Developer Study Guide: An introduction to Bluetooth Low Energy Development

Creating a Bluetooth Low Energy peripheral using Arduino

Version: 5.0.3

Last updated: 17th December 2018

Contents

Revision History

Version	Date	Author	Changes
1.0.0	1st May 2013	Matchbox	Initial version
2.0.0	1st September 2014	Martin Woolley, Bluetooth SIG	Replaced Button Click custom service with the Proximity Monitoring Service. Introduced the Time Monitoring Service. Described implementation of link loss, link loss alert level change and immediate alert use cases and included Arduino code.
2.0.4	2nd February 2015	Martin Woolley, Bluetooth SIG	Fixed some minor issues with the contents of this document; unpackaging instructions incortrectly assumed the Arduino BLE library would be in a zip file, which it is not. The circuit diagram for the suggested circuit had an error.
3.0.0	7th July 2016	Martin Woolley, Bluetooth SIG	Replaced Arduino Uno with Arduino/Genuino 101 Introduced use of Bluetooth Developer Studio in the design of the lab profile, the generation of skeleton code and testing of the peripheral device.
3.0.1	18th November	Martin Woolley, Bluetooth SIG	Added further information on setting up the Arduino development environment.

3.1.0	3rd October	Martin Woolley, Bluetooth SIG	Android code retired to the legacy folder and designated 'Android 4'. New Android lab and solution based on Android Studio instead of Eclipse and the newer Android 5+ APIs created.
3.2.0	16th December 2016	Martin Woolley, Bluetooth SIG	iOS Objective-C resources retired and replaced with new lab and solution written in Swift.
4.0.0	7th July 2017	Martin Woolley, Bluetooth SIG	Added a new lab, with associated solution source code, which explains how to use the Apache Cordova SDK to create a cross platform mobile application which uses Bluetooth Low Energy.
5.0.0	4th December 2017	Martin Woolley Bluetooth SIG	Arduino Primo now supported as well as the “end of life” Arduino 101. Added a thermistor to the electronics circuit and the Health Thermometer service to the Bluetooth profile. Indications used to deliver periodic temperature measurements. Split Arduino lab document into three documents; LE Theory, Profile Design and Profile Implementation and Testing - Arduino.

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 an Arduino. We've based the project on either the older Arduino 101 / Genuino 101 or the newer Arduino Primo, together with a simple, custom electronic circuit.

Goal

This document guides you through hardware assembly, coding and testing. You'll be turning your Arduino 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 Arduino 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 Arduino device will include an LCD display and LEDs and these will be used to indicate alarm conditions and whether the Arduino 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 Arduino
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**.

Arduino Exercise 1: Setting up

Software

Using Mac OSX:

Installing the Arduino software

1. Download the latest stable version of the Arduino Software for Mac OSX from here[1].
2. Install with reference to instructions and support information at the Arduino web site.
3. If you're using the Arduino 101, select Tools / Boards / Boards Manager and then search for and if necessary, install the item entitled "Intel Curie Boards". If you're using an Arduino Primo, **install Arduino Core for Nordic Semiconductor nRF5 based boards** from https://github.com/sandeepmistry/arduino-nRF5 following the set up instructions carefully.
4. Make sure under Tools you have the right Arduino board selected, either the Genuino/Arduino 101 or the Arduino Primo.

Using Windows:

Installing and setting up the Arduino IDE software

1. Download the latest stable version of the Arduino Software from here. It is available either as an installer or a zip archive, though both contain the same files.
2. Install with reference to instructions and support information at the Arduino web site.
3. If you're using the Arduino 101, select Tools / Boards / Boards Manager and then search for and if necessary, install the item entitled "Intel Curie Boards". If you're using an Arduino Primo, **install Arduino Core for Nordic Semiconductor nRF5 based boards** from https://github.com/sandeepmistry/arduino-nRF5 following the set up instructions carefully.
4. Make sure under Tools you have the right Arduino board selected, either the Genuino/Arduino 101 or the Arduino Primo.

