Developer Study Guide: An introduction to Bluetooth Low Energy Development

Creating a Bluetooth Low Energy peripheral using Zephyr on micro:bit

Version:	5.1
Last updated:	3rd October 2019

Contents

Revision History

Version	Date	Author	Changes
5.1.0	18th March 2019	Martin Woolley Bluetooth SIG	Initial version
5.1.1	3rd October 2019	Martin Woolley Bluetooth SIG	Changed temperature sampling to use k_delayed_work, initiated on client characteristic config descriptor set to 1. Use *west* meta tool instead of cmake to build.

Overview

This document is an introduction to the basic hardware and software setup required to begin creating a Bluetooth® Low Energy (LE) peripheral device using the Zephyr RTOS. Zephyr runs on numerous types of microcontroller. We've based this project on BBC micro:bit but with a little tweaking, you should be able to fairly easily use an alternative Zephyr board.

Goal

This document guides you through hardware assembly, coding and testing. You'll be turning your Zephyr device into a custom Bluetooth proximity device. A client application will be able to notify you when your proximity device is out of range, let you find it when you've lost it and give you a visual indication of whether you are near or far from it.

Note: This guide should not be considered a complete solution, nor the basis for any commercial product.

We'll also make the device report the ambient temperature once a second so that this can be monitored by a client application. Imagine a medical device or product which needs to be kept close at hand and at a suitable temperature.

The device will be connected to various LEDs and these will be used to indicate alarm conditions and whether the device is near or far from the connected client. A display may be available and this will be used to allow the client to tell the server device the strength of the signal it is receiving and whether this is deemed to indicate the device is near, far away or somewhere in between.

Hardware

In addition to the hardware described elsewhere, we assume you're using a BBC micro:bit and a breakout board which makes the micro:bit GPIO pins more easily available. We used this breakout board from Sparkfun, but there are many alternative products available that should meet the need: https://www.sparkfun.com/products/13989

Exercises Overview

This hands-on lab progresses through a series of exercises:

1. Setting up your hardware and software environment
2. Implementing the Bluetooth profile for Zephyr on micro:bit
3. Testing

Estimated time to complete this lab: 2 hours

Lab Conventions

From time to time you will be asked to add code to your project. You will either be given the complete set of code, e.g. a new function or if the change to be made is a relatively small change to an existing block of code, the whole block will be presented, with the code to be added or modified highlighted in **this colour**.

Zephyr Exercise 1: Setting up

Software

Follow the instructions at https://docs.zephyrproject.org/latest/getting_started/index.html to set up a Zephyr development environment on the platform of your choice.

The Zephyr Bluetooth API is documented here:
https://docs.zephyrproject.org/latest/reference/bluetooth/index.html

Initial Skeleton Code

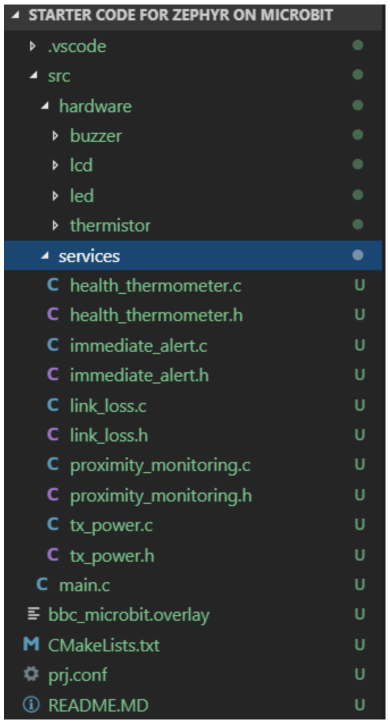
Skeleton code from which to start has been included in the study guide. You'll find it in Servers\Zephyr\Solutions\Starter Code for Zephyr on microbit.

Set up a folder to edit and build in. Copy the starter code (all files and folders) to this folder.

Create a build folder which will contain the output from the build process.

```
mkdir build
cd build
```

Open the entire folder in an editor. It should look like this:



```
STARTER CODE FOR ZEPHYR ON MICROBIT
  .vscode
  src
    hardware
      buzzer
      lcd
      led
      thermistor
    services
      health_thermometer.c        U
      health_thermometer.h        U
      immediate_alert.c           U
      immediate_alert.h           U
      link_loss.c                 U
      link_loss.h                 U
      proximity_monitoring.c      U
      proximity_monitoring.h      U
      tx_power.c                  U
      tx_power.h                  U
    main.c                        U
  bbc_microbit.overlay            U
  CMakeLists.txt                  U
  prj.conf                        U
  README.MD                       U
```

All required source files are already in place. Complete implementations of code required to interface with our hardware are in source files in the hardware\ folder. Incomplete implementations of Bluetooth services are in the services\ folder. We’ll be completing the services and other Bluetooth related code in main.c as we progress through the exercises.

Hardware

We designed a simple breadboard circuit for our lab exercises. It consists of a micro:bit, a piezo buzzer and three colored LEDs, one green, one yellow and one red, with resistors of a suitable rating in place to protect the LEDs from overloading.

Parts List

These are the parts we used. You can vary as you see fit if you’re comfortable with basic electronics.

