Developer Study Guide: An introduction to Bluetooth Low Energy Development

Creating a Bluetooth Low Energy peripheral using Raspberry Pi

Version: 5.0.3

Last updated: 17th December 2018

Contents

Revision History

Version	Date	Author	Changes
5.0.0	13th December 2017	Martin Woolley, Bluetooth SIG	First version with Raspberry Pi lab
5.0.2	15th June 2018	Martin Woolley, Bluetooth SIG	Removed use of Bluetooth Developer Studio
5.0.3	17th December 2018	Martin Woolley, Bluetooth SIG	Changed name to "Developer Study Guide: An introduction to Bluetooth Low Energy Development"

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 a Raspberry Pi and a simple electronic circuit.

Goal

This document guides you through hardware assembly, Bluetooth profile design, coding and testing. You'll be turning your Raspberry Pi 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 lab should not be considered a complete solution, nor the basis for any commercial product.

We'll also make the Raspberry Pi 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 Raspberry Pi device will include LEDs and these will be used to indicate alarm conditions and whether the Raspberry Pi is near or far from the connected client as well, so both devices will be able to keep an eye on how far they are from the other.

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 Raspberry Pi
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**.

Raspberry Pi Exercise 1: Setting up

Hardware

Your Raspberry Pi ("Pi") must support Bluetooth Low Energy (LE). A Pi 3 has LE support built in. If you're using an older Pi, you need to plug in a suitable dongle or other LE adapter. We won't be covering the steps needed to get external adapters or the Linux BlueZ stack working and on the whole you should find that it works without problem. There are numerous guides on the internet which will help you get Bluetooth working on an older Pi, should you need assistance. We assume in the remainder of this lab that your Bluetooth stack is working.

We designed a simple breadboard circuit for our lab exercises. It's depicted below. As you can see, it consists of an Raspberry Pi, a piezo buzzer, a thermistor 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.

The circuit diagram was drawn using an open source, free software tool called Fritzing by the way.

Parts List

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

Raspberry Pi 3

1 x red LED

1 x yellow LED

1 x green LED

3 x 180 Ohms 5% tolerance resistors

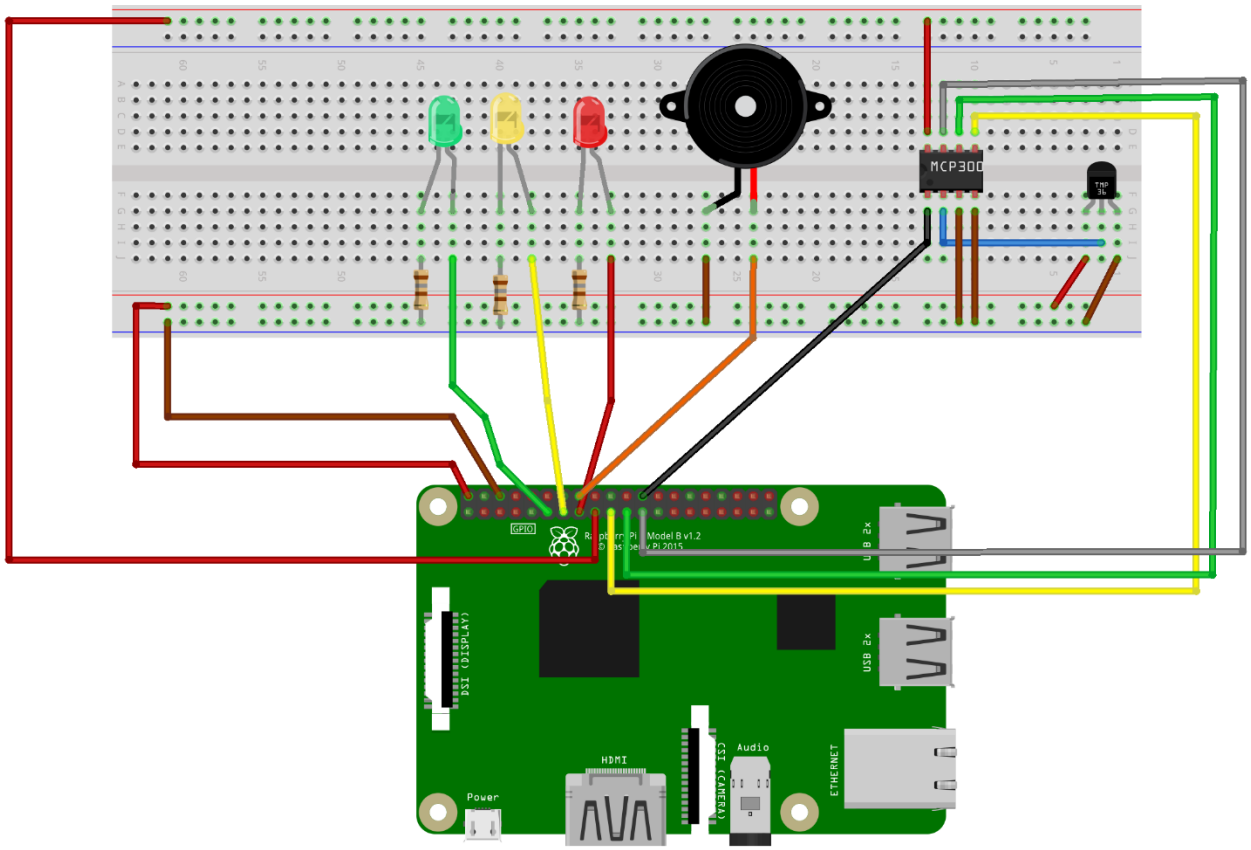
1 x kingstate kPEG200A buzzer

1 x breadboard

1 x TMP36 temperature sensor or similar (optional)

