your device should then show up in MCP like this:



If you click on "Select device" and then "Discover services" you should see this:



As you can see, even before we have done anything, there are already two mandatory services set up for us:

[The Generic Access service](#). Service UUID 0x1800. Three mandatory characteristics:

1. [Characteristic: Device name](#). UUID 0x2A00.

2. [Characteristic: Appearance](#). UUID 0x2A01.
3. [Characteristic: Peripheral Preferred Connection Parameters](#). UUID 0x2A04.

[The Generic Attribute service](#). UUID 0x1801. One optional characteristic:

1. [Characteristic: Service Changed](#). UUID 0x2A05.

The Generic Access Service contains general information about the device. You can recognize a characteristic holding the device name “OurService”. The second characteristic holds the appearance value and in our case we haven't set the value to anything so it just shows 0x0000. The third characteristic holds various parameters used to establish a connection. You can recognise values from the #defines in the example called: MIN_CONN_INTERVAL, MAX_CONN_INTERVAL, SLAVE_LATENCY, and CONN_SUP_TIMEOUT. [Here](#) is a short explanation regarding these parameters.

The second service is the Generic Attribute Service. Simply put, this service can be used to notify the central of changes made to the fundamental structure of services and characteristics on the peripheral. Short explanation [here](#).

Our first service

As you can see I have included two files in the example; `our_service.h` and `our_service.c`. Inside I have declared some empty functions and a little struct so that it will be easier for us to get started. I have also used nRFgo Studio to create and define a 128-bit base UUID, defined as `BLE_UUID_OUR_BASE_UUID`. Search for STEP 1-7 in the project files and you will find the places where we need to add code.

Step 1: Declare a service structure

First of all we need a place to store all data and information relevant to our service and to do this we will use the `ble_os_t` structure. As you can see in `our_service.h` the structure only holds one entry by now. The `service_handle` is a number identifying this particular service and is assigned by the SoftDevice. Declare a variable called `m_our_service` of type `ble_os_t` in `main.c` so that we can pass it to various functions and have complete control of our service.

Step 2: Initialize the service

In `main.c` there is already a function called `services_init()` waiting for you. Inside this function we will call `our_service_init()`. It takes a pointer to a `ble_os_t` struct as a parameter so make sure that you point to our `m_our_service` variable:

```
our_service_init (&m_our_service);
```

Step 3: Add UUIDs to BLE stack table

Look up the definition of `our_service_init()` in `our_service.c`. As you can see there is almost no code so we have some work to do. We must first create a UUID for our service.