[1] http://arduino.cc/en/Main/Software

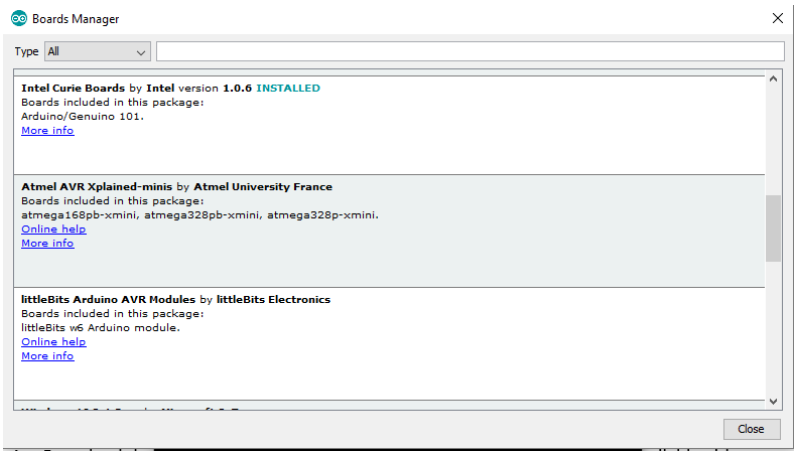


Figure 1 - installing the Intel Curie Boards package for Arduino 101

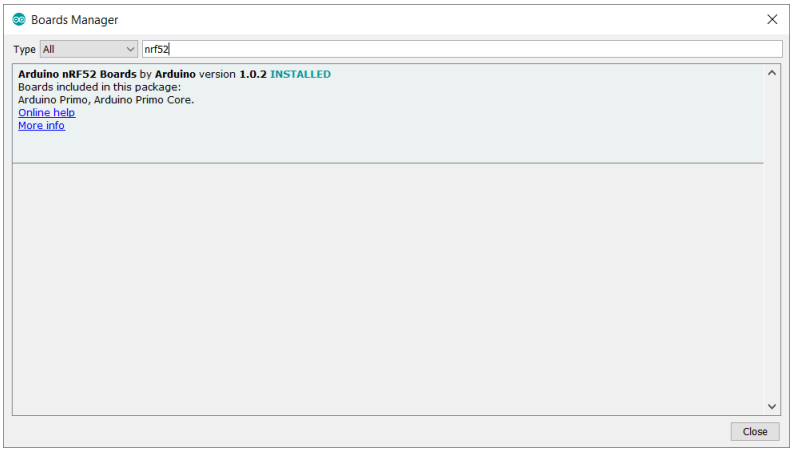


Figure 2 - installing the nRF52 Boards package for Arduino Primo

Hardware

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

Note that the supported Arduino boards have a Bluetooth Low Energy module built in so you do not need a seperate "shield" as was the case with older Arduino boards like the Uno.

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.

Arduino 101 or Arduino Primo

USB cable for connecting the Arduino to a computer

1 x red LED

1 x yellow LED

1 x green LED

3 x 56 Ohms 5% tolerance resistors

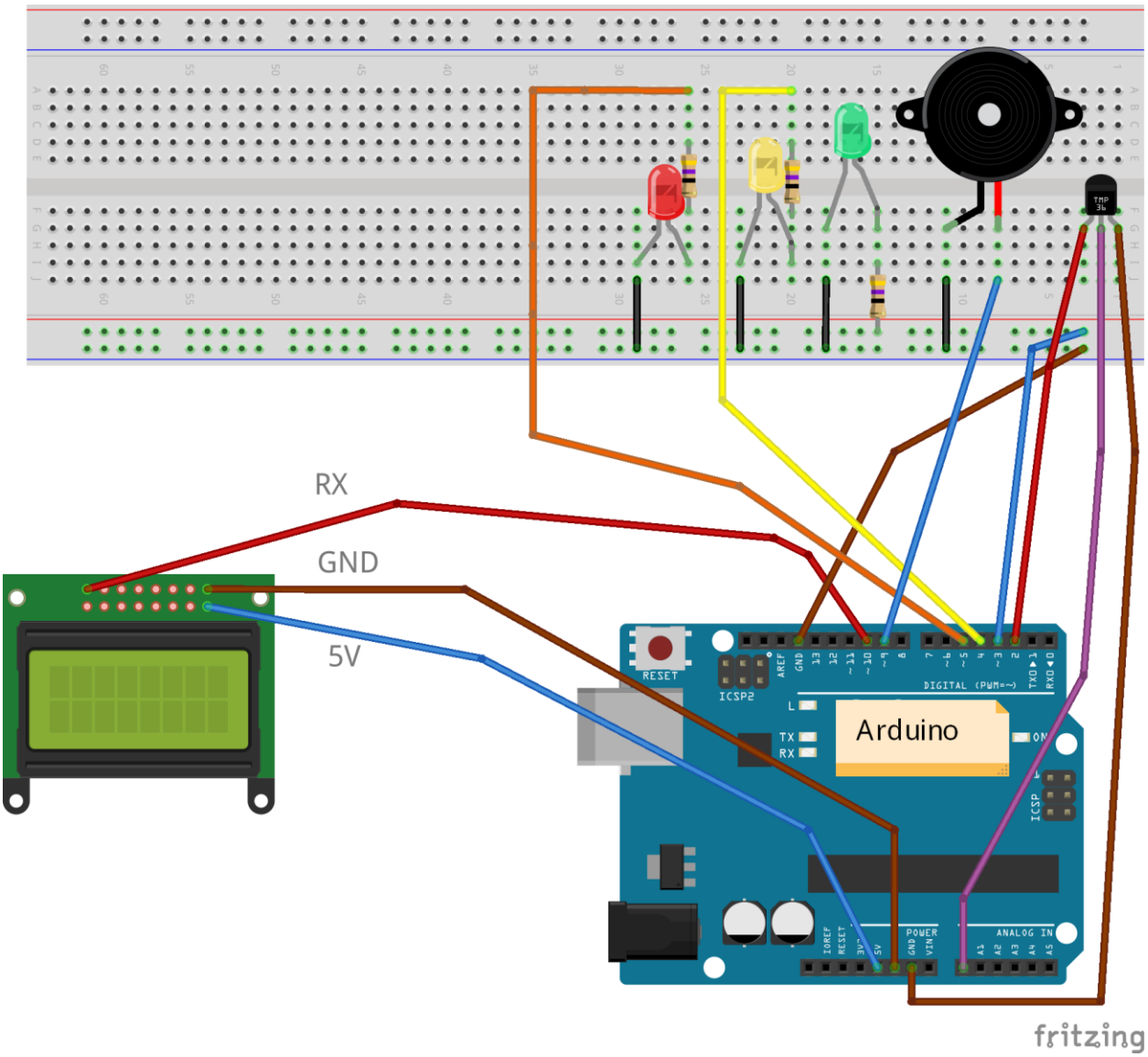
1 x ABT-402-RC PIEZO TRANSDUCER, 5V, PCB

