



# Getting Started with Silicon Labs Bluetooth SDK (v2.11.0 or later)

---

This training gives introduction to the usage of Silicon Labs Bluetooth SDK. The most important use cases are demonstrated with sample applications, and the most important API commands are introduced while adding them to the sample applications. The basics can be learned by implementing a GATT server, and advanced skills can be acquired by implementing a GATT client.

## KEY FEATURES

- Starting Bluetooth Development
- Implementing GATT server
- Implementing GATT client

## 1 Bluetooth Basics

The Bluetooth connection is an asymmetric connection between a central device (e.g. a smartphone) and a peripheral device (e.g. a sensor). Typically the central device is the client that queries data from the peripheral device which is in this case a server.

The most common use case is the following:

- 1) The peripheral device is advertising itself
- 2) The central device is scanning for devices and finds the peripheral device
- 3) The central device initiates a connection
- 4) The central device discovers the database of the peripheral device
- 5) The central device reads/writes the database of the peripheral device (e.g. reads sensor data).

However

- The central device can also advertise and the peripheral device can also initiate connection
- The peripheral device can also discover/read/write the database of the central device

In this latter case the central device is the server and the peripheral device is the client.

All Bluetooth devices that implement server functionality (basically all Bluetooth devices) have to implement a so called GATT database. This database has a fix structure, which cannot be changed during a connection, and most commonly it is not changed during the lifetime of the device. When a client connects to the server, the first step is the discovery of this database. When this is done

- The client can read the attributes of the database
- The client can write the attributes of the database
- The server can send notification, that some attribute has changed

In the following we demonstrate how to implement the key steps of a Bluetooth connection, like

- Advertising
- Scanning
- Discovering remote database
- Reading / writing remote database

with Silicon Labs Bluetooth SDK.

Note: All Bluetooth connections have a master and a slave device. This is, however, not to be confused with the server and client roles. The master and slave roles are used in the physical layer (master is who sends packets first), while server and client roles are used in the application layer (client queries data from the server). These roles are independent from each other.

## 2 Start Development

### 2.1 Prepare your device

This training assumes you already have installed Simplicity Studio with Bluetooth stack on your computer. If you haven't done so yet, please follow the instructions of *QSG139: Bluetooth Development with Simplicity Studio*.

Before creating a Bluetooth project, it is important to note, that all Bluetooth project assumes, that you have already flashed a bootloader to your device. Without a bootloader Bluetooth projects will not start. Hence, if you have not flashed a bootloader yet, follow the procedure described in this section. If you have, then you can skip this section.

You can either flash

- a dummy bootloader or
- a Gecko Bootloader

The dummy bootloader will do nothing, but start your application (it jumps directly to the application area). You can find the image of a dummy bootloader in the following directory:

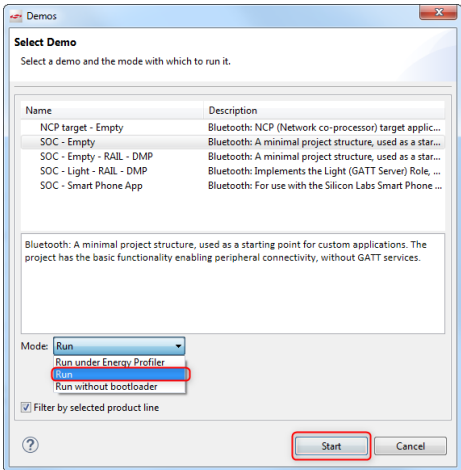
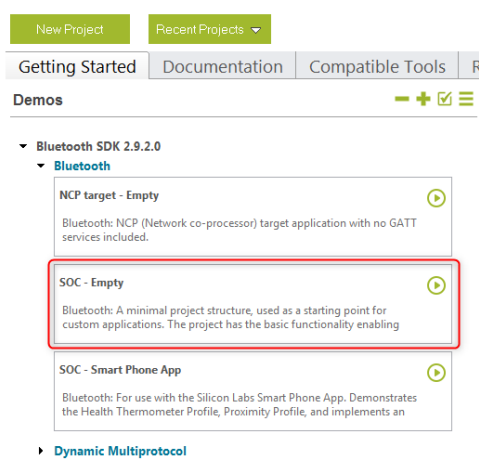
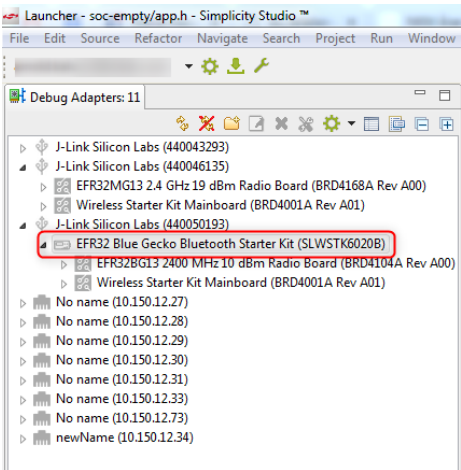
```
C:\SiliconLabs\SimplicityStudio\v4\developer\sdk\gecko_sdk_suite\v2.5\platform\bootloader\util\bin
```

Pick the one corresponding to your part and flash it to your device using Simplicity Commander.

Gecko Bootloader has many features including firmware update via UART and OTA (over-the-air), as described in *UG266: Silicon Labs Gecko Bootloader User's Guide*. The recommended Gecko Bootloader configuration for Bluetooth applications is *Bluetooth in-place OTA DFU Bootloader*. The easiest way to flash this bootloader to your device is by starting the *SoC – Empty* demo from Simplicity Studio. Demos will flash both a bootloader and an application to your device.