1 x MCP2002 analogue to digital converter (optional)

Breadboard wires for forming connections


MCP300
GPIO
Model B v1.2
USB 2x
USB 2x
DSI (DISPLAY)
HDMI
CSI (CAMERA)
Audio
ETHERNET
Power

fritzing

Raspberry Pi PIN Connections

Breadboard	Raspberry Pi	Note
Ground bus	Physical pin 6	
+ve bus (5v)	Physical pin 2	5v
Green LED +ve	Physical pin 11	
Yellow LED +ve	Physical pin 13	
Red LED +ve	Physical pin 15	
All LED -ve pins connected to breadboard ground bus via 180ohm resistors		
Buzzer +ve	Physical pin 12	
Buzzer -ve to breadboard ground bus		
Second breadboard power bus (3.3v)	Physical pin 17	3.3v
TMP36 temperature sensor right pin (ground) to breadboard ground bus		
TMP36 sensor centre pin (data) to MCP3002 pin 2 (CH0)		
TMP36 sensor left pin (power) to breadboard 5v power bus		optional component
MCP3002 pin 1 (CS)	Physical pin 24	optional component
MCP3002 pin 2 (CH0) to TMP36 sensor centre pin (data)		optional component
MCP3002 pin 3 (CH1) to breadboard ground bus		CH1 is not used optional component
MCP3002 pin 4 (VSS) to breadboard ground bus		optional component
MCP3002 pin 5 (DIN)	Physical pin 19 (MOSI)	optional component

MCP3002 pin 6 (DOUT)	Physical pin 21 (MISO)	optional component
MCP3002 pin 7 (CLK)	Physical pin 23 (SCLK)	optional component
MCP3002 pin 8 (VDD) to breadboard 3.3v power bus		optional component

See https://pinout.xyz/pinout/pin19_gpio10 for the Raspberry Pi pin layout and numbering scheme.

Note: Connections for your temperature sensor may vary. See https://www.sparkfun.com/products/10988 for information on the TMP36.

Note: for temperature monitoring, you have a choice of using real temperature data acquired from a sensor and processed via the ADC or you can generate simulated temperature readings in the code you're going to write for the Raspberry Pi. The choice is yours.

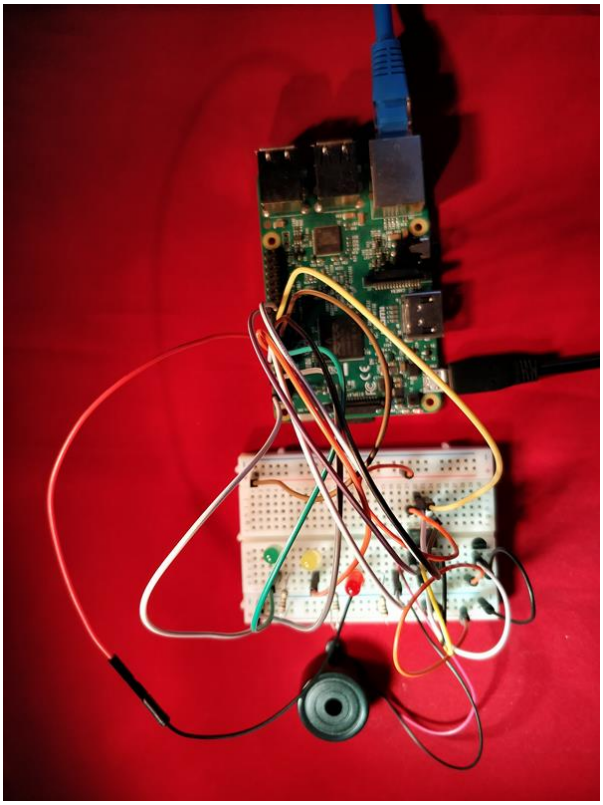


Figure 1 - A Raspberry Pi 3 connected to the breadboard circuit

Hardware Setup and Construction

Your first task is to build the breadboard circuit and connect it as specified to the Raspberry Pi pins.

Raspberry Pi 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 suitable software framework. 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.

Raspberry Pi Programming Language and APIs

For this lab, we'll be programming in node.js and use several modules to provide us with APIs for Bluetooth LE, switching GPIO pins on and off and for interfacing with the MCP3002 ADC.

Your first job therefore is to install node.js on your Raspberry Pi, assuming it's not already there. We used version 8.9.3 of node.js. You can check the version on your machine by running the command "node -v".

Now, let's create a folder to host the code you're going to write and install the required node.js modules. From a shell, execute the following steps:

```
pi@raspberrypi:~/projects $ mkdir bdsk
pi@raspberrypi:~/projects $ cd bdsk
pi@raspberrypi:~/projects/bdsk $ npm install bleno
.......
+ bleno@0.4.2
```

We now have a folder to work in and we've installed the bleno module, which provides APIs for Bluetooth GAP Peripherals.