1 x breadboard

1 x SparkFun serLCD compatible LCD display

1 x TMP36 temperature sensor or similar

Wires for forming connections



Arduino PIN Connections

PIN 2 -> TMP36 left pin (power)

PIN 3 -> Green LED +ve	PIN 4-> Yellow LED +ve
PIN 5 -> Red LED +ve	PIN 9 -> Buzzer
PIN10 -> LCD panel RX	PIN A0 -> TMP36 central pin (sensor values)
5V -> 5V LCD panel connector	GND #1 -> circuit ground
GND #2 -> GND LCD panel connector	GND #3 -> TMP36 right pin (ground)

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

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

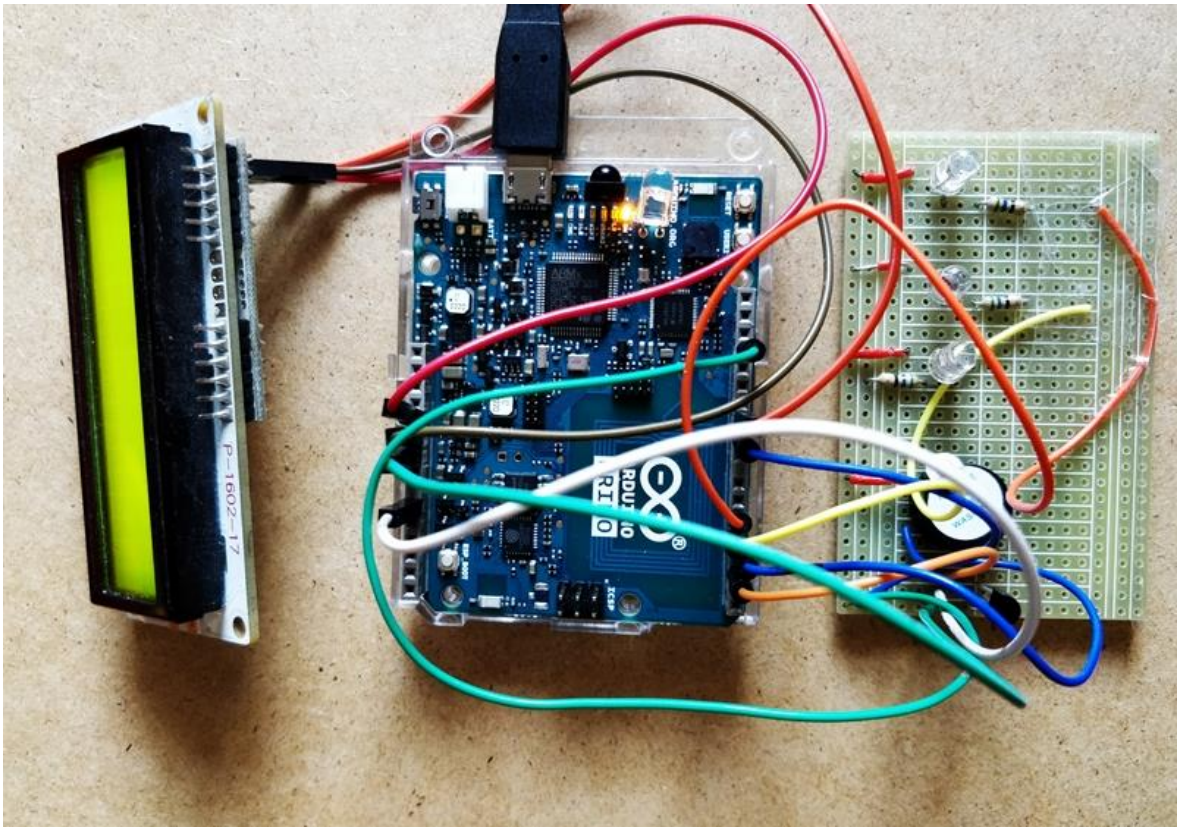


Figure 3 - Once happy with your breadboard, consider making a more permanent, soldered version!

Arduino 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 Arduino.