1 x BBC micro:bit

USB cable for connecting the micro:bit to a computer

1 x red LED

1 x yellow LED

1 x green LED

3 x 56 Ohms 5% tolerance resistors

1 x ABT-402-RC PIEZO TRANSDUCER, 5V, PCB

1 x breadboard

1 x SparkFun serLCD compatible LCD display

1 x TMP36 thermistor (optional)

Wires for forming connections

PIN Connections

Feature	**Circuit Board Component**	**micro:bit pin**
Power	GND	GND pin #1
Power	3.3V power rail	3V3
LEDs	Green LED	pin 3
LEDs	Amber LED	pin 4
LEDs	Red LED	pin 5
Buzzer	Buzzer	pin 6
LCD	LCD RX pin	pin 0

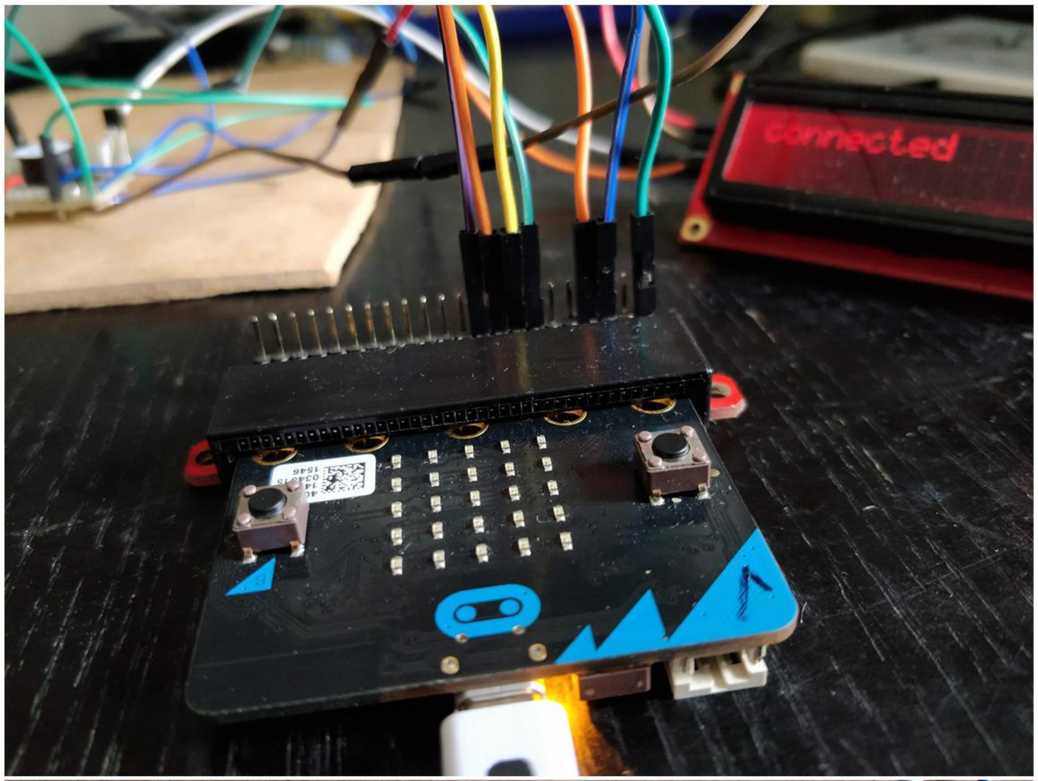
LEDs and the buzzer must also be connected to GND. LEDs should be protected by suitable resistors. The LCD display should also be connected to GND and 3.3V power.

Note: Connections for your serial LCD display may vary, depending on the precise component you're using. See https://www.sparkfun.com/categories/148

To check your connections and circuit are working as required. open main.c and look for a call to a function called io_test(). It is currently commented out. Uncomment it and build using the command *west build -b bbc_microbit* and then flash the binary to your micro:bit over USB using *west flash.*

Each of the three LEDs should flash twice in sequence, the buzzer should then shound three times. A friendly message of "hello martin" should appear on the LCD display. If you'd like to change that message…. go ahead.

When you're satisfied your hardware is connected properly, comment out the call to io_test() again.



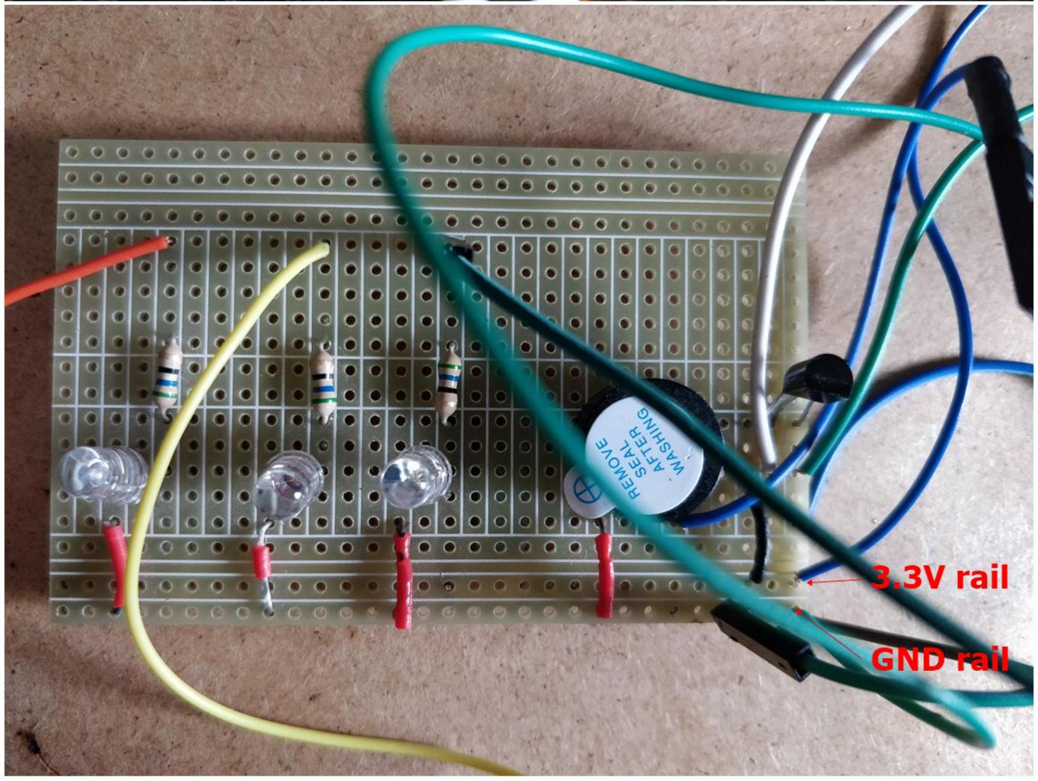


Figure 1 - micro:bit connected to LEDs, buzzer, LCD and thermistor (optional)

Zephyr Exercise 2 – Implementing and Testing the Profile

About Profile Implementation

Implementing a profile involves writing code for the target device and making use of APIs from a software framework that is provided by the stack vendor. There are no standards here but all such APIs and frameworks ultimately reflect the same, underlying technical standards from the Bluetooth SIG so it doesn't tend to take long to understand a specific vendor's APIs, armed with the relevant theoretical knowledge about, for example, services, characteristics and descriptors.