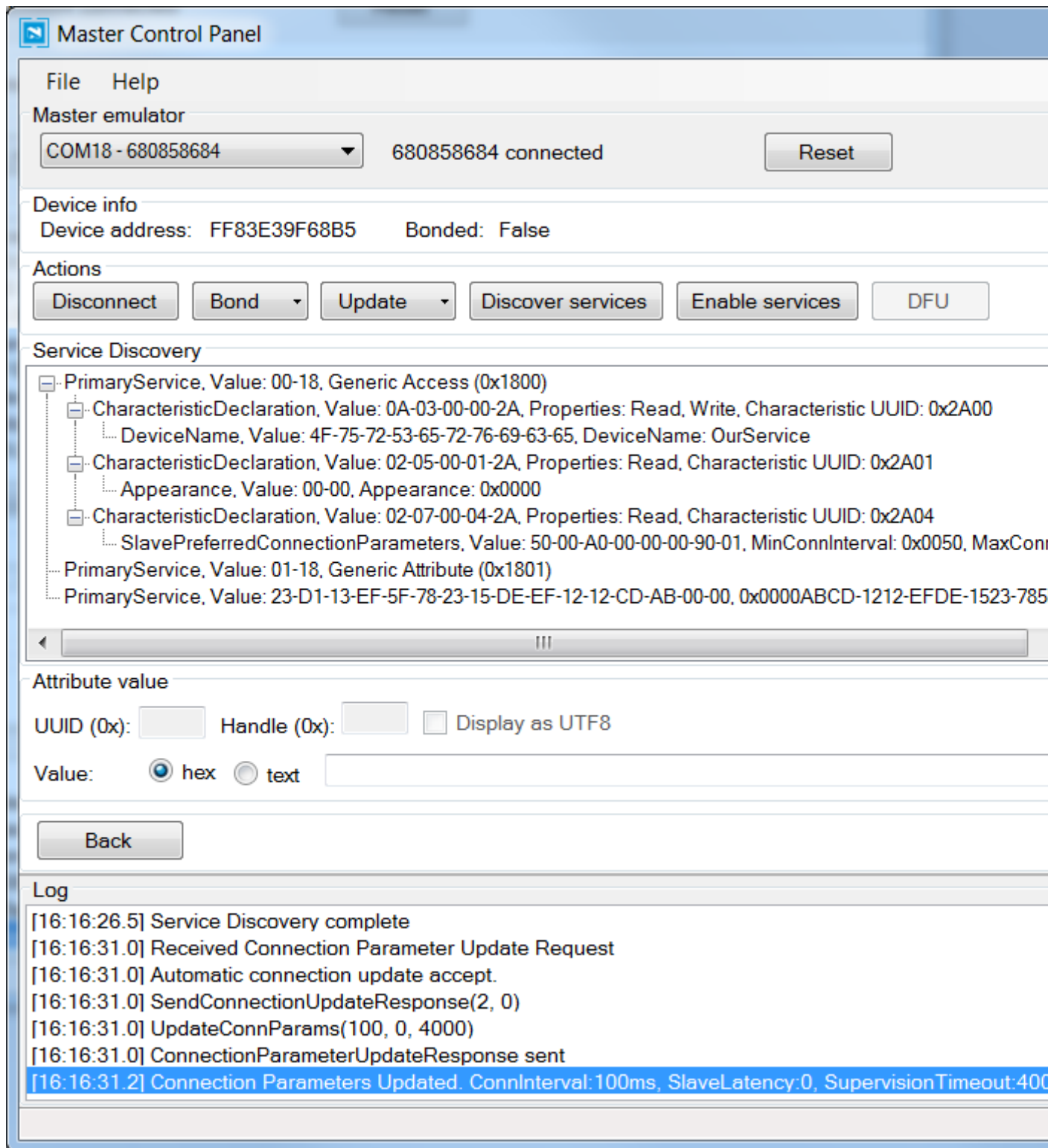

```

                                &p_our_service->service_handle);
APP_ERROR_CHECK(err_code);

SEGGER_RTT_WriteString(0, "Exectuing our_service_init().\n");
SEGGER_RTT_printf(0, "Service UUID: 0x%#04x\n", service_uuid.uuid);
SEGGER_RTT_printf(0, "Service UUID type: 0x%#02x\n",
service_uuid.type);
SEGGER_RTT_printf(0, "Service handle: 0x%#04x\n", p_our_service-
>service_handle);
}

```

Compile, download your code, and open MCP again. Hit connect and do another service discovery. Now you should see our service with its custom UUID at the bottom. You can recognize the base UUID from the #define in our_service.h and if you look closely you should also recognize our 16-bit service UUID: 0000 **ABCD** -1212-EFDE-1523-785FEF13D123.



If you have opened the Segger RTT you will also see some information about the application flow. **Remember to uncomment the four SEGGER_RTT lines at the bottom of our_service_init().** When the application starts you can see what values are used in our service. For example, the service handle is set to 0x000C. If you highlight our service in MCP you can recognize this value:

The screenshot shows a configuration window for a Bluetooth service. The top section lists service details:

- PrimaryService, Value: 01-18, Generic Attribute (0x1801)
 - CharacteristicDeclaration, Value: 20-0A-00-05-2A, Properties: Indicate, Characteristic UUID: 0x2A05
 - ServiceChanged, (No values read)
 - ClientCharacteristicConfiguration, Value: 00-00, CharacteristicConfigurationBits: None (0x0000)