To start the *SoC – Empty* demo:

- 1) Open Simplicity Studio
- 2) Connect your device
- 3) Select your device on the Debug Adapters tab
- 4) Check that the Preferred SDK contains Bluetooth SDK v2.11.0 or later
- 5) Click on SOC – Empty in the Demos column
- 6) Select "Run" mode
- 7) Click Start

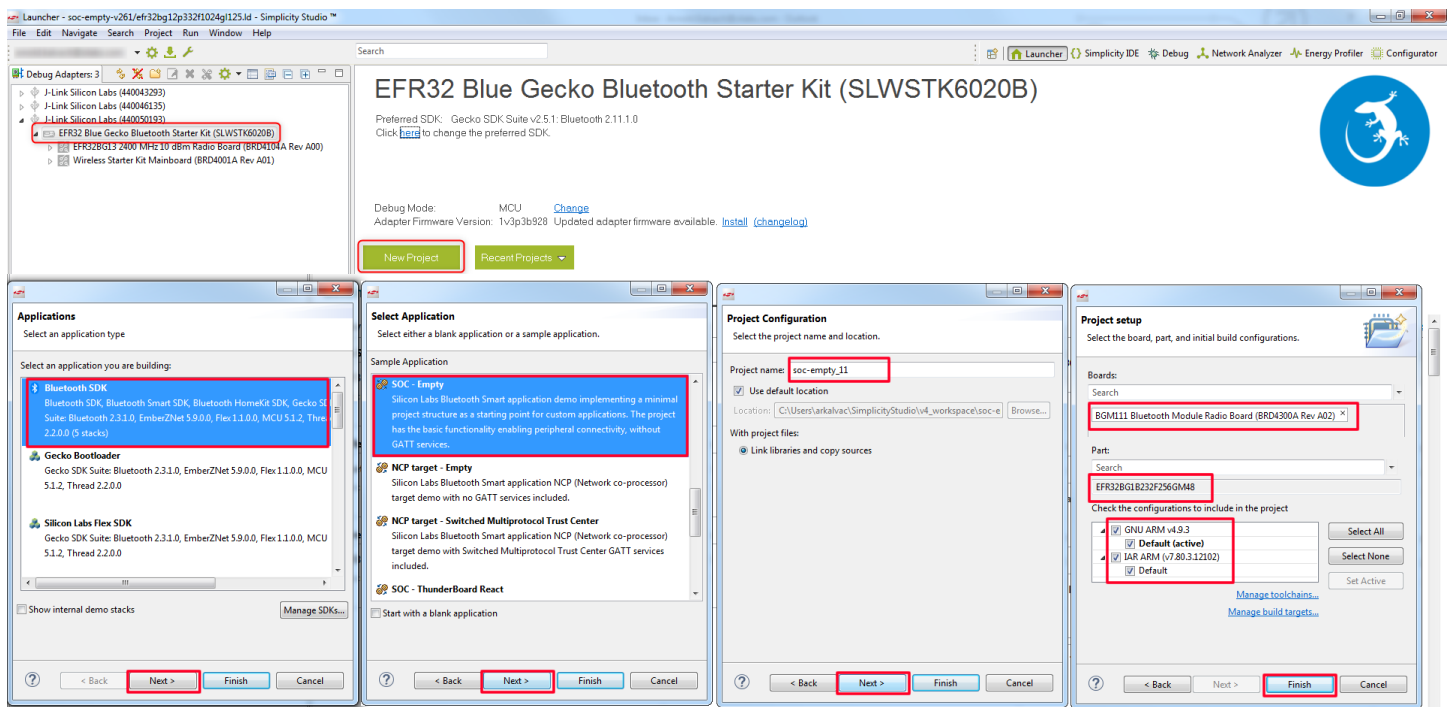


## 2.2 Create a new Bluetooth project

When starting development, it is strongly recommended to start with a Software Example project instead of building up a project from the beginnings.

For a basic setup use the SoC – Empty example.

- 1) Open Simplicity Studio
- 2) Connect your device
- 3) Select your device on the Debug Adapters tab (or on the My Products tab)
- 4) Click on New Project
- 5) Select Bluetooth SDK, click Next
- 6) If you have multiple versions of Bluetooth SDK installed, select the one you want to use on the next window, click Next
- 7) Select SOC – Empty example, click Next
- 8) Name your project
- 9) Check your device. If you want to develop for another device, you can change the target device here
- 10) Select the toolchain you want to use (IAR / GCC).



Alternatively


- 1) Open Simplicity Studio
- 2) Connect your device
- 3) Select your device on the Debug Adapters tab
- 4) Check that the Preferred SDK contains Bluetooth SDK v2.11.0 or later
- 5) Click on SOC – Empty in the Software Examples column

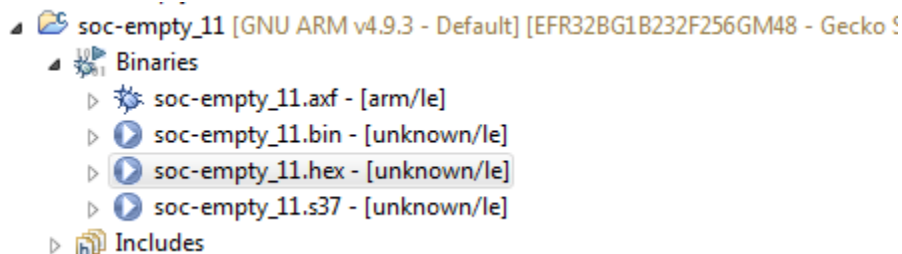
## 2.3 Add Debug Messages

The easiest way to display debug messages is using the UART interface. If you have a WSTK, the UART interface of your device can be easily connected to your PC via USB using virtual COM port (VCOM). Since Bluetooth SDK v2.11.0, the *SoC – Empty* software example is prepared for logging debug messages via the virtual COM port of the WSTK. To enable debug messages, simply open *app.h* in your *SoC – Empty* project and define `DEBUG_LEVEL` to any value greater than 0. To add custom debug messages to the code, use the `printLog()` function the same way as you would use `printf()`.

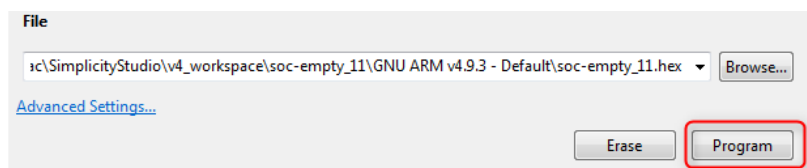
## 2.4 Build and flash your code

The *SoC – Empty* example is ready to build. To build and upload your project to your device




- 1) Click the build icon 
- 2) Find the `.hex` file in your project in the Binaries directory



- 3) Right click and select **Flash to Device**
- 4) Click **Program**



Alternatively

- 1) Click the Debug icon 
- 2) After the project was built and uploaded click the start button 
- 3) If you do not want to debug, disconnect your device with the disconnect button . The code is still running on the device.

Now if you open the COM port on your PC with a terminal program (e.g. TeraTerm) you should see the debug messages printed.

## 3 Implementing the Server Side

### 3.1 The GATT Database

Every Bluetooth connection has a GATT client and a GATT server. The server holds a GATT database: a collection of Characteristics that can be read and written by the client. The Characteristics are grouped into Services, and the group of Services determines a Bluetooth Profile.