Next, install the node.js onoff package, which provides us with a GPIO API.

```
npm install onoff
.......
+ onoff@1.1.8
```

Finally, install the node.js mcp-spi-adc package, which provides us with an API to use with the MCP3002 ADC, from which temperature sensor data will be read.

```
npm install mcp-spi-adc
.......
+ mcp-spi-adc@1.0.0
```

API documentation for the three APIs can be found at the following URLs:

https://github.com/sandeepmistry/bleno

https://github.com/fivdi/onoff

https://github.com/fivdi/mcp-spi-adc

You're all set to start coding!

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. You will use a smartphone application called nRF Connect, which is available for both Android and iOS or similar.

GAP Advertising

We'll start by making our Pi perform Bluetooth advertising, accept connections and host the profile we designed, as a skeleton, with no associated behaviours. At the end of the next few steps, your Raspberry Pi will advertise with the required GAP advertising packet content and when you connect to it, you'll see that it has the required GATT services, characteristics and descriptors. But performing operations such as writing to a characteristic, will have no discernible effect, at least as far as the connected circuit is concerned. The latter falls to bespoke application code by which we'll complete the requirements we specified.

Create a text file called bdsk.js and using your favourite text editor, add the following to it:

```
var bleno = require('bleno');

bleno.on('stateChange', function (state) {
        console.log('on -> stateChange: ' + state);

        if (state === 'poweredOn') {
                bleno.startAdvertising('BDSK', ['1803']);
        } else {
                bleno.stopAdvertising();
        }
});
```

We've included the node.js bleno module and added an event handler to respond to changes in Bluetooth adapter state. When powered on, we start advertising with a local name of "BDSK" and with the Link Loss service UUID in the 16-bit services list.

Checkpoint

Run your code like this:

```
pi@raspberrypi:~/projects/bdsk $ sudo node bdsk.js
```

```
on -> stateChange: poweredOn
```

Note that the bleno module requires you to run with "sudo".

It should execute with no errors, display the stateChange message shown in the console and be advertising with a name of "BDSK" and with service 1803. You can verify this by installing any of a number of available Bluetooth "GATT explorer" applications on your smartphone. nRF Connect is one such example. Install it or your preferred application on your phone, launch it, find your Raspberry Pi, which should be advertising with the name "BDSK". Look at the advertising data and compare it with the following screenshot:

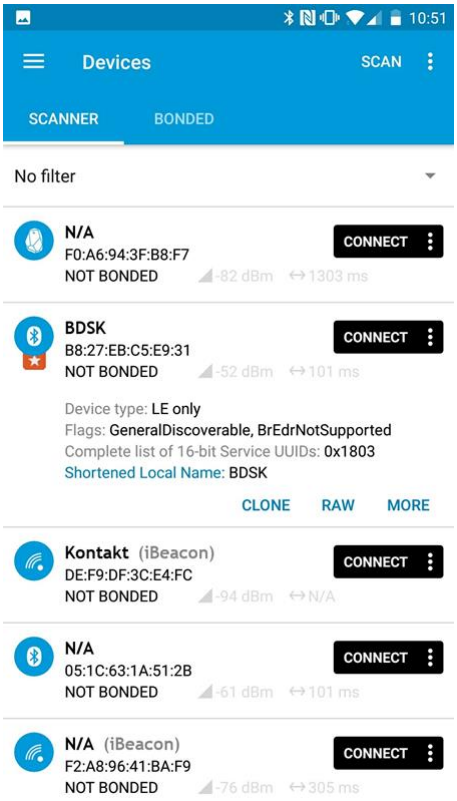


Figure 2 - BDSK Advertising Packet Details

It will be assumed that you're using nRF Connect in the remainder of this text. Substitute the name of your chosen tool if you're using something else.

GATT Services, Characteristics and Descriptors

Now we'll add the services and their characteristics and descriptors. We'll specify their UUIDs and properties, per the profile design but do little more than include some console logging for the supported operations such as read and write.