Our next job is to create the code which will implement our custom profile for Zephyr.

About Testing

We'll test as we go so that we can validate each feature as we add it, rather than have to test everything all in one go and perhaps find bigger problems to diagnose and solve than if we'd progressed with smaller increments. To test your solution, you will use a smartphone application called nRF Connect, which is available for both Android and iOS or similar.

Stack initialisation

Zephyr requires a series of steps to be carried out to enable and initialise the Bluetooth stack before we can use it. Open main.c. We'll be making a series of changes to this source file.

The *bt_enable* function enables the Bluetooth stack and makes a callback to a function with the result. After receiving this callback, we can proceed to set up the GATT skeleton services which our profile defines.

Add the following code below the comment "//TODO enable the Bluetooth stack".

```
	err = bt_enable(bt_ready);
	if (err)
	{
		return;
	}
```

bt_ready is a callback function which we must implement. Add the following underneath the comment "//TODO implement callback function for post Bluetooth stack initialisation".

```
static void bt_ready(int err)
{
	if (err)
	{
		printk("Bluetooth init failed (err %d)\n", err);
		return;
	}
```

```
        link_loss_init();
        immediate_alert_init();
        tx_power_init();
         void (*f_ptr)() = &start_sampling_temperature;
        health_thermometer_init(f_ptr);
        proximity_monitoring_init();

}
```

The function calls being made here, currently do little or nothing. We'll implement them fully later on.

GAP Advertising

We now need to define advertising packet content and start advertising.

Add the following struct below the comment "// TODO define advertising data".

```
static const struct bt_data ad[] = {
               BT_DATA_BYTES(BT_DATA_FLAGS, (BT_LE_AD_GENERAL | BT_LE_AD_NO_BREDR)),
               BT_DATA(BT_DATA_NAME_COMPLETE, DEVICE_NAME, DEVICE_NAME_LEN),
};
```

This defines the advertising packet as comprising the Flags and Complete Local Name fields using Zephyr macros. See the Bluetooth Core Specification Supplement for details of the fields which may be included in advertising packets of various types: https://www.bluetooth.com/specifications/bluetooth-core-specification

Next, we'll initiate advertising. Add the following code to the *bt_ready* function after the call to *proximity_monitoring_init()*.

```
        err = bt_le_adv_start(BT_LE_ADV_CONN, ad, ARRAY_SIZE(ad), NULL, 0);
        if (err)
        {
               printk("Advertising failed to start (err %d)\n", err);
               return;
        }

        printk("Advertising successfully started\n");
```

Checkpoint

Build with *west build -b bbc_microbit* and install on your micro:bit with *west flash*. Run the nRF Connect application on a smartphone. You should see a device with the name "BDSK" advertising. That's your micro:bit. Connect to it from nRF Connect and you should see that it already has four Bluetooth GATT services, namely Generic Attribute, Generic Access, Device Information and TX Power. We won't be doing anything special with these services but if you can see them, it's an indication that things are going well so far.

Connection State Handling

Our next job is to implement connection state handling. We want our code to know when a connection has been established and when it has been lost. Add the following code under the comment "//TODO implement callback function for post Bluetooth stack initialisation".

```
static struct bt_conn_cb conn_callbacks = {
                .connected = connected,
                .disconnected = disconnected,
};
```

This defines the functions that will be called in each of the two connection state change scenarios. They don't exist yet. We'll implement them soon.

Now add the following code under the comment "//TODO register connection state change callback functions".

```
        bt_conn_cb_register(&conn_callbacks);
```

This Zephyr API call registers out connection state change handler functions. We'll implement those two functions next, starting with the function which handles new connections having been established.

Add this code under the comment "//TODO implement connected state change callback function".

```
static void connected(struct bt_conn *conn, u8_t err)
{
        if (err)
        {
                printk("Connection failed (err %u)\n", err);
        }
        else
        {
                default_conn = bt_conn_ref(conn);
                set_connection(conn);
                write_string("connected       ", 16, 0, 0);
        }
}
```

On being informed a connection has been established, we store the Zephyr Bluetooth connection object of type bt_conn and pass it to the health_thermometer service for use in sending temperature data to a connected client. Have a look at the *start_sampling_temperature* function. You'll see that we're using simulated temperature data rather than real temperature data from a sensor. This is implemented for you in hardware/thermistor.c if you're interested in seeing what's going on there.

Now we'll implement the *disconnected* function, which we've registered to be called from the connection drops. Add the following code below the comment "//TODO implement disconnected state change callback function".

```
static void disconnected(struct bt_conn *conn, u8_t reason)
{
        printk("Disconnected (reason %u)\n", reason);
        printk("Disconnected (alert_level %d)\n", get_link_loss_alert_level());
        write_string("disconnected    ", 16, 0, 0);
        stop_sampling_temperature();
        if (default_conn)
        {
                bt_conn_unref(default_conn);
                default_conn = NULL;
        }
}
```

All we're doing in this function, is stopping the worker which is sampling the simulated temperature data and using the Zephyr API function *bt_conn_unref* to decrement a connection reference count which is maintained by the stack.

Next, we'll implement GATT services which address our list of requirements.

Requirement 2 – Link Loss, Alert Level

We need to implement the GATT link loss service and handle writes to its alert level characteristic. We also need to respond to the Bluetooth connection being lost by flashing LEDs and buzzing the piezo buzzer, if the link loss service's alert level is set to 1 or 2.

A partial implementation of the link loss service can be found in the src/services/link_loss.c and src/services/link_loss.h files. Open the .c file in your editor.

