Lab 3: Local Connectivity

Learning outcome: Understand how Bluetooth LE communication works, and how it can be used to transmit and receive basic data

# Introduction

## Lab overview

In this lab, you will learn how to communicate with the outside world through Bluetooth, and more specifically Bluetooth Low Energy -- or BLE. We will program the embedded device to transmit live on-board sensor data to a mobile app that can communicate with BLE devices.

## Hardware and Software Requirements

* The DISCO-L475VG-IOT01A board
* An Android mobile phone, with a Bluetooth LE Scanner App
* Mbed Studio or another suitable development environment

Before getting started, you will need to include the BLE utility library from here in your Mbed Studio project:  
<https://github.com/ARMmbed/mbed-os-ble-utils>

Just follow the same process for adding the board support library (which will also need to be included) to the project in the earlier lab.

# Working with Bluetooth

Detailed information on the Mbed OS Bluetooth APIs is available here:

<https://os.mbed.com/docs/mbed-os/v6.10/apis/bluetooth-apis.html>

And, you can find a range of examples here:

<https://github.com/ARMmbed/mbed-os-example-ble/>

## Configuration

Bluetooth LE can appear quite complex to get started with, but by writing our code in an efficient manner, we can make it quite straightforward.

# Application Code

We will program the embedded device to advertise its presence over BLE, and transmit live readings from the temperature, humidity, and pressure sensors that we used in the previous lab. We will also allow the on-board LED to be toggled remotely.

## High Level Overview

* Initialise BLE
* Advertise presence
* Accept connections, and periodically transmit sensor readings over the connection
* Respond to write events, to allow the on-board LED to be toggled

## Initialising BLE

To get started with BLE, we need to create some utility classes that will manage the interface to the BLE API.

First, create a class to manage the BLE process:

| class Lab3ServerProcess : public BLEProcess {  public:  Lab3ServerProcess(events::EventQueue &event\_queue, BLE &ble\_interface)  : BLEProcess(event\_queue, ble\_interface) {}  const char \*get\_device\_name() override {  static const char name[] = "Lab 3 Server";  return name;  }  }; |
| --- |

Here, we’ve created a class called Lab3ServerProcess, which subclasses BLEProcess, and simply returns the name that we want to advertise.