Add the following code and then review it carefully to make sure you understand it.

```
bleno.on('advertisingStart', function (error) {
	console.log('on -> advertisingStart: ' + (error ? 'error ' + error : 'success'));

	if (!error) {
		bleno.setServices([
			//  link loss service
			new bleno.PrimaryService({
				uuid: '1803',
				characteristics: [
					// Alert Level
					new bleno.Characteristic({
					    value: 0,
					    uuid: '2A06',
					    properties: ['read', 'write'],
					     onReadRequest: function (offset, callback) {
					         console.log('link loss.alert level READ:');
					         callback(this.RESULT_SUCCESS, octets);
					     },
					     onWriteRequest: function (data, offset,
withoutResponse, callback) {
					       console.log("link loss.alert level WRITE "+data[0])
					        callback(this.RESULT_SUCCESS);
					     }
					})
				]
			}),
			//  immediate alert service
			new bleno.PrimaryService({
				uuid: '1802',
				characteristics: [
					// Alert Level
					new bleno.Characteristic({
					        value: 0,
					        uuid: '2A06',
					        properties: ['writeWithoutResponse'],
					        onWriteRequest: function (data, offset,
withoutResponse, callback) {
```

```
                        console.log("immediate alert.alert level WRITE "+data[0]);
                                        callback(this.RESULT_SUCCESS);
                                }
                        })
                ]
        }),
        //  TX power service
        new bleno.PrimaryService({
                uuid: '1804',
                characteristics: [
                        // Power Level
                        new bleno.Characteristic({
                                value: 0,
                                uuid: '2A07',
                                properties: ['read'],
                                onReadRequest: function (offset, callback) {
                                        console.log('TX Power.level read
request:');
                                        data = [0x0A];
                                        var octets = new Uint8Array(data);
                                        callback(this.RESULT_SUCCESS, octets);
                                }
                        })
                ]
        }),
        //  proximity monitoring service
        new bleno.PrimaryService({
                uuid: '3E099910293F11E493BDAFD0FE6D1DFD',
                characteristics: [
                        // client proximity
                        new bleno.Characteristic({
                                value: 0,
                                uuid: '3E099911293F11E493BDAFD0FE6D1DFD',
                                properties: ['writeWithoutResponse'],
                                onWriteRequest: function (data, offset,
withoutResponse, callback) {
                        console.log("proximity monitoring.client proximity WRITE
"+data[0]);
                                        callback(this.RESULT_SUCCESS);
                                }
                        })
                ]
        }),
        //  health thermometer service
```

```
                        new bleno.PrimaryService({
                                uuid: '1809',
                                characteristics: [
                                        // temperature measurement
                                        new bleno.Characteristic({
                                                value: 0,
                                                uuid: '2A1C',
                                                properties: ['indicate'],
                                        })
                                ]
                        }),
                ]);
        }
});

bleno.on('servicesSet', function (error) {
        console.log('on -> servicesSet: ' + (error ? 'error ' + error : 'success'));
});
```

As you can see, we've added two event handlers, one to handle the 'advertisingStart' event and the other for the 'servicesSet' event. Node.js is very event oriented and asynchronous, so if it's a new language for you, you'll soon get accustomed to this style of programming.

'advertisingStart' is an event which is generated when we call bleno.startAdvertising and 'servicesSet' occurs after bleno.setServices has completed.

In the advertisingStart event handler, we construct our GATT profile as a collection of services, characteristics and descriptors, with associated but as yet incomplete handlers for events like read or write requests. Compare the code with the profile design and requirements which you studied in the separate Profile Design document.

It's useful to monitor events such as a client connecting to our peripheral or disconnecting from it, so add the following code, which will log a message and the device address of the client as it connects or disconnects.

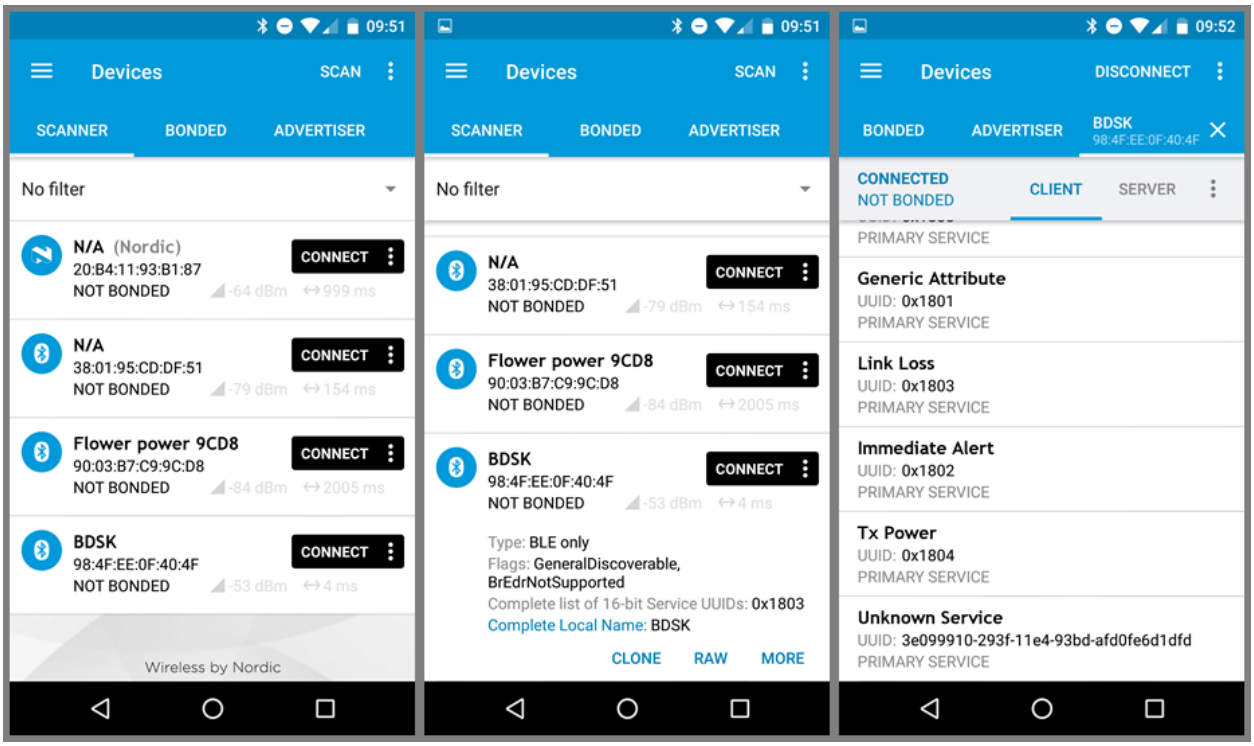
```
bleno.on('accept', function (clientAddress) {
        console.log('on -> accept, client: ' + clientAddress);
});

bleno.on('disconnect', function (clientAddress) {
        console.log("Disconnected from address: " + clientAddress);
});
```

Checkpoint

Run your code and then using nRF Connect, discover the BDSK device and this time, connect to it. You should see messages similar to the following ones on the Raspberry Pi console:

```
on -> stateChange: poweredOn
on -> advertisingStart: success
on -> servicesSet: success
on -> accept, client: 77:50:ce:47:20:89
```

nRF Connect will show you the GATT services, characteristics and descriptors that it discovered. Explore with nRF Connect and make sure what you see is what you expected.