About Testing

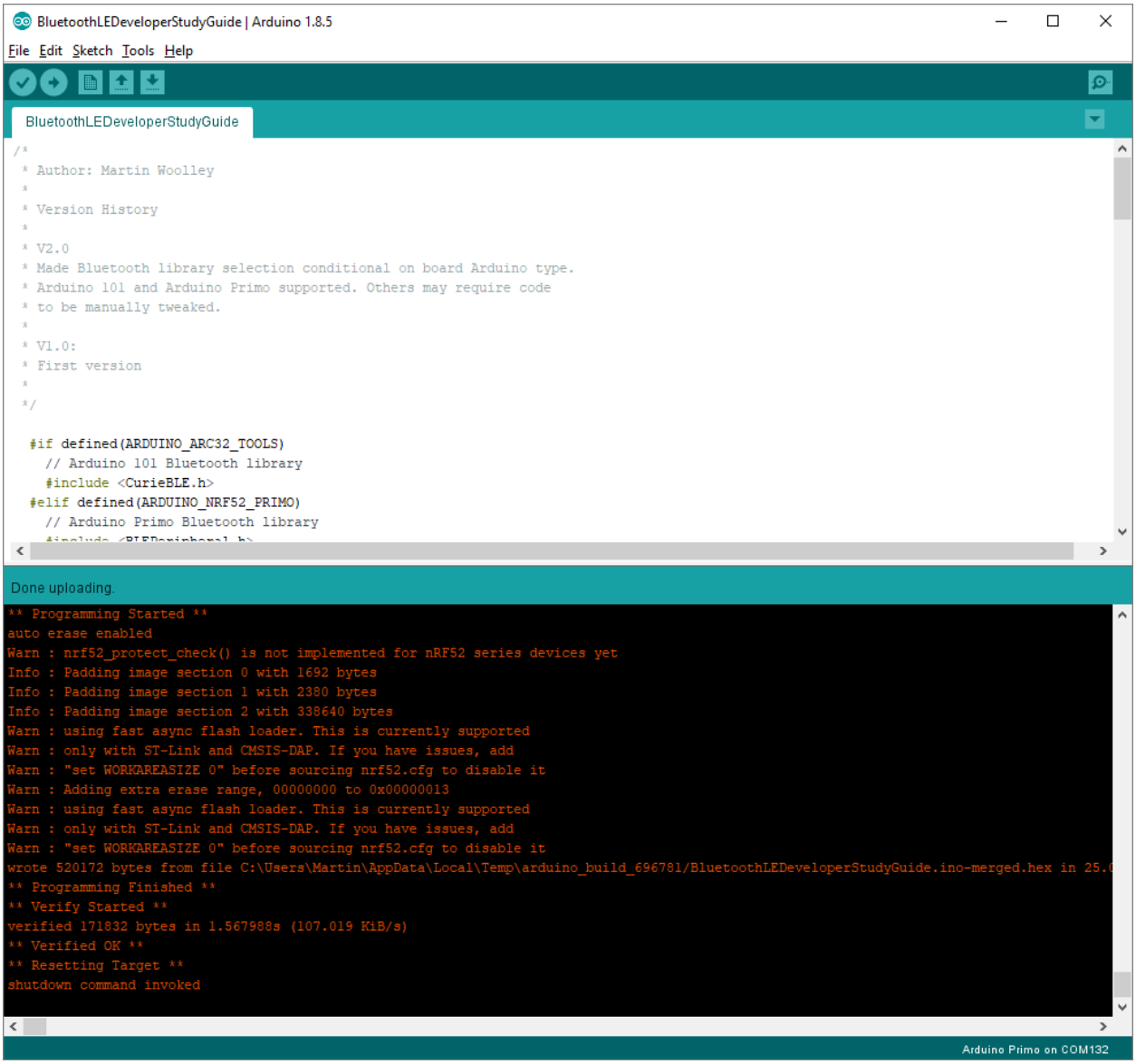
We'll test as we go so that we can validate each feature as we add it, rather than have to test everything all in one go and perhaps find bigger problems to diagnose and solve than if we'd progressed with smaller increments. To test your solution, you will use a smartphone application called nRF Connect,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

Initial Skeleton Code

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

Open the BluetoothLEDeveloperStudyGuide.ino file in the Arduino IDE.