If you are implementing a GATT server (typically on the peripheral device), you have to define a GATT database structure. This structure cannot be modified during runtime (except that some services/characteristics can be temporarily disabled), so it has to be designed in advance. If you are implementing a GATT client you can leave the GATT database as it is.

When creating a new project, or when opening the .isc file in a project, the BLE GATT configurator automatically opens. The GATT configurator is a simple-to-use tool to help you build your own GATT database. A list of predefined Profiles/Services/Characteristics/Descriptors is shown in a pane in the upper left and your current GATT database structure is shown in a pane in the upper right. An options menu is provided to the right of the Database pane.

Click an item in the Database pane to see and modify its settings in a pane in the lower right. To add a Profile/Service/Characteristic/Descriptor to your database, simply drag and drop it from the list to your database.

To get more information about a Profile/Service/Characteristic/Descriptor, click it either in the list or in your database. The description is displayed in the lower-left pane. You can find a detailed description of any Profile/Service/Characteristic/Descriptor on <https://www.bluetooth.com/specifications/gatt>.

To learn more about the GATT configurator, see *UG365: GATT Configurator User's Guide*.

The screenshot shows the Silicon Labs Bluetooth Smart Framework BLE GATT Configurator. The interface is divided into several panes:

- Source filters:** Includes checkboxes for SIG, Apple HomeKit, and Silabs.
- Navigation tabs:** Profiles, Services, Characteristics (selected), and Descriptors.
- Characteristics list:** A scrollable list of predefined characteristics, with 'Current Time' highlighted.
- Database structure:** A tree view showing the current GATT database structure, including 'Custom BLE GATT', 'Generic Access', 'Device Information', and 'Silicon Labs OTA'.
- Characteristic details:**
  - General settings:** Name: Manufacturer Name String, User description.
  - Characteristic settings:** ID, UUID: 2A29, SIG type: org.bluetooth.characteristic.manufact.
  - Value settings:** Value: Silicon Labs, Value type: utf-8, Length: 12 byte, Variable length: unchecked.
  - Properties table:**

Name	Requirement	State	
Read	Mandatory	True	
Const	Optional	True	

### 3.2 Advertising

In order to be able to create connection between two Bluetooth devices, one of the devices has to advertise itself. This has two purposes:

- To see which devices are in range
- Connection can be requested only from a device that is currently advertising

The advertisement packet is a 31 byte packet that usually contains the name of the advertiser device and the UUIDs of the most important services it has in its database. The advertisement packet is automatically assembled by the stack based on the GATT database – unless user type advertisement mode is selected.

To change the device name that will be advertised

- 1) **Open the GATT configurator**
- 2) **Select the Device Name characteristic under Generic Access service.**
- 3) **Change the Value field from “Empty Example” e.g. to “Bob’s device”. Use a custom name.**
- 4) **Change the Length field to the length of the device name. E.g. to 12 in case of “Bob’s device”**
- 5) Click Generate

To change the services to be advertised

- 1) **Open the GATT configurator**
- 2) **Select the service to be advertised: select Silicon Labs OTA service.**
- 3) **Tick the Advertise service checkbox**
- 4) **Click Generate**

To start advertisement:

- 1) Set the advertisement interval and duration:

```
gecko_cmd_le_gap_set_advertise_timing(0,160,160,0,0);
```

This will set the advertisement interval to 100ms and the duration to infinite. For more info read the *Bluetooth Smart Software API Reference Manual*.

- 2) Start advertising in discoverable and connectable mode:

```
gecko_cmd_le_gap_start_advertising(0, le_gap_general_discoverable,
le_gap_undirected_connectable);
```

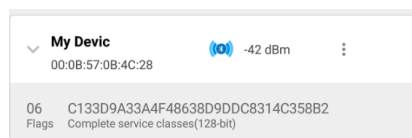
**These steps are already implemented in SOC – Empty example project in the boot event, so you do not have to modify the code!**

You can stop advertising with the following command:

```
gecko_cmd_le_gap_stop_advertising(0);
```

**Note:** the advertisement is automatically stopped when a connection has been established! If you want to connect to multiple devices, you have to restart advertisement upon each connection establishment.

**After modifications, build and flash your project to the device again, and find your device with the Blue Gecko smartphone app. Now you should see your device name changed to e.g. “Bob’s De”. The “vice” is missing from the end, because you are advertising a service with 128bit UUID which uses up 16 byte from the 31-byte advertisement packet. Flags uses 3 more bytes, device name header uses 2 more bytes and service uuid header uses 2 more bytes again. This leaves only 8 bytes for the device name!**



### 3.3 Add a predefined service

Bluetooth SIG has defined a number of services with assigned 16bit UUIDs that can be used by any devices to provide interoperability between them. E.g. if you want to set the current time on the device, it is suggested to add the predefined Current Time Service.

To add the predefined service (group of characteristics):

- 1) Open GATT configurator by double clicking on the .isc file of the project
- 2) Select the Services tab on the left pane
- 3) Drag and drop the service from the left pane to the right pane. E.g. add Current Time Service to your GATT database

To make Current Time characteristic writable (needed if you want to set time on the device):

- 4) Select Current Time characteristic in your database
- 5) Click on the State of the Write property in the lower left pane
- 6) Set it to True
- 7) Set the Length to 10
- 8) Set the Value to D0070101000000000000 (which corresponds to 2000-01-01 00:00:00)






The screenshot displays the Bluetooth SDK BLE GATT Configurator interface. The top bar shows the project name and tabs for source files. The main window is titled "BLE GATT Configurator" and has a "General" tab selected. On the left, the "Services" tab is active, showing a list of services. The "Current Time Service" is selected and highlighted with a red box and arrow (3). On the right, the "Custom BLE GATT" tree shows the "Current Time Service" expanded, with the "Current Time" characteristic selected (4). The bottom pane shows the configuration for the "Current Time" characteristic. The "Value settings" section has the "Value" field set to "D0070101000000000000" (7) and the "Length" set to "10" (7). The "Properties" table at the bottom right shows the "Write" property state set to "True" (6) and the "Write Without Response" property state set to "False" (5).