Define and Register the GATT Service

Under the comment "TODO Link Loss Service Declaration" add this code:

```
static struct bt_gatt_attr attrs[] = {
               BT_GATT_PRIMARY_SERVICE(BT_UUID_LINK_LOSS),
               BT_GATT_CHARACTERISTIC(BT_UUID_ALERT_LEVEL, BT_GATT_CHRC_READ |
BT_GATT_CHRC_WRITE,

        BT_GATT_PERM_READ | BT_GATT_PERM_WRITE, read_link_loss_alert_level,
write_link_loss_alert_level, NULL),
};


static struct bt_gatt_service link_loss_svc = BT_GATT_SERVICE(attrs);
```

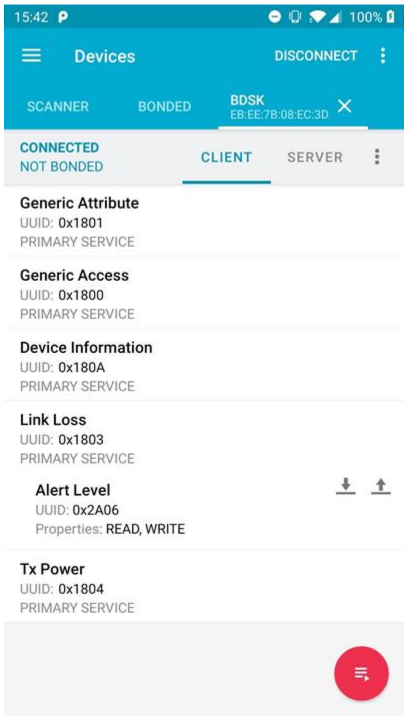
As you can see, Zephyr provides a type, *bt_gatt_attr* which allows a GATT attribute of some sort to be specified. Here we're defining an array of GATT attributes, using Zephyr macros to include the GATT link loss service, whose UUID of 0x1803 was already defined in the skeleton code, followed by the alert level characteristic. Parameters to the BT_GATT_CHARACTERISTIC macro indicate that the alert level characteristic supports read and write operations and callback functions for each of these two operations are specified. We then define an instance of bt_gatt_service with that array of attributes as an argument. We've completed our definition of the link loss service.

In the link_loss_init() function, we now need to register the service. Add the following call:

```
        bt_gatt_service_register(&link_loss_svc);
```

Checkpoint

Build your code and copy the zephyr.hex file onto your microbit. Use nRF Connect to connect to it and review the GATT services shown. You should see the link loss service with its alert level characteristic is now present.



Generic Attribute
UUID: 0x1801
PRIMARY SERVICE

Generic Access
UUID: 0x1800
PRIMARY SERVICE

Device Information
UUID: 0x180A
PRIMARY SERVICE

Link Loss
UUID: 0x1803
PRIMARY SERVICE

Alert Level
UUID: 0x2A06
Properties: READ, WRITE

Tx Power
UUID: 0x1804
PRIMARY SERVICE

Handle Alert Level reads and writes

Now we need to complete the implementation of the alert level read and write functions. Update the read_link_loss_alert_level function so that the return statement uses the bt_gatt_attr_read function as shown:

```
ssize_t read_link_loss_alert_level(struct bt_conn *conn,
                                       const struct bt_gatt_attr *attr, void *buf,

                                       u16_t len, u16_t offset)
{
        return bt_gatt_attr_read(conn, attr, buf, len, offset, &link_loss_alert_level,
sizeof(link_loss_alert_level));
}
```

bt_gatt_attr_read is a Zephyr API function which places the specified attribute value into a buffer and from there, transmits a response to a read request.

Now we'll complete the write_link_loss_alert_level function. We'll validate the received value and if it's valid, cause the appropriate LED to flash, per our requirements. Note that hardware/led.c already contains the code required to flash the LEDs. Update the write_link_loss_alert_level function so that it looks like this:

```
ssize_t write_link_loss_alert_level(struct bt_conn *conn,

                                             const struct bt_gatt_attr *attr,

                                             const void *buf, u16_t len, u16_t offset,

                                             u8_t flags)
{
        const u8_t *new_alert_level = buf;
        if (!len)
        {
                return BT_GATT_ERR(BT_ATT_ERR_INVALID_ATTRIBUTE_LEN);
        }
        if (*new_alert_level >= 0 && *new_alert_level <= 2)
        {
                link_loss_alert_level = *new_alert_level;
        }
        else
        {
                return BT_GATT_ERR(BT_ATT_ERR_VALUE_NOT_ALLOWED);
        }
        switch (link_loss_alert_level)
        {
        case 0:
                flash(true, false, false, 4);
                break;
        case 1:
                flash(false, true, false, 4);
                break;
        case 2:
                flash(false, false, true, 4);
                break;
        }
        return len;
}
```

Checkpoint

Build your code and copy the zephyr.hex file onto your microbit. Use nRF Connect to connect to it and try reading and writing to the link loss service's alert level characteristic. Check that the right LED flashes for each of the three valid values which may be written to the characteristic.

Requirement 3 – Immediate Alert, Alert Level

Requirement 3 states that the smartphone can send an immediate alert level value of 0, 1 or 2 and in response, the peripheral device will flash the LEDs in unison, either 3, 4 or 5 times, possibly accompanied by the buzzer. Check the requirement in the Profile Design document to be clear about the behaviour we need to implement.

This time, the Alert Level characteristic we're going to be writing to belongs to the Immediate Alert service rather than the Link Loss service.

So, we need to implement the GATT immediate alert service and handle writes to its alert level characteristic.

A partial implementation of the immediate alert service can be found in the src/services/immediate_alert.c and src/services/immediate_alert.h files. Open the .c file in your editor.