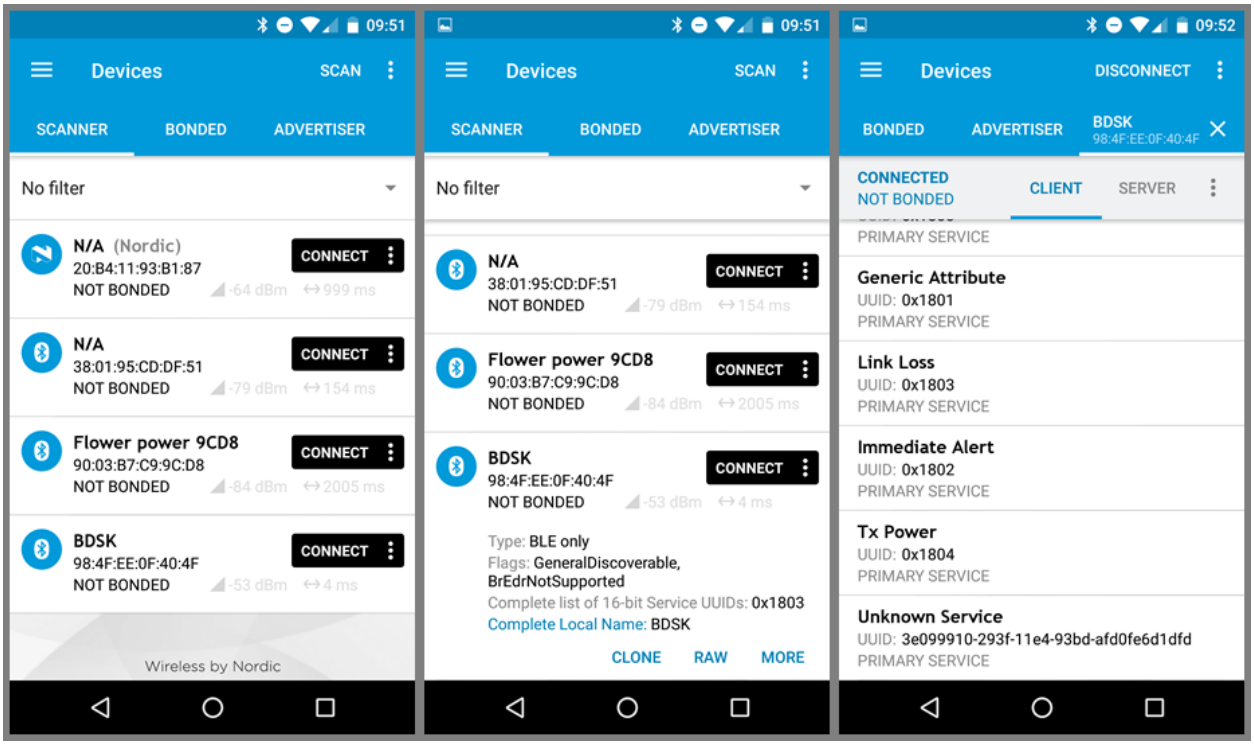
Plug your Arduino into your computer over USB and click the Upload button in the Arduino IDE.



Checkpoint

Your Arduino is now running code which creates the profile's services, characteristics and descriptors. It doesn't yet have the application logic we require but in all other respects, the profile is there. You can verify this for yourself 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 Arduino, which should be advertising with the name "BDSK". If you can't see it, try pressing the master reset button on your Arduino and then try scanning again after a few seconds.

Your GATT explorer application will hopefully let you see the content of advertising packets and the services and characteristics it has. Here are some screenshots from nRF Connect.



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.

Looking good! Mostly…..

If you’re using an Arduino Primo, you might notice an extra, unexpected service in the list. It’s the Device Firmware Update (DFU) service from Nordic Semiconductor. The Primo contains a Nordic nRF52 module in it and the toolchain automatically instantiates their DFU service. It does no harm but we don’t need it and unused services and characteristics consume memory, so if you do see this service, let’s get rid of it now.

Find the setup() function and add the lines shown here after the message which indicates that the attribute table has been constructed.

```
  Serial.println("attribute table constructed");
#if defined(ARDUINO_NRF52_PRIMO)
  // Arduino Primo Bluetooth library
  removeDfuService(true);
#endif
```

Build and flash to your Arduino and check that the DFU service is no longer included in the nRF Connect service list.

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. We'll also initialise the Arduino pins that we're using and our circuit board in the way we need.

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

```
#include <SoftwareSerial.h>
int link_loss_alert_level = 0;
int immediate_alert_level = 0;
int ledPin1     = 3;
int ledPin2     = 4;
int ledPin3     = 5;
int speakerPin  = 9;
int thermistorPin = 2;
int alert_may_be_required = 0;
#define lcdTxPin 10
SoftwareSerial LCD = SoftwareSerial(0, lcdTxPin);
const int LCDdelay=500;
char buf[16];
```

Add the following functions, which are concerned with use of the LCD display:

```
void lcdPosition(int row, int col) {
  LCD.write(0xFE);   //command flag
  LCD.write((col + row*64 + 128));    //position
  delay(LCDdelay);
}
void lcdText(String text) {
    clearLCD();
    LCD.print(text);
}
void clearLCD(){
  LCD.write(0xFE);   //command flag
  LCD.write(0x01);   //clear command.
  delay(LCDdelay);
}
```

```
void initLcd() {
  // backlight on
  LCD.write(0x7C);
  LCD.write(157);
  //turns the underline cursor off
  LCD.write(0xFE); //command flag
  LCD.write(12); // 0x0C
  //this turns the box cursor off
  LCD.write(0xFE); //command flag
  LCD.write(12); // 0x0C
  delay(LCDdelay);
}
void serCommand(){   //a general function to call the command flag for issuing all other commands
  LCD.write(0xFE);
}
```