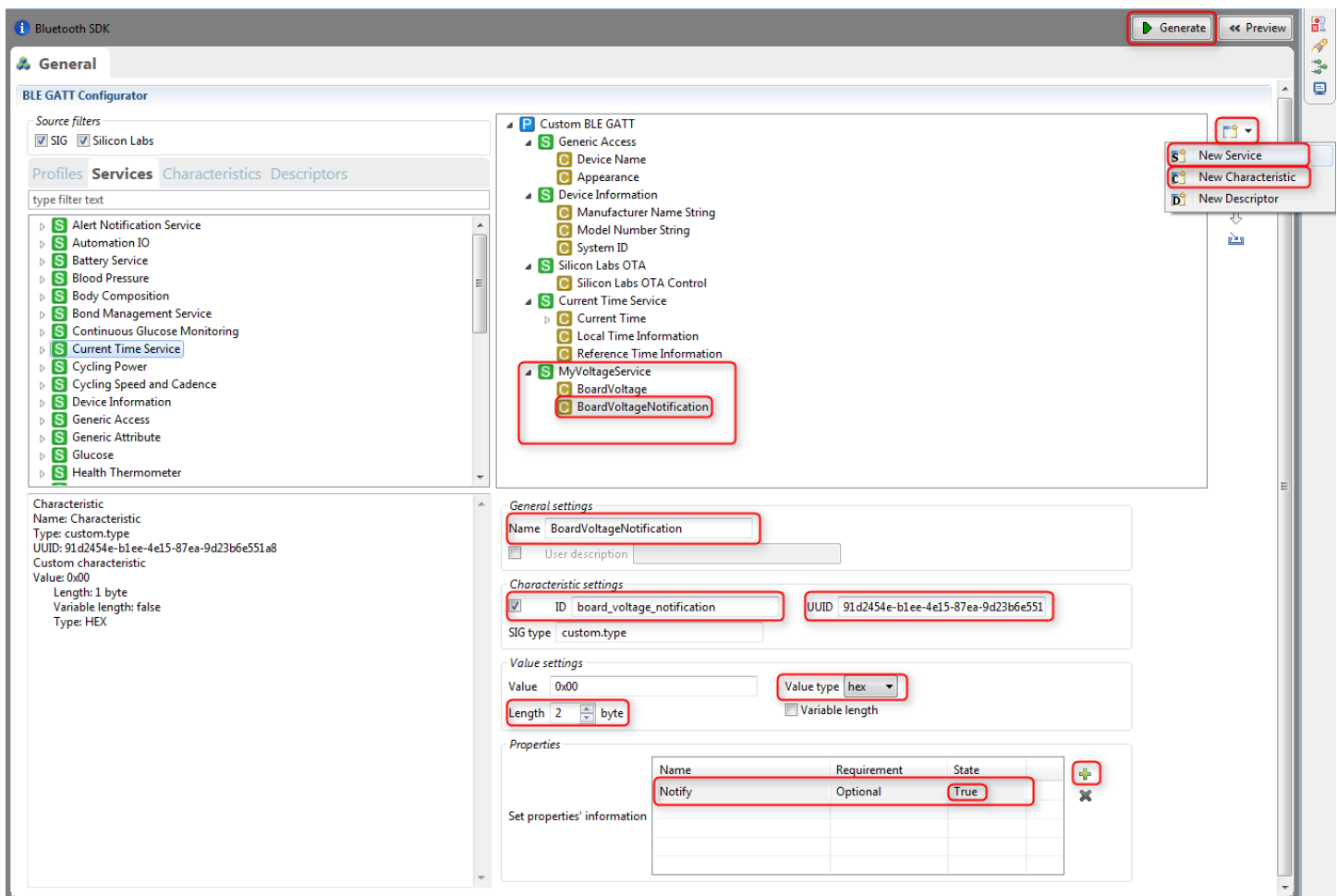


### 3.4 Add a custom service

Often you need a characteristic that you cannot find among the predefined ones. Let us say you want to read the voltage of your board in millivolts. In this case you can create custom characteristics within a custom service.

To define a custom characteristic for this specific case do the followings:

- 1) **Open GATT configurator by double clicking on the .isc file of the project**
- 2) **Click the create new item icon** 
- 3) **Click New Service**
- 4) **Select the new Service and rename it in the name field, e.g. MyVoltageService**
- 5) **Note down the 128-bit UUID of your service for future reference**
- 6) **Click the create new item icon** 
- 7) **Click New Characteristic**
- 8) **Select the new Characteristic and rename it in the name field, e.g. BoardVoltage**
- 9) **Tick the checkbox near to ID, and give it an ID, e.g. board\_voltage**
- 10) **Set the length field to 2 (16bit will be enough to describe the voltage about 3300)**
- 11) **Set the type field to hex**
- 12) **Add Read property by clicking Add new item (  ) in the Properties tab , and selecting Read**
- 13) **Set the state of the Read property to True**
- 14) **Click again the create new item icon** 
- 15) **Click New Characteristic**
- 16) **Select the new Characteristic and rename it in the name field, e.g. BoardVoltageNotification**
- 17) **Tick the checkbox near to ID, and give it an ID, e.g. board\_voltage\_notification**
- 18) **Set the length field to 2**
- 19) **Set the type field to hex**
- 20) **Add Notify property by clicking Add new item (  )in the Properties tab , and selecting Notify**
- 21) **Set the state of the Notify property to True**



### 3.5 Generate Database

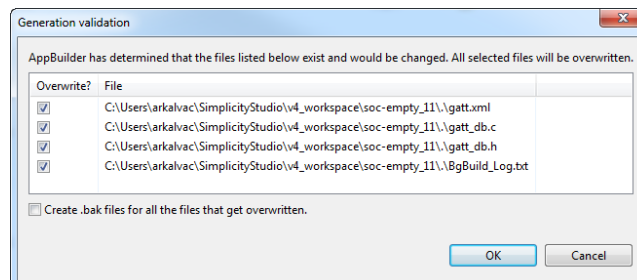
When you finished editing the database, **click Generate in the upper-right corner of the GATT editor**. This generates the following files:

**gatt.xml** – an xml format description of your database structure.

**gatt\_db.h** – a header file that contains the definitions for your characteristic handles. You can read and write the values of your characteristics by referring to these definitions. The definition names are generated from the IDs given in the GATT editor.