The screenshot on the left shows our Arduino amongst other devices which have been discovered. Touching that entry causes it to expand and to reveal the advertising packet details, which are as we expect them to be. Connecting to the device shows us the list of services which includes those from the Proximity Profile, the Health Thermometer service plus one “unknown service” which is our custom, Proximity Monitoring service.

Note: The Health Thermometer Service is hidden off the bottom of the screen in the screenshots above. Make sure you scroll down and check that it is there.

Application Layer Code

Before we start to deal with our lab requirements, we'll get started by adding some useful functions which we'll use when we implement those requirements.

Add the following code near to the top of your source code:

```
var bleno = require('bleno');
var mcpadc = require('mcp-spi-adc');
var Gpio = require('onoff').Gpio;

var led1 = new Gpio(17, 'out');
var led2 = new Gpio(27, 'out');
var led3 = new Gpio(22, 'out');
var buzzer = new Gpio(18, 'out');

var link_loss_alert_level = 0;
var immediate_alert_level = 0;

var temperature_timer;
var celsius = 0.00;

var flash_count = 0;
var flash_state = 1;
var beep_count = 0;
var beep_state = 1;
var alert_timer;
var flashing = 0;
var beeping = 0;
var active_leds = [];
```

We've included the two other node.js modules which we need and a number of variables, whose use will become clear as we progress. The Gpio class comes from the onoff module and as you can see, we've created one for each of our LEDs and for the buzzer. Note that the onoff API uses the BCM pin numbering scheme, not the physical pin numbering used in our circuit diagram above. See https://pinout.xyz/pinout/pin19_gpio10 for the various pin numbering schemes the Pi uses.

Now add the following functions, which will let us control the LED lights and buzzer in various ways.

```
function startFlashing(led, interval, times) {
        flash_count = times * 2;
        active_leds.push(led);
        alert_timer = setInterval(flash_leds, interval);
        flashing = 1;
```

```
}

function flash_leds() {
        for (var i = 0; i < active_leds.length; i++) {
                active_leds[i].writeSync(flash_state);
        }
        if (flash_state == 1) {
                flash_state = 0;
        } else {
                flash_state = 1;
        }
        flash_count--;
        if (flash_count == 0 && beeping == 0) {
                clearInterval(alert_timer);
                active_leds = [];
        }
}
function allLedsOff() {
    led1.writeSync(0);
    led2.writeSync(0);
    led3.writeSync(0);
}
function beepOff() {
    beeping = 0;
    buzzer.writeSync(0);
}
function beep() {
        buzzer.writeSync(beep_state);
        if (beep_state == 1) {
                beep_state = 0;
        } else {
                beep_state = 1;
        }
        beep_count--;
        if (beep_count == 0) {
                beep_state = 0;
                buzzer.writeSync(beep_state);
                beeping = 0;
                clearInterval(alert_timer);
                active_leds = [];
        }
}
```

```
function startBeepingAndflashingAll(interval, alert_level) {
	if (alert_level > 0) {
		beep_count = flash_count;
		beeping = 1;
	} else {
		beeping = 0;
	}
	active_leds.push(led1);
	active_leds.push(led2);
	active_leds.push(led3);
	flashing = 1;
	alert_timer = setInterval(beepAndFlashAll, interval);
}
function beepAndFlashAll() {
	flash_leds();
	if (beeping == 1) {
		beep();
	}
}
```

We'll make use of a handy utility functions for logging hex values. Add the following function:

```
function u8AToHexString(u8a) {
	if (u8a == null) {
		return '';
	}
	hex = '';
	for (var i = 0; i < u8a.length; i++) {
		hex_pair = ('0' + u8a[i].toString(16));
		if (hex_pair.length == 3) {
			hex_pair = hex_pair.substring(1, 3);
		}
		hex = hex + hex_pair;
	}
	return hex.toUpperCase();
}
```

Finally, update the accept event handler to initialise some variables and switch all LEDs off and stop the buzzer when a client connects:

```
bleno.on('accept', function (clientAddress) {
```

```
        console.log('on -> accept, client: ' + clientAddress);
        flash_count = 0;
        flashing = 0;
        clearInterval(alert_timer);
        beepOff();
        allLedsOff();
});
```

Checkpoint

Run your code just to check you haven't introduced any errors when editing. It should start with no errors and log to the console like this:

```
pi@raspberrypi:~/projects/bdsk $ sudo node bdsk.js
on -> stateChange: poweredOn
on -> advertisingStart: success
on -> servicesSet: success
```

Requirement 2 – Link Loss, Alert Level

Next, we implement requirement 2, which you will recall involves handling writes to the Link Loss Service's Alert Level characteristic by storing the value provided and flashing one of the LEDs, according to the value received. We'll also make sure our device can respond to read requests on this characteristic as this is something which client applications are likely to want to do so that they can initialise their user interfaces correctly.