And add the following functions, which will let us control the LED lights and buzzer in various ways. What we're doing here is standard Arduno programming, controlling things connected to the Arduino with digital and analogue write operations. The Arduino web site has more information on this.

```
void flash(unsigned char led,uint16_t delayms, unsigned char times){
  for (int i=0;i<times;i++) {
    digitalWrite(led, HIGH);
    delay(delayms);
    digitalWrite(led, LOW);
    delay(delayms);
  }
}
void beepAndFlashAll(uint16_t delayms){
  Serial.println(F("BEEP + FLASH ALL"));
  digitalWrite(ledPin1, HIGH);
  digitalWrite(ledPin2, HIGH);
  digitalWrite(ledPin3, HIGH);
  if (buzzer_alert_level > 0) {
      tone(speakerPin, 262, delayms);
  }
  delay(delayms);
  digitalWrite(ledPin1, LOW);
  digitalWrite(ledPin2, LOW);
  digitalWrite(ledPin3, LOW);
```

```
  delay(delayms);
}

void allLedsOff() {
  digitalWrite(ledPin1, LOW);
  digitalWrite(ledPin2, LOW);
  digitalWrite(ledPin3, LOW);
}
// determine the pin to write to to control the colour of LED related to an alert level value
int getPinNumber(uint8_t level) {
  switch (level) {
    case 0:   return ledPin1;
    case 1:   return ledPin2;
    case 2:   return ledPin3;
    default:  return ledPin1;
  }
}
```

Our last step before we turn our attention to the Bluetooth operations and our lab requirements is to add some code to the Arduino setup() function as shown here:

```
void setup(void)
{
// ..............
// keep all the generated code and add the following lines towards the end of the function

  pinMode(ledPin1, OUTPUT);
  pinMode(ledPin2, OUTPUT);
  pinMode(ledPin3, OUTPUT);
  pinMode(thermistorPin, OUTPUT);
  digitalWrite(thermistorPin, HIGH);

  allLedsOff();

  LCD.begin(9600);
  backlightOn() ;
  clearLCD();
  lcdPosition(0,0);
  lcdText("Ready...");
}
```

What we've done here is to incorporate a library which lets us use the LCD display (SoftwareSerial.h) and then added some functions which allow us to use it and we've also added a few functions which allow us to flash LEDs with or without the buzzer being sounded. We've also initialised the pins we're using on the Arduino. Note that the thermistorPin is actually being used to power the temperature sensor and this is achieved by setting it up as a digital output pin and writing a value of HIGH to it.

Checkpoint

Upload your code to the connected Arduino. After a short delay, you should see 'Ready...' appear on the LCD display. LEDs on the circuit board should not be lit and the buzzer should be silent.

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. In the loop() function we have a while loop which executes as long as we have a Bluetooth connection. Add the highlighted code to react to a client writing to the Alert Level characteristic of the Link Loss service as shown.

```
if (central) {
  lcdText("Connected");
  while (central.connected()) {

      if (LinkLoss_AlertLevel.written()) {
        Serial.println("LinkLoss_AlertLevel.written()");
        link_loss_alert_level = LinkLoss_AlertLevel.value()[0];
        int ledpin = getPinNumber(link_loss_alert_level);
        flash(ledpin,250,4);
      }
      if (ImmediateAlert_AlertLevel.written()) {
       Serial.println("ImmediateAlert_AlertLevel.written()");
      }
      if (ProximityMonitoring_ClientProximity.written()) {
       Serial.println("ProximityMonitoring_ClientProximity.written()");
      }
  }
  lcdText("Disconnected");
```

Let's give the link loss alert level a default value too. Add the following to setup()

```
LinkLoss_AlertLevel.setValue(initial_ll_alert_level, 1);
```

Checkpoint

Upload your modifed code to the Arduino and test it now using the nRF Connect application. Scan and find the BDSK device and connect to it. The LCD display should say "Connected". 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 Arduino and the LCD display should show “Disconnected”.

Requirement 3 – Immediate Alert, Alert Level

Requirement 3 states that the smartphone can send the Arduino an immediate alert level value of 0, 1 or 2 and in response, the Arduino 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 following constants to your code, near the top. These are used in controlling the buzzer.

```
#define SOUND_BUZZER_FOR_3_SECONDS 3
#define SOUND_BUZZER_FOR_15_SECONDS 15
#define SOUND_BUZZER_FOR_30_SECONDS 30
#define BUZZER_OFF          0
#define ONE_SECOND          1000
#define HALF_SECOND         500
int alert_counter = BUZZER_OFF;
int buzzer_alert_level = 0;
```

Add the alertControl function. This will be called from our main loop() function and a counter will be used to control whether or not the buzzer should sound. We decrement the counter each time the function is called so that eventually, when it hits zero, the buzzer makes no sound.

```
void alertControl(void) {
  if (alert_counter > BUZZER_OFF && alert_counter-- > BUZZER_OFF) {
     beepAndFlashAll(HALF_SECOND);
  }
}
```