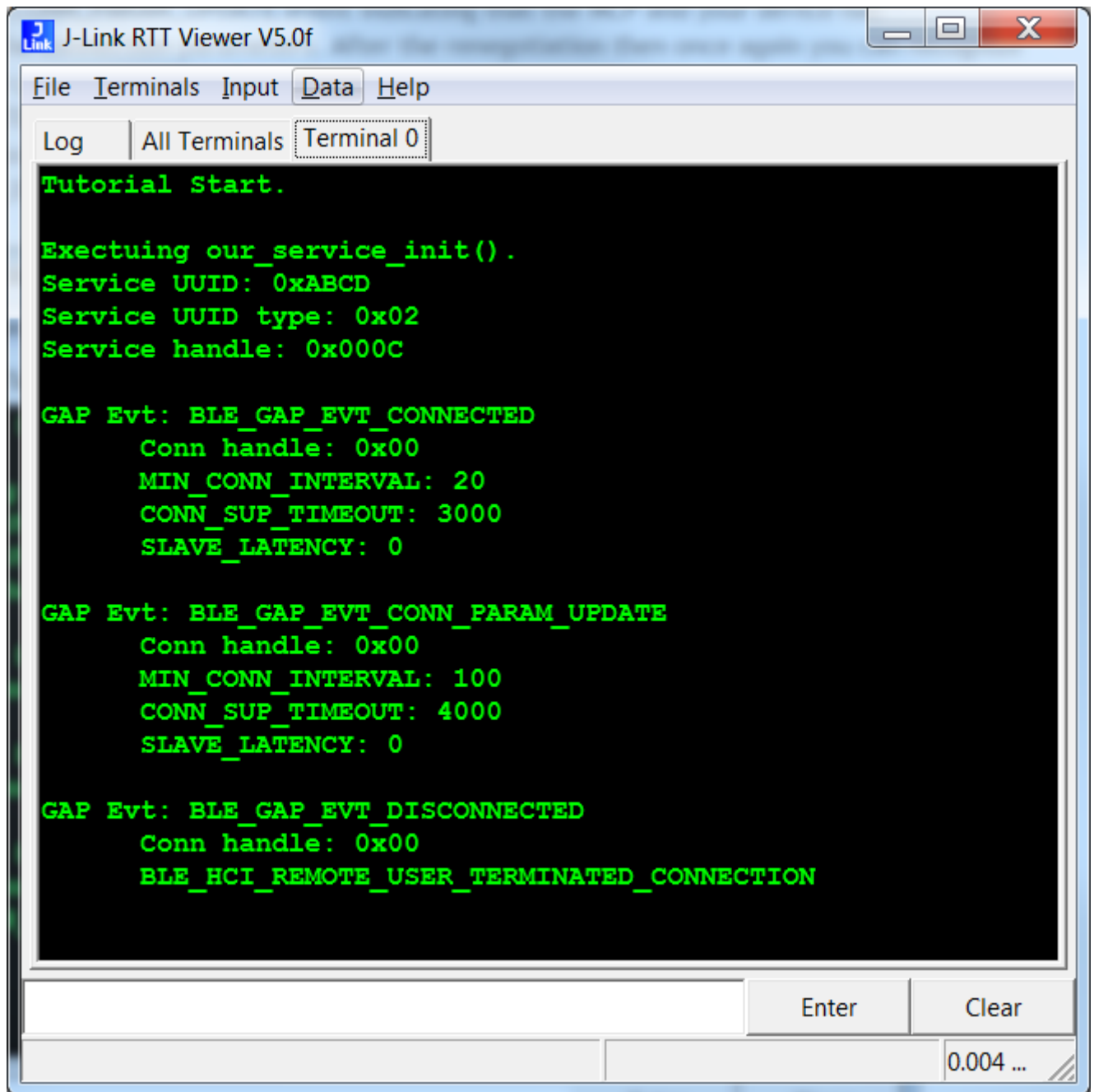
 The second entry, 'PrimaryService, Value: 23-D1-13-EF-5F-78-23-15-DE-EF-12-12-CD-AB-00-00, 0x0000ABCD-1212-EFDE-1523-785...', is highlighted with a blue selection bar.

 Below this, the 'Attribute value' section contains:

- UUID (0x): 2800
- Handle (0x): 000C (This field and its label are circled in blue)
- ☐ Display as UTF8
- Value: ☒ hex ☐ text 23-D1-13-EF-5F-78-23-15-DE-EF-12-12-CD-AB-00-00

 At the bottom left is a 'Back' button.

When you connect to your device using MCP you will also see what events occur in the BLE stack. These events are "captured" in the `on_ble_evt()` function in `main.c`. The first event is a Generic Access Profile (GAP) event, `BLE_GAP_EVT_CONNECTED`, indicating that a connection has been set up with connection handle value `0x00`. If you ever make an application that has several connections you will get several connection handles, each with a unique handle value. After a few seconds you will get the `BLE_GAP_EVT_CONN_PARAM_UPDATE` event indicating that the MCP and your device have renegotiated the connection parameters. After the renegotiation you can once again recognize the connection parameter values from the `#defines` in the example, `MIN_CONN_INTERVAL`, `SLAVE_LATENCY`, and `CONN_SUP_TIMEOUT`. [Here](#) is a short explanation of the negotiation process. Finally, when you hit disconnect in MCP you will receive the `BLE_GAP_EVT_DISCONNECTED` event and the reason for the disconnect.