Add the following function, which will return the appropriate Gpio object for the LED we need to flash in response to a given alert level:

```
function getLed(level) {
        switch (level) {
                case 0: return led1;
                case 1: return led2;
                case 2: return led3;
                default: return led1;
        }
}
```

Add the highlighted code to react to a client reading or writing to the Alert Level characteristic of the Link Loss service as shown.

```
bleno.setServices([
    //  link loss service
```

```
    new bleno.PrimaryService({
        uuid: '1803',
        characteristics: [
            // Alert Level
            new bleno.Characteristic({
                value: 0,
                uuid: '2A06',
                properties: ['read', 'write'],
                onReadRequest: function (offset, callback) {
                    console.log('link loss.alert level READ:');
                    data = [ 0x00 ];
                    data[0] = link_loss_alert_level;
                    var octets = new Uint8Array(data);
                    console.log(octets);
                    callback(this.RESULT_SUCCESS, octets);
                },
                onWriteRequest: function (data, offset, withoutResponse, callback) {
                    console.log("link loss.alert level WRITE:");
                    this.value = data;
                    var octets = new Uint8Array(data);
                    console.log("0x"+u8AToHexString(octets).toUpperCase());
                    // application logic for handling WRITE or WRITE_WITHOUT_RESPONSE on
characteristic Link Loss Alert Level goes here
                    link_loss_alert_level = octets[0];
                    var led = getLed(link_loss_alert_level);
                    allLedsOff();
                    startFlashing(led, 250, 4);
                    callback(this.RESULT_SUCCESS);
                }
            })
        ]
    })
```

The new code handles write requests to the link loss service's alert level characteristic. We log the value, select the LED Gpio object for the LED we want to flash, using the written alert level value to determine this and then call the startFlashing function which uses the standard JavaScript setInterval function to set up a specified number of periodic calls to the flash_leds function. In short, we make the green LED flash if alert_level is 0, the yellow LED flash if it is 1 and the red LED if it is 2. Review those LED related functions again, from earlier in this lab to refresh your memory as to what they do.

Checkpoint

Run your revised code on the Raspberry Pi and test it now using the nRF Connect application. Scan and find the BDSK device and connect to it. Expand the Link Loss service entry in nRF Connect and select the

up facing arrow icon next to the Alert Level characteristic. This should bring up a dialogue and allow us to select one of three values to write to the characteristic. Select and send each value in turn, watching the circuit board as you do. Sending a High Alert level of 0x02 should result in the red LED flashing 4 times. Sending 0x01 will do the same with the yellow LED and sending 0x00 should flash the green LED. This is a visual acknowledgement that our write was processed and the new link loss alert level has been set for future use. Disconnect from the Raspberry Pi.

During your tests, the Raspberry Pi console should have displayed messages similar to these:

```
pi@raspberrypi:~/projects/bdsk $ sudo node bdsk.js
on -> stateChange: poweredOn
on -> advertisingStart: success
on -> servicesSet: success
on -> accept, client: 43:f4:4a:3b:ce:4b
link loss.alert level WRITE:
0x02
link loss.alert level WRITE:
0x01
link loss.alert level WRITE:
0x00
Disconnected from address: 43:f4:4a:3b:ce:4b
```

Requirement 3 – Immediate Alert, Alert Level

Requirement 3 states that the smartphone can send the Raspberry Pi an immediate alert level value of 0, 1 or 2 and in response, the Raspberry Pi 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.

Add the highlighted code to the onWriteRequest function attached to your immediate alert service's alert level characteristic. We're handling the write to the immediate alert service's alert level characteristic and initiating LED flashing and possibly, beeping of the buzzer, depending on the alert level value.

```
//  immediate alert service
new bleno.PrimaryService({
    uuid: '1802',
    characteristics: [
        // Alert Level
        new bleno.Characteristic({
            value: 0,
            uuid: '2A06',
            properties: ['writeWithoutResponse'],
```

```
            onWriteRequest: function (data, offset, withoutResponse, callback) {
                console.log("immediate alert.alert level WRITE:");
                this.value = data;
                var octets = new Uint8Array(data);
                console.log("0x"+u8AToHexString(octets).toUpperCase());
                immediate_alert_level = octets[0];
                flash_count = (immediate_alert_level + 3) * 2;
                startBeepingAndflashingAll(250, immediate_alert_level);
                callback(this.RESULT_SUCCESS);
            }
        })
    ]
}),
```

Have a look at the startBeepingAndFlashingAll function to see how it initiates periodic calls to a function beepAndFlashAll, which flashes all LEDs once and if alert level > 1, beeps the buzzer once, every time it is called.