Define and Register the GATT Service

Under the comment "TODO Immediate Alert Service Declaration" add this code:

```
/* Immediate Alert Service Declaration */
static struct bt_gatt_attr attrs[] = {
                BT_GATT_PRIMARY_SERVICE(BT_UUID_IMMEDIATE_ALERT),
                BT_GATT_CHARACTERISTIC(BT_UUID_ALERT_LEVEL, BT_GATT_CHRC_WRITE_WITHOUT_RESP,

         BT_GATT_PERM_WRITE, NULL, write_immediate_alert_alert_level, NULL),
};

static struct bt_gatt_service immediate_alert_svc = BT_GATT_SERVICE(attrs);
```

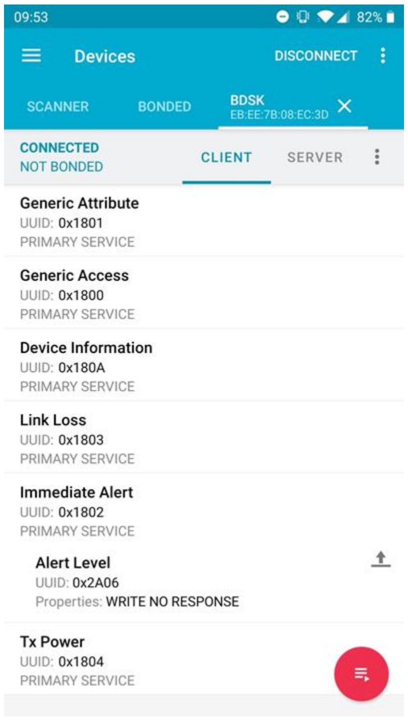
Once again, we're defining an array of GATT attributes, using Zephyr macros to include the GATT immediate alert service, whose UUID of 0x1802 was already defined in the skeleton code, followed by the alert level characteristic. Parameters to the BT_GATT_CHARACTERISTIC macro indicate that when owned by the immediate alert service, the alert level characteristic supports only the *write without response* operation and also specifies a callback function for the operation. We then define an instance of bt_gatt_service with that array of attributes as an argument. We've completed our definition of the immediate alert service.

In the immediate_alert_init () function, we now need to register the service. Add the following call:

```
void immediate_alert_init()
{
        // TODO register service
        bt_gatt_service_register(&immediate_alert_svc);
}
```

Checkpoint

Build your code and copy the zephyr.hex file onto your microbit. Use nRF Connect to connect to it and review the GATT services shown. You should see the immediate alert service with its alert level characteristic is now present.



Generic Attribute
UUID: 0x1801
PRIMARY SERVICE

Generic Access
UUID: 0x1800
PRIMARY SERVICE

Device Information
UUID: 0x180A
PRIMARY SERVICE

Link Loss
UUID: 0x1803
PRIMARY SERVICE

Immediate Alert
UUID: 0x1802
PRIMARY SERVICE

Alert Level
UUID: 0x2A06
Properties: WRITE NO RESPONSE

Tx Power
UUID: 0x1804
PRIMARY SERVICE

Now we need to complete the implementation of the alert level *write without response* functions.

We'll validate the received value and if it's valid, flash LEDs and sound the buzzer, per requirement #3. Note that hardware/led.c already contains the code required to flash the LEDs and buzzer control is implemented in hardware/buzzer.c.

Update the write_immediate_alert_alert_level function so that it looks like this:

```
ssize_t write_immediate_alert_alert_level(struct bt_conn *conn,const struct bt_gatt_attr
*attr,const void *buf, u16_t len, u16_t offset,u8_t flags)
{
        const u8_t *new_alert_level = buf;
        if (!len)
        {
                return BT_GATT_ERR(BT_ATT_ERR_INVALID_ATTRIBUTE_LEN);
        }
        if (*new_alert_level >= 0 && *new_alert_level <= 2)
        {
                immediate_alert_alert_level = *new_alert_level;
        }
        else
        {
                return BT_GATT_ERR(BT_ATT_ERR_VALUE_NOT_ALLOWED);
        }
        printk("write_immediate_alert_alert_level(%d)\n", immediate_alert_alert_level);
        switch (immediate_alert_alert_level)
        {
        case 0:
                flash(true, true, true, 3);
                break;
        case 1:
                flash(true, true, true, 4);
                buzz(4);
                break;
        case 2:
                flash(true, true, true, 5);
                buzz(5);
                break;
        }
        return len;
}
```

Checkpoint

Run your revised code and test it using either nRF Connect. Write the values 0, 1 and 2 in succession to the Alert Level characteristic of the Immediate Alert Service. The expected results are:

0 – All three LEDs flash a total of three times, the buzzer is silent

1 – All three LEDs flash a total of four times and the buzzer sounds four times in sync with the LEDs

2 – All three LEDs flash a total of five times and the buzzer sounds five times in sync with the LEDs

Requirement 4 – Proximity Monitoring

This requirement should produce the following result: "*The connected smartphone application will track its distance from the peripheral device using the received signal strength indicator (RSSI) and classify it as near (green), middle distance (yellow) or far away (red). The peripheral device must allow the smartphone to communicate its proximity classification and signal strength as measured at the smartphone. The LED of the colour corresponding to the proximity classification should be illuminated.*"

So in short, our smartphone app(s) will be measuring and classifying how far away they are from the peripheral device and want to be able to share that information back to it so it can use the information to light one of the LEDs. We introduced a custom service for this purpose called the Proximity Monitoring service which has a single characteristic called Client Monitoring.

Open the services/proximity_monitoring.c source file in your editor.