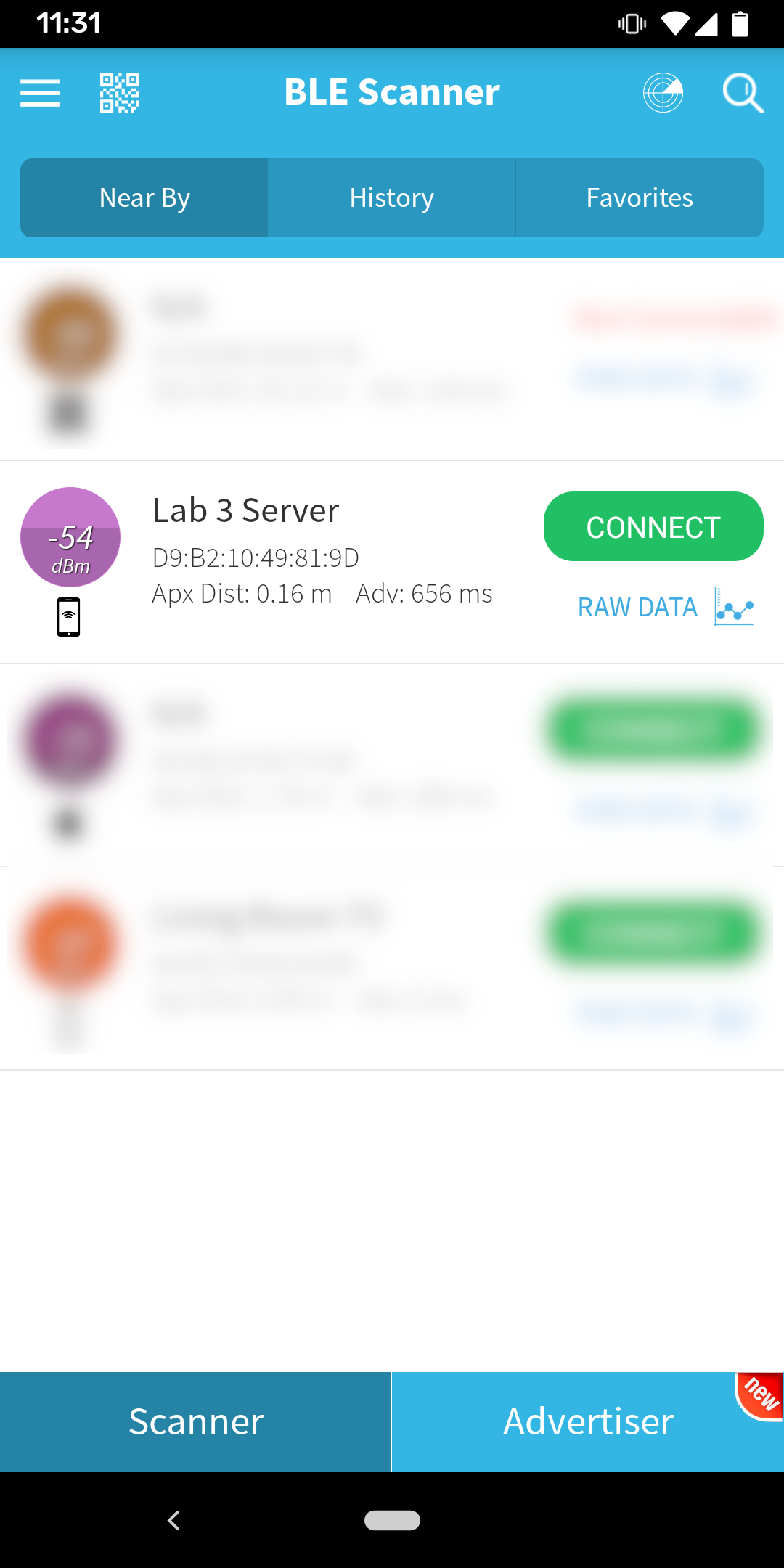
Now, we need to create a server class that will manage the service and characteristics that we support:

| class Lab3Server : ble::GattServer::EventHandler {  public:  Lab3Server() {}  ~Lab3Server() {}  void start(BLE &ble, events::EventQueue &event\_queue) {  const UUID uuid = GattService::UUID\_ENVIRONMENTAL\_SERVICE;  GattCharacteristic \*charTable[] = {};  GattService sensorService(uuid, charTable,  sizeof(charTable) / sizeof(charTable[0]));  ble.gattServer().addService(sensorService);  ble.gattServer().setEventHandler(this);  printf("Service started.\n");  }  }; |
| --- |

This should be enough to advertise our embedded system’s presence over BLE, so to try it out, initialise BLE in the main function:

| static EventQueue event\_queue(10 \* EVENTS\_EVENT\_SIZE);  int main() {  BLE &ble = BLE::Instance();  printf("Lab 3 - BLE");  Lab3ServerProcess ble\_process(event\_queue, ble);  Lab3Server server;  ble\_process.on\_init(callback(&server, &Lab3Server::start));  ble\_process.start();  return 0;  } |
| --- |

Build and run this on the embedded device, and open up the BLE scanning app on your mobile device. Depending on which BLE scanning app you use, you should see something like the following:



If you connect to the advertised server, you will see that an “Environmental” service exists, but with no further details. This is the service that we will be populating with sensor data.

## Advertising Sensor Data

As in Lab 2, we have to start by initialising the sensors we are going to use. We will be using the Temperature, Humidity, and Pressure sensors in this lab, so create a function to initialise them, and augment main to call this before the Bluetooth setup code:

| static void initialiseSensors() {  BSP\_TSENSOR\_Init();  BSP\_HSENSOR\_Init();  BSP\_PSENSOR\_Init();  } |
| --- |

Now that the sensors are initialised, we can think about how we transmit their readings.

To do this, for each sensor we are going to create a BLE Characteristic that is associated with the Environmental service we have already created. To help us do this, we’ll create a helper class for managing the “current value” of that characteristic.