Checkpoint

Run your revised code and test it using either nRF Connect on a smartphone or Bluetooth Developer Studio. 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

Disconnect when you have completed your testing. The Pi console should look something like this:

```
pi@raspberrypi:~/projects/bdsk $ sudo node bdsk.js
on -> stateChange: poweredOn
on -> advertisingStart: success
on -> servicesSet: success
on -> accept, client: 5c:18:23:97:79:00
immediate alert.alert level WRITE:
0x00
immediate alert.alert level WRITE:
0x01
immediate alert.alert level WRITE:
0x02
Disconnected from address: 5c:18:23:97:79:00
```

Requirement 4 – Proximity Monitoring

This requirement should produce the following result: "*The connected smartphone application will track its distance from the Raspberry Pi using the received signal strength indicator (RSSI) and classify it as near (green), middle distance (yellow) or far away (red). The Raspberry Pi must allow the smartphone to communicate its proximity classification and signal strength as measured at the smartphone. The Raspberry Pi's 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 Raspberry Pi and want to be able to share that information back to the Raspberry Pi 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.

Add the following, highlighted code:

```
//  proximity monitoring service
new bleno.PrimaryService({
    uuid: '3E099910293F11E493BDAFD0FE6D1DFD',
    characteristics: [
        // client proximity
        new bleno.Characteristic({
            value: 0,
            uuid: '3E099911293F11E493BDAFD0FE6D1DFD',
            properties: ['writeWithoutResponse'],
            onWriteRequest: function (data, offset, withoutResponse, callback) {
                console.log("proximity monitoring.client proximity WRITE:");
                this.value = data;
                var octets = new Uint8Array(data);
                console.log("0x" + u8AToHexString(octets).toUpperCase());
                var proximity_band = octets[0];
                var client_rssi = octets[1];
                client_rssi = (256 - client_rssi) * -1;
                allLedsOff();
                if (proximity_band == 0) {
                    // means the user has turned off proximity sharing
                    // so we just want to switch off the LEDs
                    console.log("Proximity Sharing OFF");
                } else {
                    var proximity_led = getLed(proximity_band - 1);
                    proximity_led.writeSync(1);
                    console.log("Client RSSI: " + client_rssi);
                }
                callback(this.RESULT_SUCCESS);
            }
```

```
        })
    ]
}),
```

The value written to the Client Proximity characteristic consists of two parts or “fields”. You may remember this from the work we did to design the profile earlier on. They’re each of type uint8 and the first is the “proximity band” which takes a value of 0, 1, 2 or 3 where 0 means “stop proximity monitoring”, 1 means “near”, 2 means “middle distance” and 3 means “far away but still in range”.

Checkpoint

Run the modified code and test it using either nRF Connect on a smartphone or Bluetooth Developer Studio. Write the 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 console as a Client RSSI value of -56. 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.

Console output should look similar to this and of course you should see the appropriate LED illuminated.

```
pi@raspberrypi:~/projects/bdsk $ sudo node bdsk.js
on -> stateChange: poweredOn
on -> advertisingStart: success
on -> servicesSet: success
on -> accept, client: 61:87:ab:a7:c4:91
proximity monitoring.client proximity WRITE:
0x01C0
Client RSSI: -64
proximity monitoring.client proximity WRITE:
0x02C2
Client RSSI: -62
proximity monitoring.client proximity WRITE:
0x03C8
Client RSSI: -56
proximity monitoring.client proximity WRITE:
0x0000
Proximity Sharing OFF
Disconnected from address: 61:87:ab:a7:c4:91
```

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.*"

Let's add some more code.

```
bleno.on('disconnect', function (clientAddress) {
        console.log("Disconnected from address: " + clientAddress);
        if (link_loss_alert_level > 0) {
                console.log("Alerting due to link loss");
                flash_count = link_loss_alert_level * 4 * 15; // 4 per second, 15 seconds or 30
seconds
                startBeepingAndflashingAll(250, link_loss_alert_level);
        }
});
```

Checkpoint

Run the modified code and test it using either nRF Connect on a smartphone or Bluetooth Developer Studio. Write the values which have 0x00, 0x01 or 0x02 to the Alert Level characteristic of the Link Loss service and then in each case, disconnect from the Raspberry Pi and observe the results.

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.

So let's add some code which will respond to temperature indications being enabled or disabled. When enabled, we'll transmit an indication once every second.

```
//  health thermometer service
new bleno.PrimaryService({
    uuid: '1809',
    characteristics: [
        // temperature measurement
        new bleno.Characteristic({
            value: 0,
            uuid: '2A1C',
```

```
            properties: ['indicate'],
            onSubscribe: function (maxValueSize, updateValueCallback) {
                console.log("subscribed to temperature measurement indications");
                // simulateTemperatureSensor(updateValueCallback);
                sampleTemperatureSensor(updateValueCallback);
            },

            // If the client unsubscribes, we stop broadcasting the message
            onUnsubscribe: function () {
                console.log("unsubscribed from temperature measurement indications");
                clearInterval(temperature_timer);
            }
        })
    ]
}),
```