Define and Register the GATT Service

Under the comment "TODO Proximity Monitoring Service Declaration" add this code:

```
static struct bt_gatt_attr attrs[] = {
               BT_GATT_PRIMARY_SERVICE(BT_UUID_PROXIMITY_MONITORING),
               BT_GATT_CHARACTERISTIC(BT_UUID_CLIENT_PROXIMITY,
BT_GATT_CHRC_WRITE_WITHOUT_RESP,

        BT_GATT_PERM_WRITE, NULL, write_client_proximity, NULL),
};

static struct bt_gatt_service proximity_monitoring_svc = BT_GATT_SERVICE(attrs);
```

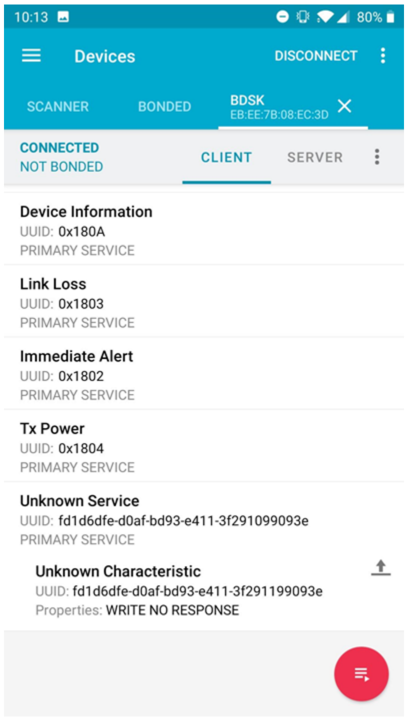
Our array of GATT attributes includes the GATT proximity monitoring service, for which the required 128-bit custom UUID was already defined in the skeleton code, followed by the client proximity characteristic. Parameters to the BT_GATT_CHARACTERISTIC macro indicate that the client proximity characteristic supports only the write request operation and a callback function for the operation is provided. We then define an instance of bt_gatt_service with that array of attributes as an argument. We've completed our definition of the proximity monitoring service.

In the proximity_monitoring_init () function, we now need to register the service. Add the following call:

```
void proximity_monitoring_init()
{
       bt_gatt_service_register(&proximity_monitoring_svc);
}
```

Checkpoint

Build your code and copy the zephyr.hex file onto your microbit. Use nRF Connect to connect to it and review the GATT services shown. You should see the custom proximity monitoring service with its client proximity characteristic is now present.



Device Information
UUID: 0x180A
PRIMARY SERVICE

Link Loss
UUID: 0x1803
PRIMARY SERVICE

Immediate Alert
UUID: 0x1802
PRIMARY SERVICE

Tx Power
UUID: 0x1804
PRIMARY SERVICE

Unknown Service
UUID: fd1d6dfe-d0af-bd93-e411-3f291099093e
PRIMARY SERVICE

Unknown Characteristic
UUID: fd1d6dfe-d0af-bd93-e411-3f291199093e
Properties: WRITE NO RESPONSE

Handle Client Proximity writes

Now we need to complete the implementation of the client proximity write function. Update the write_client_proximity function as shown:

```
ssize_t write_client_proximity(struct bt_conn *conn,

                        const struct bt_gatt_attr *attr,

                        const void *buf, u16_t len, u16_t offset,

                        u8_t flags)
{
        printk("write_client_proximity\n");
        const u8_t *new_proximity_data = buf;
        if (len != 2)
        {
                return BT_GATT_ERR(BT_ATT_ERR_INVALID_ATTRIBUTE_LEN);
        }
        proximity_band = new_proximity_data[0];
        client_rssi = (s8_t)new_proximity_data[1];
        printk("proximity_band: (%d)\n", proximity_band);
        printk("client-rssi: (%d)\n", client_rssi);
        char lcdbuf[16];
        switch (proximity_band)
        {
        case 0:
                set_led_states(false, false, false);
                clear_lcd();
                return len;
        case 1:
                set_led_states(true, false, false);
                break;
        case 2:
                set_led_states(false, true, false);
                break;
        case 3:
                set_led_states(false, false, true);
                break;
        }
        sprintf(lcdbuf, "Client RSSI: %d  ", client_rssi);
        write_string(lcdbuf, 16, 0, 0);
        return len;
}
```

Once again, we validate the length of the value received. if the *proximity _band* value, from the first octet of the two octet value is greater than 0x00, we switch on one of the coloured LEDs and then use the *write_string* function, which is defined in hardware/lcd/lcd.c to display a message on the connected serial LCD display. If *proximity_band* is 0x00 we clear the LCD and exit from the function.

Checkpoint

Run the modified code and test it using nRF Connect. Write values which have 0x00, 0x01, 0x02 and 0x03 in byte position 0 and a reasonable seeming value for the RSSI value in byte position 2 such as 0xC8. We're using the binary complement here so 0xC8 (decimal 200) should appear on the LCD display as a Client RSSI value of -56. Remember that 0x00 in byte position one means that the user of the client application has switched off proximity sharing and so in this special situation, we need to switch all the LEDs off.

Requirement 5 – Link Loss

The description for this requirement says: "*If the connection between client and peripheral device is lost then if the alert level set in feature (2) is greater than 0, the peripheral device circuit should make a noise and flash all LEDs for an extended period of time. The duration of the alert made should be 30 seconds if the alert level set in feature (2) is 2 or 15 seconds if it was set to 1.*"

Open main.c in your editor and find the *disconnected* function. Update it as shown:

```
static void disconnected(struct bt_conn *conn, u8_t reason)
{
        printk("Disconnected (reason %u)\n", reason);
        printk("Disconnected (alert_level %d)\n", get_link_loss_alert_level());
        write_string("disconnected    ", 16, 0, 0);
        set_led_states(false, false, false);
        stop_sampling_temperature();
        if (get_link_loss_alert_level() > 0)
        {
                // handle link loss per the alert level
                flash(true, true, true, get_link_loss_alert_level() * 15);
                buzz(get_link_loss_alert_level() * 15);
        }
        if (default_conn)
        {
                bt_conn_unref(default_conn);
                default_conn = NULL;
        }
}
```

Checkpoint

Build and install your code. Connect to your device and set the link loss alert level to 1. Disconnect and check that your device beeps and flashes as required. Repeat the test but set alert level to 2. On disconnecting, the beeping and flashing should last for twice as long as the previous test. Finally, test with alert level set to zero. There should be no flashing and beeping on disconnect.

Requirement 6 – Temperature Monitoring

The description for this requirement says: "*The peripheral device must be able to measure the ambient temperature once per second and to communicate measurements to the connected client. The client must be able to enable or disable this behaviour.*"

Our profile includes the Health Thermometer service with the Temperature Measurement characteristic to allow this requirement to be met and the Temperature Measurement characteristic supports indications.

Open the services/health_thermometer.c source file in your editor.