Add the highlighted code to your loop() function. We’re intercepting the write to the immediate alert service’s alert level characteristic and setting the alert_counter variable according to the value being written. We’ve also added a call to the alertControl function we just added so that it’s called every time through this loop.

```
  if (central) {
    lcdText("Connected");
    while (central.connected()) {

        if (LinkLoss_AlertLevel.written()) {
          Serial.println("LinkLoss_AlertLevel.written()");
          link_loss_alert_level = LinkLoss_AlertLevel.value()[0];
```

```
          int ledpin = getPinNumber(link_loss_alert_level);
          flash(ledpin,250,4);
        }
        if (ImmediateAlert_AlertLevel.written()) {
         Serial.println("ImmediateAlert_AlertLevel.written()");
         immediate_alert_level = ImmediateAlert_AlertLevel.value()[0];
         buzzer_alert_level = immediate_alert_level;
         // flash more times for higher alert levels
         alert_counter = SOUND_BUZZER_FOR_3_SECONDS + immediate_alert_level;
        }
        if (ProximityMonitoring_ClientProximity.written()) {
         Serial.println("ProximityMonitoring_ClientProximity.written()");
        }
        alertControl();
    }
    lcdText("Disconnected");
```

Checkpoint

Upload your revised code and test it using nRF Connect on a smartphone. 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.

Requirement 4 – Proximity Monitoring

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

```
  if (central) {
    lcdText("Connected");
```

```
    while (central.connected()) {

      if (LinkLoss_AlertLevel.written()) {
        Serial.println("LinkLoss_AlertLevel.written()");
        link_loss_alert_level = LinkLoss_AlertLevel.value()[0];
        int ledpin = getPinNumber(link_loss_alert_level);
        flash(ledpin,250,4);
      }
      if (ImmediateAlert_AlertLevel.written()) {
       Serial.println("ImmediateAlert_AlertLevel.written()");
       immediate_alert_level = ImmediateAlert_AlertLevel.value()[0];
       // flash more times for higher alert levels
       alert_counter = SOUND_BUZZER_FOR_3_SECONDS + immediate_alert_level;
      }
      if (ProximityMonitoring_ClientProximity.written()) {
       Serial.println("ProximityMonitoring_ClientProximity.written()");
       int proximity_band = ProximityMonitoring_ClientProximity.value()[0];
       int client_rssi = ProximityMonitoring_ClientProximity.value()[1];
       client_rssi = (256 - (int) 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
          return;
       }
       int ledpin = getPinNumber(proximity_band - 1);
       digitalWrite(ledpin, HIGH);
       char buf[16];
       sprintf(buf, "Client RSSI: %d", client_rssi);
       lcdText(buf);
      }
      alertControl();
    }
    lcdText("Disconnected");
    allLedsOff();
```

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

Upload the modified code and test it using nRF Connect on a smartphone. 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 be displayed as a Client RSSI value of -56 on the LCD display.

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. This time we're initiating the alerting process but setting the control variable 'alert_counter' so that the buzzer sounds and LEDs flash for a whole 90 seconds.

```
if (central) {
  lcdText("Connected");
  alert_may_be_required = 1;
  while (central.connected()) {

      if (LinkLoss_AlertLevel.written()) {
        Serial.println("LinkLoss_AlertLevel.written()");
        link_loss_alert_level = LinkLoss_AlertLevel.value()[0];
        int ledpin = getPinNumber(link_loss_alert_level);
        flash(ledpin,250,4);
      }
      if (ImmediateAlert_AlertLevel.written()) {
       Serial.println("ImmediateAlert_AlertLevel.written()");
       immediate_alert_level = ImmediateAlert_AlertLevel.value()[0];
       // flash more times for higher alert levels
       alert_counter = SOUND_BUZZER_FOR_3_SECONDS + immediate_alert_level;
      }
      if (ProximityMonitoring_ClientProximity.written()) {
       Serial.println("ProximityMonitoring_ClientProximity.written()");
       int proximity_band = ProximityMonitoring_ClientProximity.value()[0];
       int client_rssi = ProximityMonitoring_ClientProximity.value()[1];
       client_rssi = (256 - (int) 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
          return;
       }
       int ledpin = getPinNumber(proximity_band - 1);
```

```
          digitalWrite(ledpin, HIGH);
          char buf[16];
          sprintf(buf, "Client RSSI: %d", client_rssi);
          lcdText(buf);
         }
         alertControl();
    }
    sprintf(buf, "Disconnected: %d", link_loss_alert_level);
    lcdText(buf);
    allLedsOff();
    if (link_loss_alert_level > 0 && alert_may_be_required == 1) {
      allLedsOff();
      buzzer_alert_level = link_loss_alert_level;
      if (link_loss_alert_level == 1) {
          alert_counter = SOUND_BUZZER_FOR_15_SECONDS;
      } else {
          alert_counter = SOUND_BUZZER_FOR_30_SECONDS;
      }
      alert_may_be_required = 0;
    }
    alertControl();
}
```

Note that we've replaced our simple LCD "Disconnected" message with a similar message which also includes the value of the Alert Level characteristic of the Link Loss service, which we saved in a convenient variable. The value of this variable is key in what happens next. If it's not zero then the device should alert. If it's zero if should not.

Checkpoint

Upload the modified code and test it using nRF Connect on a smartphone. 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 Arduino 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.