You have a choice now of acquiring real temperature data from the sensor in our circuit or, if you chose not to include it, using simulated temperature data generated by the Raspberry Pi code. Code for either scenario appears next. Copy whichever you're intending to choose into your code and make sure it's called from the onSubscribe function in the previous code fragment.

```
function simulateTemperatureSensor(updateValueCallback) {
        temperature_timer = setInterval(function () {
                celsius = celsius + 0.1;
                if (celsius > 100.0) {
                        celsius = 0.00;
                }
                console.log("simulated temperature: " + celsius + "C");
                celsius_times_10 = celsius * 10;
                var value = [0, 0, 0, 0, 0];
                // temperatures are in IEEE-11073 FLOAT format which consists of a single
                // octet exponent followed by three octets comprising the mantissa. These
                // 4 octets are in little endian format.
                value[4] = 0xFF; // exponent of -1
                value[3] = (celsius_times_10 >> 16);
                value[2] = (celsius_times_10 >> 8) & 0xFF;
                value[1] = celsius_times_10 & 0xFF;
                var buf = Buffer.from(value);
                updateValueCallback(buf);
        }, 1000);
}
```

```
function sampleTemperatureSensor(updateValueCallback) {
	var tempSensor = mcpadc.open(0, { speedHz: 1200000 }, function (err) {
		if (err) throw err;
		temperature_timer = setInterval(function () {
			// sensor provides a voltage output that is linearly proportional to the
Celsius (centigrade) temperature.
			// TMP36 characteristics: 500mV offset, Scaling 10mV per degree C.
			// Example: 750mV = 25 degrees C
			//  750mV - 500mV = 250mV
			//  250mV / 10 = 25 degrees C
			//
			// sensor value is in the range 0-1 and represents a fraction of the
reference voltage
			// in our circuit the reference voltage is 3300mV
			//
			// with a multimeter I measured only 1600mV at the 3.3V pin so made
adjustments accordingly
			tempSensor.read(function (err, reading) {
				if (err) throw err;
				celsius = ((reading.value * 3300) - 500) / 10;
				console.log("temperature: " + celsius + "C");
				celsius_times_10 = celsius * 10;
				var value = [0, 0, 0, 0, 0];
				value[4] = 0xFF; // exponent of -1
				value[3] = (celsius_times_10 >> 16);
				value[2] = (celsius_times_10 >> 8) & 0xFF;
				value[1] = celsius_times_10 & 0xFF;
				var buf = Buffer.from(value);
				updateValueCallback(buf);
			});
		}, 1000);
	});
}
```

These two functions acquire a temperature sensor value, real or simulated and then formulate the temperature measurement characteristic value in the format given in the Bluetooth SIG specification and sets it in the characteristic. Setting the value when indications have been subscribed to is all we need to do for the indication to be transmitted.

I've included various comments in this code block. The key things to know are that the temperature measurement characteristic contains more than one field. We're only using two of them, but if you take a look at the specification you'll see there are others. The field field is a single octet and called FLAGS. When set to 0x00, the temperature measurement value field contains a Celsius value. If set to 0x01, it

contains a Fahrenheit value. I'm working in Celsius but if you prefer to use Fahrenheit, go ahead and change the code.

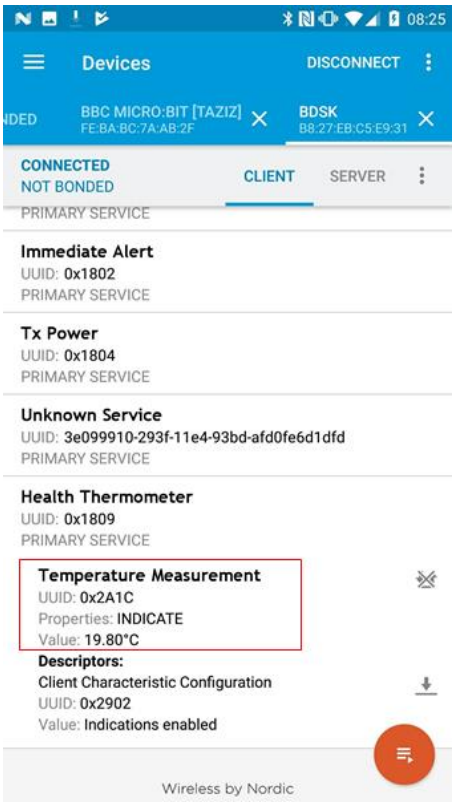
Note: You'll need to ensure that the calculation of temperature values from sensor readings is as specified in the data sheet for your particular make and model of sensor. If necessary, calibrate against readings from a thermometer and adjust your code until you're satisfied with the values reported.

Checkpoint

Run your modified code on the Raspberry Pi. Connect to the device using nRF Connect or Bluetooth Developer Studio. Subscribe to indications on the Temperature Measurement characteristic and watch values arrive at the client and logged in the Raspberry Pi console. Unsubscribe and ensure that temperature sampling stops on the Pi. Disconnect. Console output should look something like this:

```
pi@raspberrypi:~/projects/bdsk $ sudo node bdsk.js
on -> stateChange: poweredOn
on -> advertisingStart: success
on -> servicesSet: success
on -> accept, client: 62:3f:22:78:6a:46
subscribed to temperature measurement indications
temperature: 21.671554252199417C
temperature: 21.21212121212121C
temperature: 20.75268817204301C
temperature: 21.21212121212121C
temperature: 20.293255131964816C
temperature: 19.83382209188661C
temperature: 20.293255131964816C
temperature: 22.13098729227762C
unsubscribed from temperature measurement indications
Disconnected from address: 62:3f:22:78:6a:46
```

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



Testing - Videos

The Servers\Videos folder contains a video, 'testing_with_nrf_connect.mp4'. It involves an Arduino for most of the tests rather than Raspberry Pi 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 Raspberry Pi. Nice work!

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