Define and Register the GATT Service

Under the comment “TODO Health Thermometer Service Declaration” add this code:

```
static struct bt_gatt_attr attrs[] = {
                BT_GATT_PRIMARY_SERVICE(BT_UUID_HEALTH_THERMOMETER),
                BT_GATT_CHARACTERISTIC(BT_UUID_TEMPERATURE_MEASUREMENT, BT_GATT_CHRC_INDICATE,

         BT_GATT_PERM_NONE, NULL, NULL, NULL),
                BT_GATT_CCC(tm_ccc_cfg, tm_ccc_cfg_changed),
};

static struct bt_gatt_service health_thermometer_svc = BT_GATT_SERVICE(attrs);
```

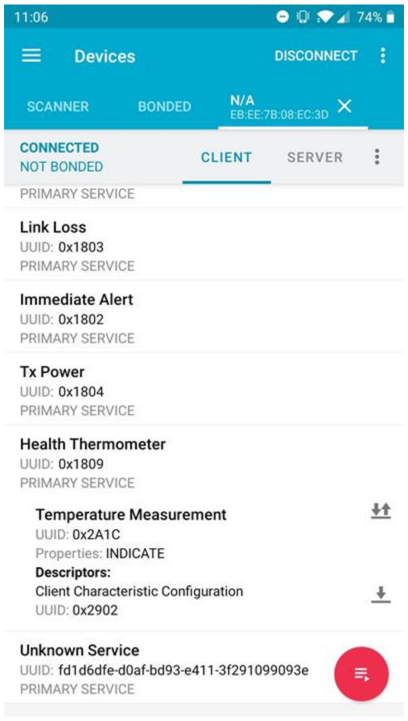
Our array of GATT attributes includes the GATT health thermometer service, for which the UUID 0x1809 was already defined in the skeleton code, followed by the temperature measurement characteristic. Parameters to the BT_GATT_CHARACTERISTIC macro indicate that the temperature measurement characteristic supports only the *indicate* request operation. Using the BT_GAT_CCC macro, we’ve also included the client characteristic configuration descriptor, which is required for the purpose of enabling or disabling indications from the temperature measurement characteristic. It’s declaration includes a callback function (tm_ccc_cfg_changed) which is called whenever the value of this descriptor gets changed by the connected client. We then define an instance of bt_gatt_service with that array of attributes as an argument. We’ve completed our definition of the health thermometer service.

In the health_thermometer_init () function, we now need to register the service. Add the following call:

```
void health_thermometer_init()
{
        flags = 0x00;
        bt_gatt_service_register(&health_thermometer_svc);
}
```

Checkpoint

Build your code and copy the zephyr.hex file onto your microbit. Use nRF Connect to connect to it and review the GATT services shown. You should see the health thermometer service with its temperature measurement characteristic is now present.


11:06 74%

Devices DISCONNECT

SCANNER BONDED N/A EB:EE:7B:08:EC:3D

CONNECTED
NOT BONDED

CLIENT SERVER

PRIMARY SERVICE

Link Loss
UUID: 0x1803
PRIMARY SERVICE

Immediate Alert
UUID: 0x1802
PRIMARY SERVICE

Tx Power
UUID: 0x1804
PRIMARY SERVICE

Health Thermometer
UUID: 0x1809
PRIMARY SERVICE

Temperature Measurement
UUID: 0x2A1C
Properties: INDICATE
Descriptors:
Client Characteristic Configuration
UUID: 0x2902

Unknown Service
UUID: fd1d6dfe-d0af-bd93-e411-3f291099093e
PRIMARY SERVICE

Temperature Measurement Indications

Our final task is to complete a function, called by the sample_temperature() function in main.c, which will send temperature measurement indications. The function we need to complete is in health_thermometer.c and is called indicate_temperature_measurement_celsius. It takes an argument of the temperature in celsius. Update the function so that it looks like this:

```
void indicate_temperature_measurement_celsius(int celsius)
{

        celsius_times_10 = celsius * 10;
        // all our values are to one decimal place so we're sending the mantissa as 10 x
celsius value with an exponent of -1
        // temperature_measurement[0] is the FLAGS field and already contains the required
value of 0x00

        // little endian
        temperature_measurement[4] = 0xFF; // exponent of -1
        temperature_measurement[3] = (celsius_times_10 >> 16);
        temperature_measurement[2] = (celsius_times_10 >> 8) & 0xFF;
        temperature_measurement[1] = celsius_times_10 & 0xFF;

        indicate_params.attr = &attrs[2];
        indicate_params.data = &temperature_measurement;
        indicate_params.len = 5;
        indicate_params.func = indicate_cb;
        bt_gatt_indicate(the_conn, &indicate_params);
}
```

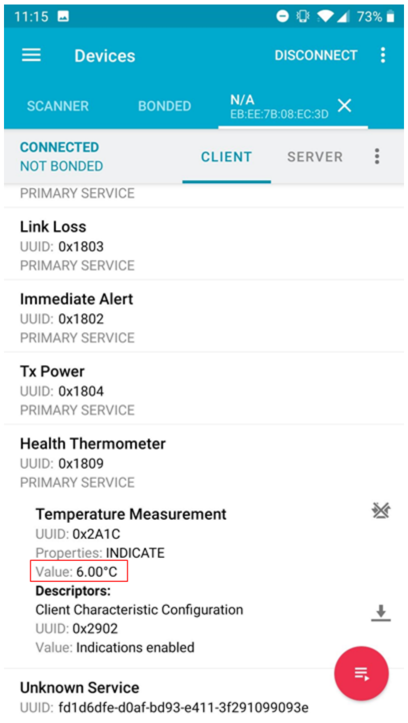
This function is more complicated than you perhaps expected. The health thermometer service specification explains the required format of the temperature measurement characteristic, whose value this function sets. You can obtain that specification from https://www.bluetooth.com/specifications/gatt/services

Having formulated the value, we use the Zephyr function bt_gatt_indicate to send the indication, with parameters which include the value of the characteristic in a struct.