Add the following variables near the top of your source code.

```
// temperature
int thermistor_pin = 0;
int thermistor_val = 0;
int temperature_subscribed = 0;
float voltage;
float celsius;
int celsius_times_10;
unsigned long timestamp;
```

We'll be using a timestamp to control transmitting temperature measurement values once per second. Initialise the timestamp variable in the setup() function:

```
  // begin advertising
  blePeripheral.begin();
  Serial.println("advertising");
 // set timestamp which we use for periodic temperature indications
  timestamp = millis();
```

We need to handle the user subscribing or unsubscribing from temperature measurement indications. Our API generates events whenever this happens, so our next task is to register functions which will handle these events. Modify your setup() code as shown:

```
  blePeripheral.addAttribute(HealthThermometer_TemperatureMeasurement);
  // establish event handlers for indication subscription/unsubscription
  HealthThermometer_TemperatureMeasurement.setEventHandler(BLESubscribed,
temperatureSubscribed);
  HealthThermometer_TemperatureMeasurement.setEventHandler(BLEUnsubscribed,
temperatureUnsubscribed);
```

```
  LinkLoss_AlertLevel.setValue(initial_ll_alert_level, 1);
  Serial.println("attribute table constructed");
```

If the user subscribes to indications, the function temperatureSubscribed() will be called. If they unsubscribe, temperatureUnsubscribed() will be called.

Add each of these functions to your source code:

```
void temperatureSubscribed(BLECentral& central, BLECharacteristic& characteristic) {
  Serial.println(F("subscribed to temperature indications"));
  temperature_subscribed = 1;
}
void temperatureUnsubscribed(BLECentral& central, BLECharacteristic& characteristic) {
  Serial.println(F("unsubscribed from temperature indications"));
  temperature_subscribed = 0;
}
```

As you can see, all we do is set or unset a flag. We'll use the flag inside the main Arduino loop() function. Let's add the code to do that next:

```
    // while the central is still connected to peripheral:
    while (central.connected()) {
      if (temperature_subscribed == 1) {
        if ((millis() - timestamp) >= 1000) {
          celsius = readTemperature(thermistor_pin);
          celsius_times_10 = celsius * 10;
          timestamp = millis();
          /*
             Ref:
https://www.bluetooth.com/specifications/gatt/viewer?attributeXmlFile=org.bluetooth.characteris
tic.temperature_measurement.xml


             The Temperature Measurement characteristic is used to send a temperature
measurement.
             Included in the characteristic are a Flags field (for showing the units of
temperature
             and presence of optional fields), the temperature measurement value and, depending
             upon the contents of the Flags field, the time of that measurement and the
temperature type.


             If the value of bit 1 of the Flags field is 0 and bit 2 is 0, the structure of the
             Temperature Measurement characteristic consists of two fields in this order;
                 Flags and Temperature Measurement Value.
```

```
            The temperature measurement field is in the IEEE 11073 32-bit FLOAT format


          */


          // in IEEE 11073 1st byte is Exponent and the following 3 bytes are the mantissa in
big endian format
          // For example, Consider a temperature measurement of 36.4 degrees Celsius with
precision of 0.1 degrees Celsius.
          // The IEEE 11073 FLOAT-Type representation is a 32-bit value consisting of an
exponent of an 8-bit signed integer
          // followed by a mantissa of a 24-bit signed integer; here, the exponent is -1 (0xFF)
and the mantissa is 364 (0x00016C).
          // Therefore, the FLOAT-Type representation of 36.4 is 0xFF00016C.


          unsigned char value[5] = {0,0,0,0,0};


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


          // little endian
          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;
          Serial.print(value[0]);
          Serial.print(",");
          Serial.print(value[1]);
          Serial.print(",");
          Serial.print(value[2]);
          Serial.print(",");
          Serial.print(value[3]);
          Serial.print(",");
          Serial.print(value[4]);
          Serial.println(",");


          Serial.print(celsius);
          Serial.println(" degrees C");
          HealthThermometer_TemperatureMeasurement.setValue(value,5);
        }
      }
```

This block of code checks to see whether indications have been subscribed to and then if it's been at least a second since we last sent an indication, it reads from the temperature sensor, formulates the temperature measurement 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.

There's still one function we haven't written. The code most recently added includes a call to a function called readTemperature. Let's add that function now.

```
float readTemperature(int pin)
{
  voltage = (analogRead(pin) * 0.004882814);
  float celsius = (voltage - 0.5) * 100.0;
  return celsius;
}
```

You'll need to check the specification for the temperature sensor you're using and ensure that this function implements the correct formulae for transforming a voltage measurement into a temperature in some unit. I followed the instructions for my TMP36 and found I was getting values that were higher than expected. I tested with the sensor connected to various different devices and I tested with several different TMP36 sensors. I got consistent values in all cases, but they were all consistently higher than the ambient temperature, measured by independent means. I accomodated this variance by changing the voltage offset used in my formula. In effect, I calibrated my sensor using a thermometer. You may need to do the same. Welcome to the real world.

Testing - Videos

The Arduino\Videos folder contains a video, 'testing_with_nrf_connect.mp4'. 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 Arduino. Nice work!

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