The helper class should look like the following:

| class SensorReadingCharacteristic : public GattCharacteristic {  public:  SensorReadingCharacteristic(const UUID &uuid)  : GattCharacteristic(  uuid, (uint8\_t \*)&stringRepr\_, sizeof(stringRepr\_),  sizeof(stringRepr\_),  GattCharacteristic::BLE\_GATT\_CHAR\_PROPERTIES\_READ |  GattCharacteristic::BLE\_GATT\_CHAR\_PROPERTIES\_NOTIFY),  internalValue\_(0) {  updateStringRepr();  }  void updateValue(BLE &ble, float newValue) {  internalValue\_ = newValue;  updateStringRepr();  ble.gattServer().write(getValueHandle(), (uint8\_t \*)&stringRepr\_,  sizeof(stringRepr\_));  }  private:  float internalValue\_;  uint8\_t stringRepr\_[16];  void updateStringRepr() {  sprintf((char \*)stringRepr\_, "%f", internalValue\_);  }  }; |
| --- |

What’s happening here is we’re tracking the current value of the sensor in the internalValue\_ field, and then converting it into a string representation, when the value is updated. Later on, we’ll call updateValue for each characteristic representing a sensor, and pass in the current reading of that sensor.

When updateValue is called, the internal value is updated, the string representation is generated, and the value of the string representation is transmitted over BLE. We’re using a string representation here so that you can see the value in your BLE scanner app. If you were writing an Android app, there’d be no need to do the string conversion.

We’re also saying that these characteristics can be read from, and they can also notify the client when their value changes. This lets us repeatedly update the value, and the client will see those changes when it is connected.

Now that we have this class, we’ll create an instance of it for each of our sensors. To do this, we’ll create local fields in our Lab3Server class, and initialise them accordingly:

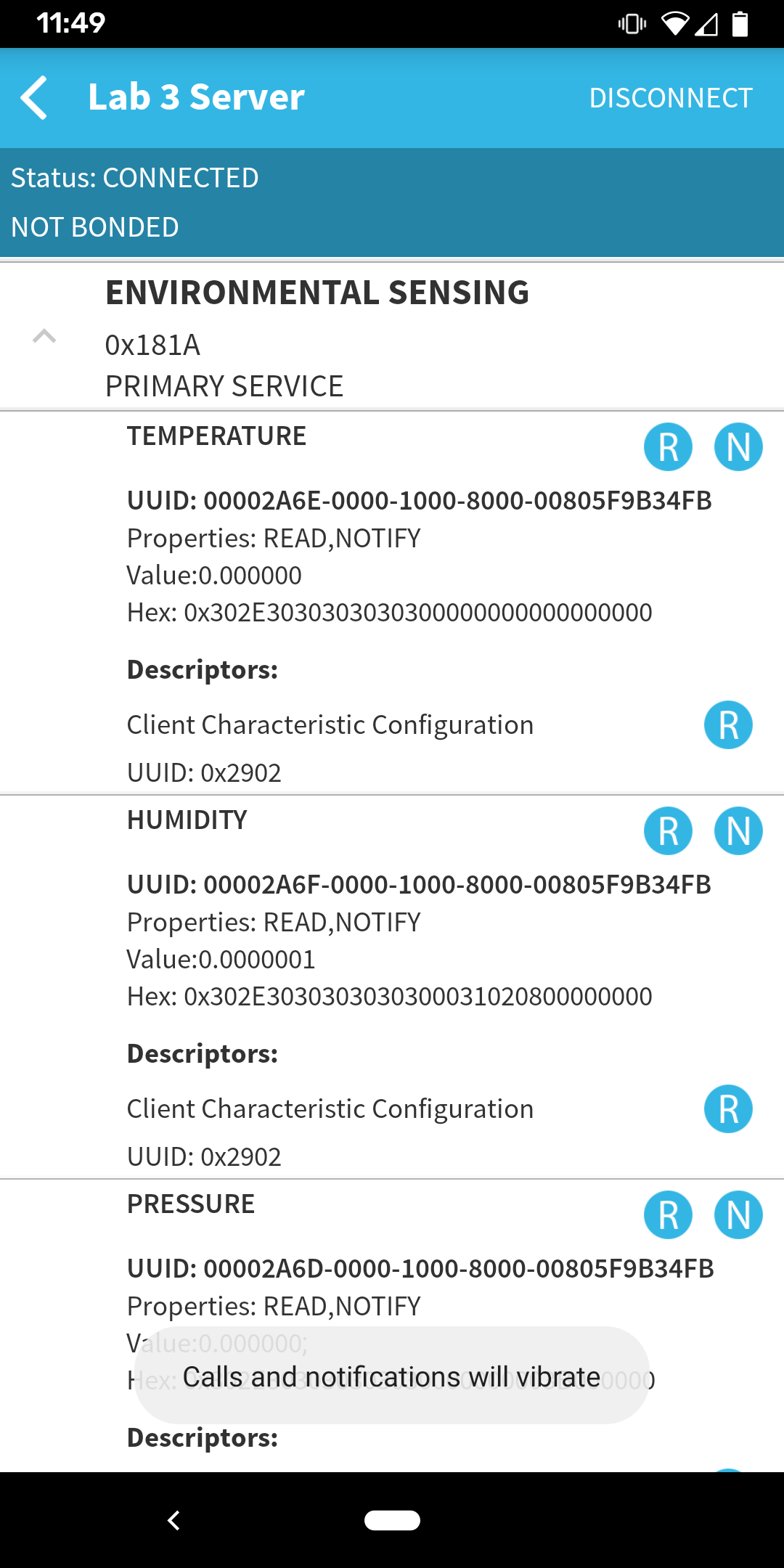
| class Lab3Server : ble::GattServer::EventHandler {  public:  Lab3Server()  : temperature\_(GattCharacteristic::UUID\_TEMPERATURE\_CHAR),  humidity\_(GattCharacteristic::UUID\_HUMIDITY\_CHAR),  pressure\_(GattCharacteristic::UUID\_PRESSURE\_CHAR) {}    // ... existing code ... //  private:  SensorReadingCharacteristic temperature\_;  SensorReadingCharacteristic humidity\_;  SensorReadingCharacteristic pressure\_;  }; |
| --- |