Checkpoint

Test your modified code. Connect to the device using nRF Connect. Subscribe to indications on the Temperature Measurement characteristic and watch values arrive at the client. Unsubscribe and ensure that temperature sampling stops. Disconnect.

And here's a screenshot from nRF Connect showing a temperature measurement value:



PRIMARY SERVICE

Link Loss
UUID: 0x1803
PRIMARY SERVICE

Immediate Alert
UUID: 0x1802
PRIMARY SERVICE

Tx Power
UUID: 0x1804
PRIMARY SERVICE

Health Thermometer
UUID: 0x1809
PRIMARY SERVICE

Temperature Measurement
UUID: 0x2A1C
Properties: INDICATE
Value: 6.00°C
Descriptors:
Client Characteristic Configuration
UUID: 0x2902
Value: Indications enabled

Unknown Service
UUID: fd1d6dfe-d0af-bd93-e411-3f291099093e

Temperature data readings are currently simulated and you should see the celsius value rise slowly before decreasing. This cycle of increasing and then decreasing will repeat until you unsubscribe from indications or disconnect.

Testing - Videos

The Servers\Videos folder contains a video, 'testing_with_nrf_connect.mp4'. It involves an Arduino for most of the tests but the behaviours under test are the same. Review your results against the behaviours shown in the video.

Conclusion

That's it! You've designed, coded and tested a custom Bluetooth profile for an Zephyr on a BBC micro:bit. Nice work!

Next, choose one of the smartphone labs and have a go at creating an application which will work with your micro:bit.

Additional Optional Exercise

As noted, temperature measurements in the skeleton starter code are simulated. You may want to replace this with a real sensor and use the Zephyr Analog to Digital Converter (ADC) APIs to read from it periodically, instead.

We used a tmp36 sensor with the sensor pin (the central pjn) connected to micro:bit **pin 1**.

Add ADC support

To include ADC support we need to add the following to the Zephyr overlay file, bbc_microbit.overlay:

```
&uart0 {
   status = "ok";
   compatible = "nordic,nrf-uart";
   current-speed = <9600>;
   tx-pin = <3>;
   rx-pin = <16>;
};
&adc {
       status ="ok";
};
```

And we need to add the following to prj.conf

```
CONFIG_ADC=y
CONFIG_ADC_NRFX_ADC=y
CONFIG_ADC_ASYNC=n
CONFIG_ADC_0=y
CONFIG_ADC_NRFX_ADC_CHANNEL_COUNT=3
```

Now, open hardware/thermistor.h and adfunction declaration:

```
u16_t get_sensor_reading();
```

Next, open hardware/thermistor.c. Under the #include statements, add the following:

```
#include <adc.h>
#include <hal/nrf_adc.h>

#define ADC_DEVICE_NAME DT_ADC_0_NAME
#define ADC_RESOLUTION 10
#define ADC_GAIN ADC_GAIN_1_3
#define ADC_REFERENCE ADC_REF_INTERNAL
#define ADC_ACQUISITION_TIME ADC_ACQ_TIME_DEFAULT
#define ADC_1ST_CHANNEL_ID 0
#define ADC_1ST_CHANNEL_INPUT NRF_ADC_CONFIG_INPUT_3

#define BUFFER_SIZE 1
static u16_t m_sample_buffer[BUFFER_SIZE];

struct device *adc_dev;
static const struct adc_channel_cfg m_1st_channel_cfg = {
               .gain = ADC_GAIN,
               .reference = ADC_REFERENCE,
               .acquisition_time = ADC_ACQUISITION_TIME,
               .channel_id = ADC_1ST_CHANNEL_ID,
               .input_positive = ADC_1ST_CHANNEL_INPUT,
};
```

This code supplies various parameters for configuring the ADC and a buffer that data will be read into. Note that NRF_ADC_CONFIG_INPUT_3 maps to pin 1 on a micro:bit.

Add a function which initialises the ADC and call it from the thermistor_init() function:

```
static struct device *setup_adc(void)
{
        int ret;
        struct device *the_adc_dev = device_get_binding(ADC_DEVICE_NAME);
        ret = adc_channel_setup(the_adc_dev, &m_1st_channel_cfg);
        (void)memset(m_sample_buffer, 0, sizeof(m_sample_buffer));
        return the_adc_dev;
}
```

```
void thermistor_init()
{
        sim_adc = 151;
        sim_delta = 1;
        // ADC
        adc_dev = setup_adc();
}
```

Add a new function which will read a value from the ADC and calculate a celsius temperature from it:

```
u16_t get_sensor_reading()
{
        int ret;
        static struct adc_sequence_options options = {
                        .callback = NULL,
        };
        const struct adc_sequence sequence = {
                        .options = &options,
                        .channels = BIT(ADC_1ST_CHANNEL_ID),
                        .buffer = m_sample_buffer,
                        .buffer_size = sizeof(m_sample_buffer),
                        .resolution = ADC_RESOLUTION,
        };
        if (!adc_dev)
        {
                printk("ERROR: no ADC device!!\n");
                return -1;
        }
        ret = adc_read(adc_dev, &sequence);
        if (ret == 0)
        {
                return m_sample_buffer[0];
        }
        printk("ERROR reading from ADC %d\n", ret);
        return 0;
}
```

Finally, modify the readTemperature() function so that it calls the new sensor reading function instead of the function which returns a simulated temperature reading:

```
float readTemperature()
{
        // u16_t adc_reading = simulate_sensor_reading();
        u16_t adc_reading = get_sensor_reading();

        // reference voltage is 1.2V and 3 x 1.2 is close enough to the micro:bit 3.3V. We used a gain (pre-scaling) of one third so we scale back up here.
        int mv = (int)adc_reading * 3600 / 1023;

        // 10 mV/°C scale factor
        // Spec for TMP36 says 750mv = 25 degrees celsius. So conversion to celsius from mV is mV / 10 - 50
        float celsius = (mv / 10) - 50;
        return celsius;
}
```

That's it. Build, install and test your new code. You should now see real temperature readings delivered as Bluetooth indication values to the temperature measurement characteristic when you connect to your device and enable indications.