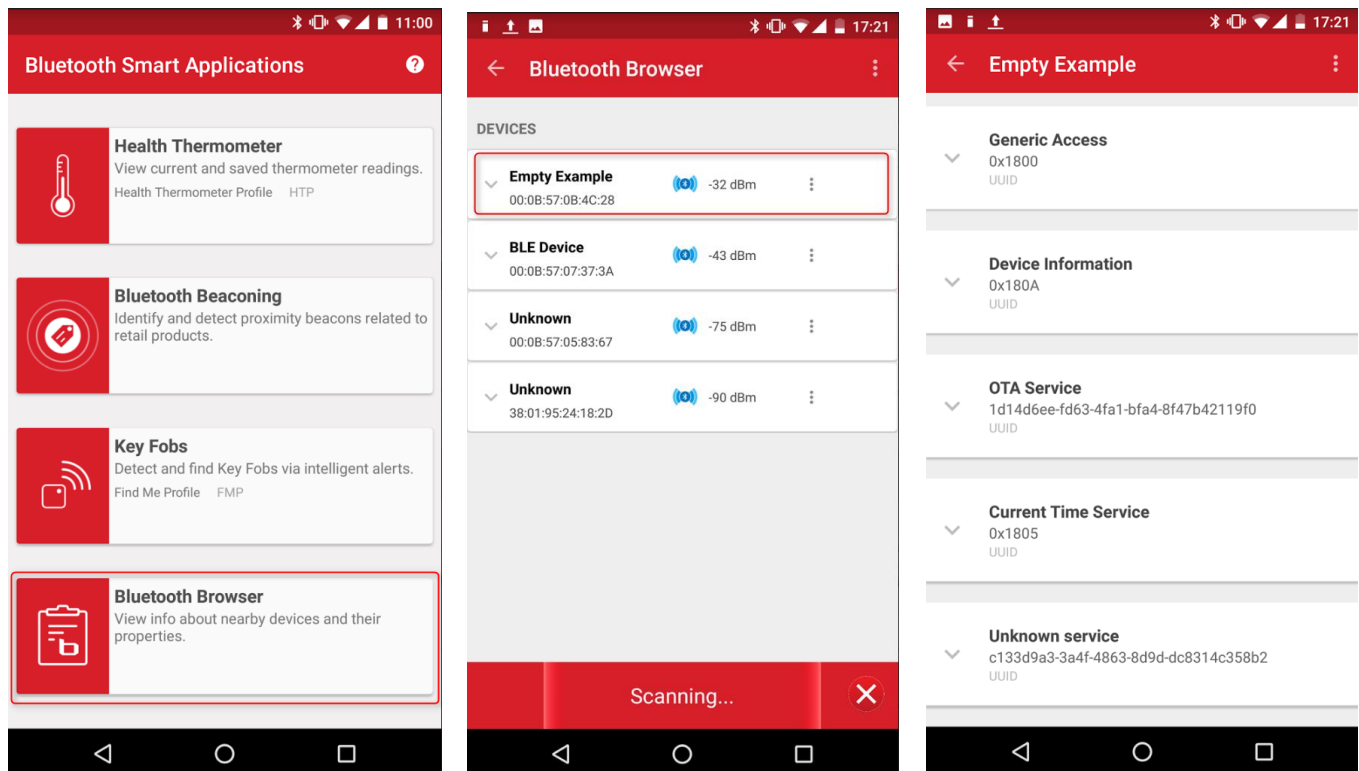
**gatt\_db.c** – a source file defining the database and the default values of the characteristics.

Your GATT database is ready for build.



To check your database, you can use your smartphone:

- 1) **Build your project and flash it to your device**
- 2) **Download the Blue Gecko app ( <http://silabs.com/bluegeckoapp> ) to your smartphone, and open it**
- 3) **Select Bluetooth Browser**
- 4) **Connect to your device (find your device name, and tap it)**
- 5) **Now you can browse your database**
- 6) **Check the entries you have added. Note, that your custom services and characteristics are listed as unknown service/characteristic, because the service/characteristic type is determined based on the UUIDs, and not on the names you are using in your database!**
- 7) **Service list is usually cached by applications. If you cannot find your newly added services, click Refresh services in the local menu while you are connected to the device.**



### 3.6 Reading/Writing the Local Database

In peripheral devices we are mostly measuring something or controlling something on the device.

To make a measurement readable by remote devices, the values has to be written into the local GATT database. Let us say we want to monitor the voltage of the board. We measure the voltage every second, and write its value to the database in our custom characteristic (BoardVoltage) every second. This value can be then read by the client any time.

We can also notify the client, that the value has changed / has been updated. We send therefore notifications via the BoartVoltageNotifi-  
cation characteristic after each measurement. Here the new value is automatically sent to the client without request.

To implement this, do the followings:

- 1) **Create a new event handler for the “connection opened” event. This event will be triggered when a device connected to our device. Find the switch() statement in appMain() in app.c and add the following case:**

```
case gecko_evt_le_connection_opened_id:
break;
```

- 2) **If the connection is opened we start measuring the voltage every second. Within the connection\_opened event handler set up a soft timer that will fire every second:**

```
case gecko_evt_le_connection_opened_id:
    gecko_cmd_hardware_set_soft_timer(32768,0,0);
break;
```

- 3) **Create a new event handler for the expired timer. This will be triggered every second:**

```
case gecko_evt_hardware_soft_timer_id:
break;
```

- 4) **Copy *em\_adc.c* and *em\_adc.h* from**

**C:\SiliconLabs\SimplicityStudio\v4\developer\sdk\gecko\_sdk\_suite\v2.5\platform\emlib\**  
**into the project and add the following line to *app.c*:**

```
#include "em_adc.h"
```

- 5) **Declare variables and initialize ADC in appMain() before `gecko_init(pconfig)` ;**

```
uint32_t adcData;
uint16 boardVoltage;
ADC_InitSingle_TypeDef initSingle = ADC_INITSINGLE_DEFAULT;
initSingle.acqTime = adcAcqTime16;
initSingle.reference = adcRef5VDIFF;
initSingle.posSel = adcPosSelAVDD;
initSingle.negSel = adcNegSelVSS;
CMU_ClockEnable( cmuClock_ADC0, true );
```