The image is a screenshot of the J-Link RTT Viewer V5.0f application window. The window has a menu bar with 'File', 'Terminals', 'Input', 'Data', and 'Help'. Below the menu bar, there are tabs for 'Log', 'All Terminals', and 'Terminal 0'. The main area is a black terminal window with green text. The text shows the execution of a service initialization function, followed by connection events and parameters, and finally a disconnection event. At the bottom of the window, there are 'Enter' and 'Clear' buttons, and a status bar showing '0.004 ...'.

```
J-Link RTT Viewer V5.0f

File Terminals Input Data Help

Log All Terminals Terminal 0

Tutorial Start.

Exectuing our_service_init().
Service UUID: 0xABCD
Service UUID type: 0x02
Service handle: 0x000C

GAP Evt: BLE_GAP_EVT_CONNECTED
  Conn handle: 0x00
  MIN_CONN_INTERVAL: 20
  CONN_SUP_TIMEOUT: 3000
  SLAVE_LATENCY: 0

GAP Evt: BLE_GAP_EVT_CONN_PARAM_UPDATE
  Conn handle: 0x00
  MIN_CONN_INTERVAL: 100
  CONN_SUP_TIMEOUT: 4000
  SLAVE_LATENCY: 0

GAP Evt: BLE_GAP_EVT_DISCONNECTED
  Conn handle: 0x00
  BLE_HCI_REMOTE_USER_TERMINATED_CONNECTION

Enter Clear

0.004 ...
```

Advertising

In the previous tutorial, "[BLE Advertising, a beginner's tutorial](#)", we discussed various aspects of the advertising packet and now it is time to advertise our base UUID. Since the base UUID is 16 bytes long and the advertising packet already contains some data there won't be enough space in the advertising packet itself. Therefore we will need to put it in the scan response packet instead.

Step 5: Declare variable holding our service UUID

Inside the `advertising_init()` function in `main.c` declare a variable holding our service uuid like this:

```
ble_uuid_t m_adv_uuids[] = {BLE_UUID_OUR_SERVICE,
BLE_UUID_TYPE_VENDOR_BEGIN};
```

BLE_UUID_OUR_SERVICE is, as you know, our service UUID and BLE_UUID_TYPE_VENDOR_BEGIN indicates that it is a part of a vendor specific base UUID. More specifically BLE_UUID_TYPE_VENDOR_BEGIN is an index pointing to our base UUID in the table of UUIDs that we initiated in our_service_init().

Step 6: Declare and instantiate the scan response

Declare and instantiate the scan response:

```
ble_advdata_t srdata;
memset(&srdata, 0, sizeof(srdata));
```

Then add the UUID to the scan response packet like this:

```
srdata.uuids_complete.uuid_cnt = sizeof(m_adv_uuids) /
sizeof(m_adv_uuids[0]);
srdata.uuids_complete.p_uuids = m_adv_uuids;
```

Step 7: Include scan response packet in advertising

Finally initiate advertising with scan response:

```
err_code = ble_advertising_init(&advdata, &srdata, &options, on_adv_evt,
NULL);
```

advertising_init() should now look something like this:

```
static void advertising_init(void)
{
    uint32_t      err_code;
    ble_advdata_t advdata;

    // Build advertising data struct to pass into ble_advertising_init().
    memset(&advdata, 0, sizeof(advdata));

    advdata.name_type          = BLE_ADVDATA_FULL_NAME;
    advdata.flags              =
BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE;

    ble_adv_modes_config_t options = {0};
    options.ble_adv_fast_enabled  = BLE_ADV_FAST_ENABLED;
    options.ble_adv_fast_interval = APP_ADV_INTERVAL;
    options.ble_adv_fast_timeout  = APP_ADV_TIMEOUT_IN_SECONDS;

    // OUR_JOB: Create a scan response packet and include the list of UUIDs
    ble_uuid_t m_adv_uuids[] = {BLE_UUID_OUR_SERVICE,
BLE_UUID_TYPE_VENDOR_BEGIN};

    ble_advdata_t srdata;
    memset(&srdata, 0, sizeof(srdata));
    srdata.uuids_complete.uuid_cnt = sizeof(m_adv_uuids) /
sizeof(m_adv_uuids[0]);
    srdata.uuids_complete.p_uuids = m_adv_uuids;
```

```

    err_code = ble_advertising_init(&advdata, &srdata, &options,
on_adv_evt, NULL);
    APP_ERROR_CHECK(err_code);
}

```

Compile and download the code again. Our device should now show up like this in MCP's list of discovered devices:



Summary

So now you know how to setup and create your first basic service. If you want to add more services you can easily just replicate the `our_service_init()` function and define more service UUIDs.

But wait! We are only halfway there you might say. Where are the characteristics supposed to hold all your data? That is the topic of this tutorial: [BLE Characteristics, a beginner's tutorial](#).