- 6) **Within the hardware\_soft\_timer event handler read the voltage**

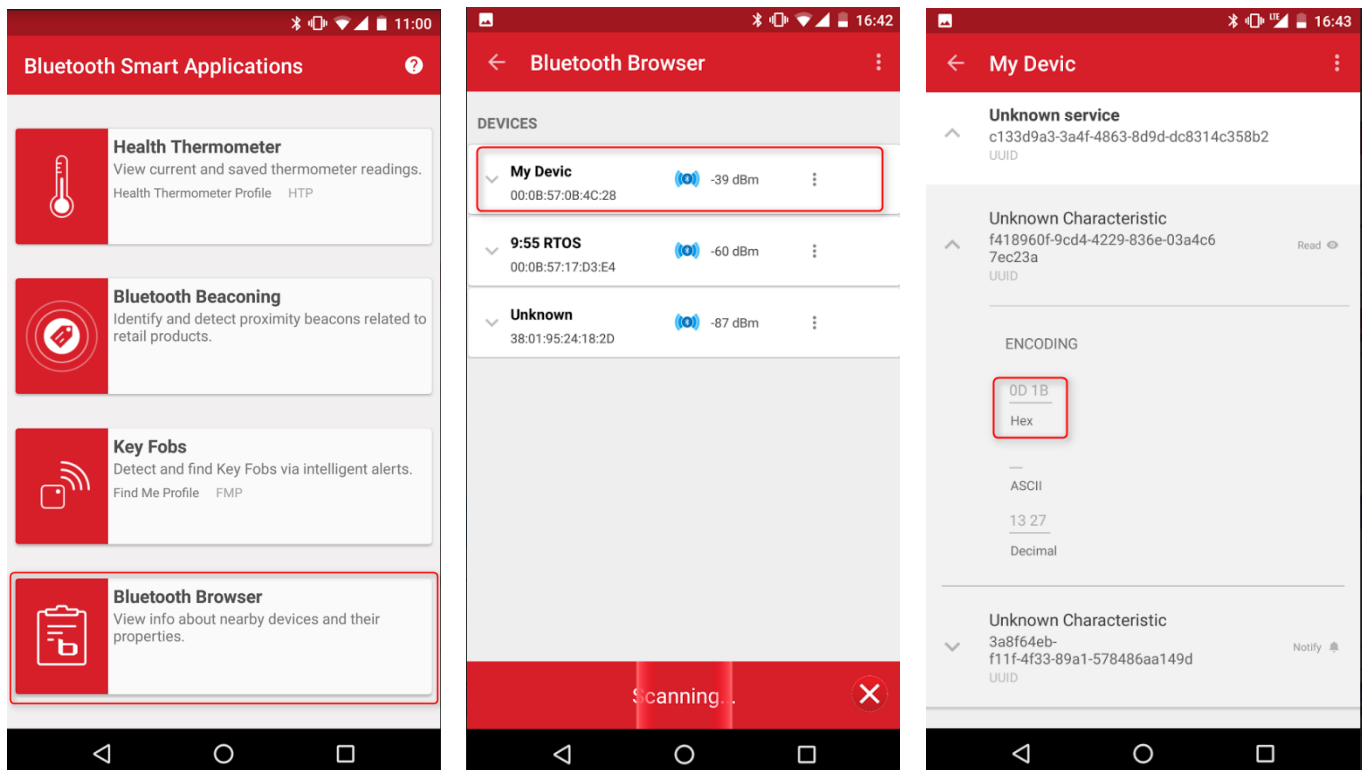
```
ADC_InitSingle(ADC0, &initSingle);
ADC_Start(ADC0, adcStartSingle);
while((ADC_IntGet(ADC0) & ADC_IF_SINGLE) != ADC_IF_SINGLE);
adcData = ADC_DataSingleGet(ADC0);
boardVoltage = (uint16)(adcData * 5000 / 4096);
printf("voltage: %d mV\r\n",boardVoltage);
```

- 7) And write it to the local database (BoardVoltage). Also send a notification with the new value (BoardVoltageNotification)

```
boardVoltage = ((boardVoltage & 0x00FF) << 8) | ((boardVoltage & 0xFF00) >> 8);
gecko_cmd_gatt_server_write_attribute_value(gattdb_board_voltage, 0, 2,
                                             (const uint8*)&boardVoltage);
gecko_cmd_gatt_server_send_characteristic_notification(0xFF,
                                                       gattdb_board_voltage_notification, 2, (const uint8*)&boardVoltage);
```

You can check the value with your smartphone.

- 1) Open the Blue Gecko app ( <http://silabs.com/bluegeckoapp> ) on your smartphone
- 2) Select Bluetooth Browser
- 3) Connect to your device
- 4) Find the unknown service with the UUID of your custom service, open it
- 5) Open the first characteristic. This will automatically read its value.
- 6) Convert the hex value to decimal, the value should be around 3300mV.
- 7) Open the second characteristic. Here you can see the voltage automatically updated every second.
- 8) Convert the hex value to decimal, the value should be around 3300mV.



To control the device / set parameters of the device you can use writable characteristic. E.g. if you want to set the current time on the device, you can write the Current Time characteristic from your smartphone. Your application will be notified about the changes, and you can read the new value from the GATT database and you can configure your device according to it. To process e.g. the updated Current Time written from your smartphone implement the followings:

1) **Create a new event handler for attribute changes:**

```
case gecko_evt_gatt_server_attribute_value_id:
break;
```

This will be triggered e.g. when a characteristic was written by a remote device

2) **Define a structure that corresponds to the Current Time characteristic structure as it is defined by Bluetooth SIG**

([https://www.bluetooth.com/specifications/gatt/viewer?attributeXmlFile=org.bluetooth.characteristic.current\\_time.xml](https://www.bluetooth.com/specifications/gatt/viewer?attributeXmlFile=org.bluetooth.characteristic.current_time.xml))

). This is important to be compatible with the standard. You can put the following code into *app.c*

```
PACKSTRUCT(struct date_time_t {
    uint16 year;
    uint8 month;
    uint8 day;
    uint8 hours;
    uint8 minutes;
    uint8 seconds;
});

PACKSTRUCT(struct day_of_week_t {
    uint8 day;
});

PACKSTRUCT(struct day_date_time_t {
    struct date_time_t date_time;
    struct day_of_week_t day_of_week;
});

PACKSTRUCT(struct exact_time_256_t {
    struct day_date_time_t day_date_time;
    uint8 fractions_256;
});

PACKSTRUCT(struct current_time_t {
    struct exact_time_256_t exact_time_256;
    uint8 adjust_reason;
});
```

3) **Process the received value within the `gatt_server_attribute_value` event handler:**

```
if (evt->data.evt_gatt_server_attribute_value.attribute == gattdb_current_time) {

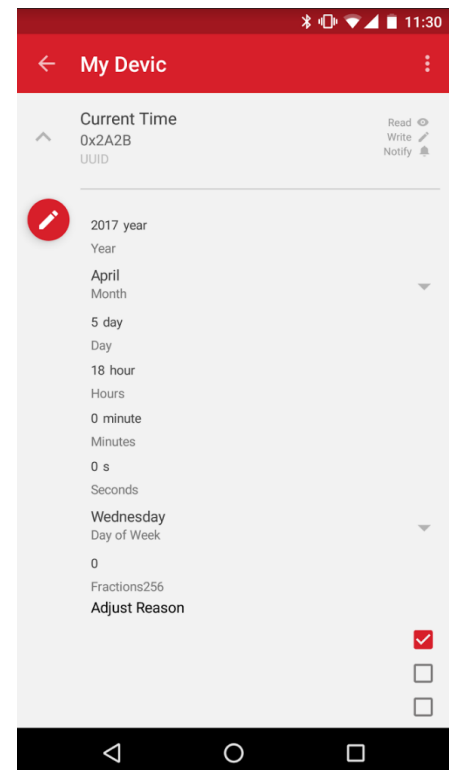
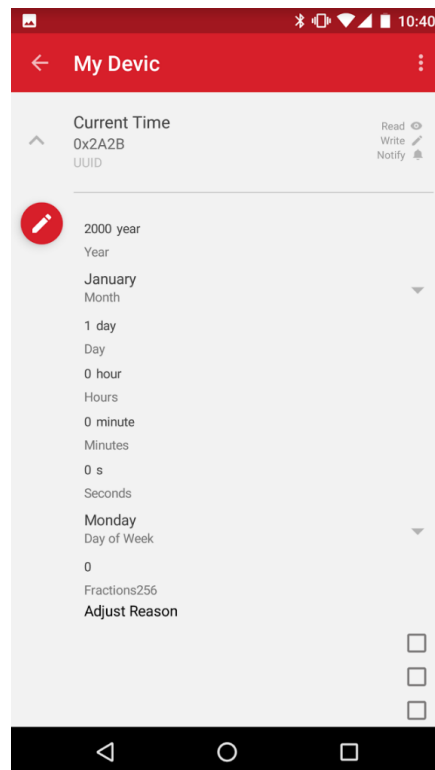
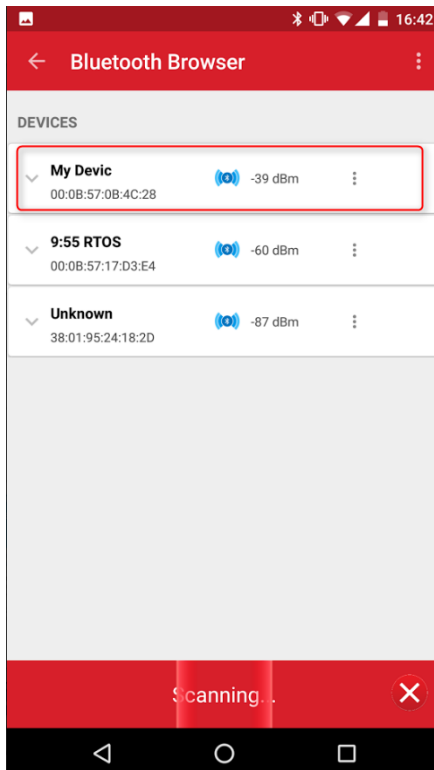
    struct current_time_t* current_time =
        (struct current_time_t*) (evt->data.evt_gatt_server_attribute_value.value.data);

    struct date_time_t* datetime =
        &(current_time->exact_time_256.day_date_time.date_time);

    printLog("current time modified: %4d-%2d-%2d %2d:%2d:%2d\r\n",
        datetime->year, datetime->month, datetime->day,
        datetime->hours, datetime->minutes, datetime->seconds);
}
```

Now you can test the working with your smartphone:

- 1) Open the Blue Gecko app ( <http://silabs.com/bluegeckoapp> ) on your smartphone
- 2) Select Bluetooth Browser
- 3) Connect to your device
- 4) Find the Current Time Service open it
- 5) Find the Current Time characteristic
- 6) Set the value to the current date
- 7) Check the value in the debug message on the COM port



```
voltage: 3356 mV
voltage: 3354 mV
current time modified: 2017- 4- 5 18: 0: 0
voltage: 3356 mV
voltage: 3356 mV
voltage: 3355 mV
```

## 4 Implementing the Client Side

The GATT server is basically only responding for requests from the client, which is relatively easy to implement. The GATT client, however, generates a series of requests, basically implementing a state machine. Hence, this section requires some more insights in the code, and **recommended only for advanced trainings**. The client code can be added to the server code, since a Bluetooth device can be used both as a server and as a client at the same time. However, to get a cleaner implementation, it is suggested starting the implementation of the client code from a new SoC-Empty project.

### 4.1 Scanning

If you are implementing a central device (a client) which will connect to an advertising peripheral device first you have to start scanning. In scanning mode the Bluetooth device is searching for nearby devices that are currently advertising in order to create a connection with one or more of them. To start scanning do the followings:

- 1) Set the scanning parameters in the system\_boot event handler in appMain() in app.c:

```
case gecko_evt_system_boot_id:
    //...
    gecko_cmd_le_gap_set_discovery_timing(le_gap_phy_lm, 160, 160);
    gecko_cmd_le_gap_set_discovery_type(le_gap_phy_lm, 1);
    break;
```

This will result in a continuous active scanning. For details see *Bluetooth Smart Software API Reference Manual*

- 2) Start scanning right after setting the setting parameters

```
case gecko_evt_system_boot_id:
    //...
    gecko_cmd_le_gap_set_discovery_timing(le_gap_phy_lm, 160, 160);
    gecko_cmd_le_gap_set_discovery_type(le_gap_phy_lm, 1);
    gecko_cmd_le_gap_start_discovery(le_gap_phy_lm, le_gap_discover_observation);
    printLog("Scanning started\r\n");
    break;
```

- 3) Set up a timer to stop scanning e.g. after 5 seconds

```
case gecko_evt_system_boot_id:
    //...
    gecko_cmd_le_gap_set_discovery_timing(le_gap_phy_lm, 160, 160);
    gecko_cmd_le_gap_set_discovery_type(le_gap_phy_lm, 1);
    gecko_cmd_le_gap_start_discovery(le_gap_phy_lm, le_gap_discover_observation);
    printLog("Scanning started\r\n");
    gecko_cmd_hardware_set_soft_timer(5*32768,1,1);
    break;
```

- 4) Create a new event handler for the expired timer:

```
case gecko_evt_hardware_soft_timer_id:
    break;
```

- 5) In the event handler stop scanning

```
case gecko_evt_hardware_soft_timer_id:
    if (evt->data.evt_hardware_soft_timer.handle == 1){
        gecko_cmd_le_gap_end_procedure();
        printLog("Scanning stopped\r\n");
    }
    break;
```



While the device is scanning for nearby devices a new `le_gap_scan_response` event is raised by the stack for each advertisement packet received. To handle these events:

1) Create an array for scanned devices in a global variable

```
#define MAX_SCANNED_DEVICES 10
struct gecko_msg_le_gap_scan_response_evt_t scanned_devices[MAX_SCANNED_DEVICES];
uint8 num_scanned_devices = 0, i, *addr;
```

2) Create an event handler for scan responses:

```
case gecko_evt_le_gap_scan_response_id:
break;
```

3) Within the event handler check if the device was already scanned, and add to the list if it was not:

```
addr = evt->data.evt_le_gap_scan_response.address.addr;
for (i=0; i<num_scanned_devices; i++)
    if (memcmp(scanned_devices[i].address.addr, addr, 6)==0)
        break;
if (i == num_scanned_devices && num_scanned_devices < MAX_SCANNED_DEVICES){
    memcpy(scanned_devices[num_scanned_devices].address.addr, addr, 6);
    scanned_devices[num_scanned_devices].address_type =
        evt->data.evt_le_gap_scan_response.address_type;
    scanned_devices[num_scanned_devices].rssi =
        evt->data.evt_le_gap_scan_response.rssi;
    printf("%d) %02x:%02x:%02x:%02x:%02x:%02x - rssi: %d\r\n", num_scanned_devices,
        addr[5], addr[4], addr[3], addr[2], addr[1], addr[0],
        scanned_devices[num_scanned_devices].rssi);
    num_scanned_devices++;
}
```

Now you can build your code, and flash it to the device. Connect to the serial port and take a look at the list of scanned devices. If you have a peripheral device in the near advertising, you have to see it listed

When scanning was completed, you can connect to any discovered device. For the sake of simplicity let us connect to the closest device, the rssi of which is at least -50dBm:

1) Declare variables:

```
int8 max_rssi;
uint8 closest_device;
```

2) Find the closest device, right after scanning was stopped in the `hardware_soft_timer` event handler:

```
max_rssi = -50;
closest_device = num_scanned_devices;
for (i = 0; i < num_scanned_devices; i++){
    if (scanned_devices[i].rssi > max_rssi){
        max_rssi = scanned_devices[i].rssi;
        closest_device = i;
    }
}
```

3) And connect to it

```
if (closest_device < num_scanned_devices)
    gecko_cmd_le_gap_open(scanned_devices[closest_device].address,
        scanned_devices[closest_device].address_type);
```

4) Declare a global variable for the connection handle

```
uint8 conn_handle;
```

5) And save the connection handle within the `le_connection_opened` event handler for future reference

```
case gecko_evt_le_connection_opened_id:  
    printLog("connected\r\n");  
    conn_handle = evt->data.evt_le_connection_opened.connection;  
break;
```

Now place a server and a client next to each other and see on the terminal if they get connected.

## 4.2 Discovering Remote Database

After the connection was established the structure of the remote database is unknown for the client. In order to discover the database, a service discovery has to be run on it. This will return with UUIDs of implemented services/characteristics and with handles of services and characteristics – which can be used as reference while reading/writing.

To start service discovery:

- 1) Declare global variables indicating service discovery state, and storing service handles:

```
uint8  service_discovery = 0;
uint32 service_handles[1] = {0xFFFFFFFF};
uint8  service_to_find[2] = {0x00,0x18};
```

- 2) Start service discovery in the connection\_parameters (connection is established) event handler:

```
case gecko_evt_le_connection_parameters_id:
    printLog("Connection established\r\n");
    gecko_cmd_gatt_discover_primary_services(conn_handle);
    service_discovery = 1;
    break;
```

- 3) Create a new event handler for discovered services:

```
case gecko_evt_gatt_service_id:
    break;
```

- 4) Save service handles within the gatt\_service event handler for services you are interested in.

E.g. for Generic Access service:

```
if (memcmp(evt->data.evt_gatt_service.uuid.data, service_to_find, 2) == 0){
    service_handles[0] = evt->data.evt_gatt_service.service;
}
```

- 5) Create a new event handler for the end of the discovery process:

```
case gecko_evt_gatt_procedure_completed_id:
    if (service_discovery){
        service_discovery = 0;
        //characteristic discovery can be started here
    }
    break;
```

To start characteristic discovery:

- 1) Declare global variables indicating characteristic discovery state, and storing characteristic handles:

```
uint8  characteristic_discovery = 0;
uint16 characteristic_handles[1] = {0xFFFF};
uint8  char_to_find[2] = {0x00,0x2a};
```

- 2) Start characteristic discovery in the gatt\_procedure\_completed event handler:

```
gecko_cmd_gatt_discover_characteristics(conn_handle, service_handles[0]);
characteristic_discovery = 1;
```

- 3) Create a new event handler for discovered characteristics:

```
case gecko_evt_gatt_characteristic_id:
break;
```

- 4) Save characteristic handles within the `gatt_characteristic` event handler for characteristics you are interested in.  
E.g. for Device Name characteristic:

```
if (memcmp(evt->data.evt_gatt_characteristic.uuid.data, char_to_find, 2) == 0){
    characteristic_handles[0] = evt->data.evt_gatt_characteristic.characteristic;
}
```

- 5) Extend the `gatt_procedure_completed` event handler:

```
else if (characteristic_discovery){
    characteristic_discovery = 0;
    //reading/writing remote database can be started here
}
```

### 4.3 Reading/Writing Remote Database

After service discovery any characteristic can be read/written, provided that you have saved the characteristic handle and you have rights to read/write the given characteristic

E.g. to read the Device Name characteristic of the remote device do the following:

- 1) Declare a receive buffer

```
uint8 remote_device_name[50];
```

- 2) Initiate read process e.g. in the `gatt_procedure_completed` event handler:

```
gecko_cmd_gatt_read_characteristic_value(conn_handle,characteristic_handles[0]);
```

- 3) Create a new event handler for the received data

```
case gecko_evt_gatt_characteristic_value_id:
break;
```

- 4) Process the received data within the event handler:

```
memcpy(remote_device_name,evt->data.evt_gatt_characteristic_value.value.data,
        evt->data.evt_gatt_characteristic_value.value.len);
remote_device_name[evt->data.evt_gatt_characteristic_value.value.len] = '\0';
printf("remote device name: %s\r\n",remote_device_name);
```

Similarly, you can write to characteristics using `gecko_cmd_gatt_write_characteristic_value(conn_handle,char_handle,len,data)`.

Now build and flash your code, place a server and a client next to each other and see on the terminal if the client is able to connect and find the device name of the server.

```
Scanning started
0) 25:d4:17:57:b:0 rssi: -63
1) 28:1c:24:f5:6b:6a rssi: -35
2) 8e:aa:5:57:b:0 rssi: -43
Scanning stopped
connected
remote device name: Nexus 5X
```