What we’ve done is to create three characteristics, one each for temperature, humidity, and pressure. We’ve also initialised them (in the constructor) with the correct identifiers, so that BLE clients understand what these values are.

Finally, we have to tell our service to advertise these characteristics, by modifying the characteristics array in the start method to include them:

| GattCharacteristic \*charTable[] = {&temperature\_, &humidity\_, &pressure\_}; |
| --- |

Once you’ve done this - try it out! If all goes well, after you build and run, you should be able to connect to the device, and see those three characteristics appear:



But - all the values are zero!

This is because we haven’t written the logic to update the values yet -- however, it’s important you reach this step before we go any further.

We’ll update these values once every second, so to do this, we’ll create a function that performs the update, and connect it to the event queue.

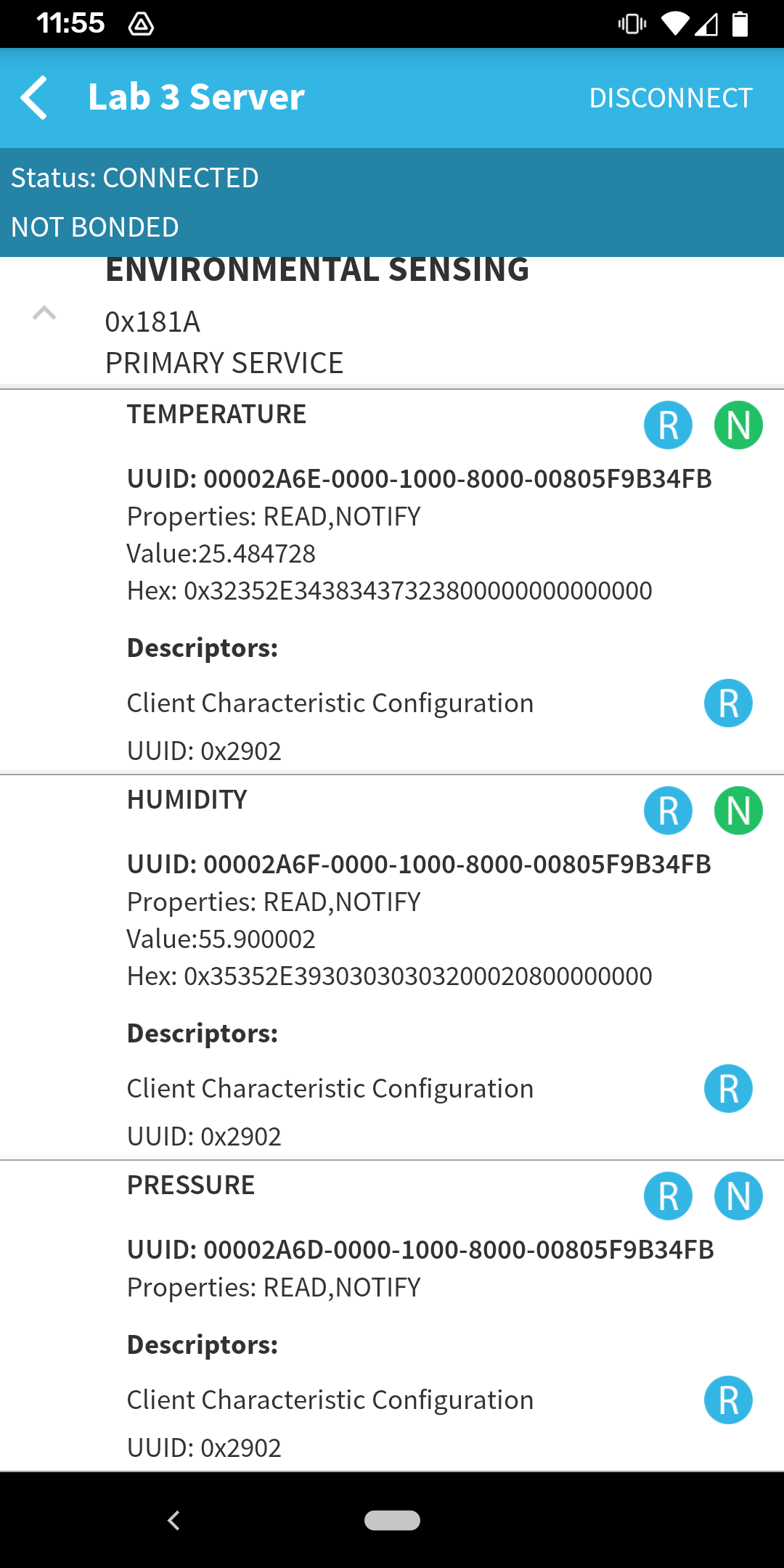
Create the following function in your Lab3Server class:

| void updateSensors(BLE &ble) {  printf("Updating sensors...\n");  temperature\_.updateValue(ble, BSP\_TSENSOR\_ReadTemp());  humidity\_.updateValue(ble, BSP\_HSENSOR\_ReadHumidity());  pressure\_.updateValue(ble, BSP\_PSENSOR\_ReadPressure());  } |
| --- |

Any time this function is called, it will call updateValue on each characteristic, with the latest reading from the sensor. To trigger this function every second, update the start method to include the following line, at the end of the routine:

| event\_queue.call\_every(1000ms, [this, &ble] { updateSensors(ble); }); |
| --- |

This tells the event queue to call the updateSensors function every second. When you connect and read the characteristic values -- or use the notify feature on the BLE scanner app -- you should see the various readings appear:



## Toggling the LED

The last piece of the puzzle is to support remote toggling of the on board LED. To achieve this, we will create a writable characteristic, i.e. one that can be written to by the client. When the value written is zero, the LED will be turned off. When the value written is one, the LED will be turned on.

The BLE API provides us with a very useful characteristic class called a ReadWriteGattCharacteristic, so we’ll use this to implement the functionality.

We need to add some additional fields to Lab3Server to keep track of things:

| DigitalOut led\_;  uint8\_t ledValue\_;  ReadWriteGattCharacteristic<uint8\_t> ledChar\_; |
| --- |

Next, we need to update the constructor to initialise them:

| Lab3Server()  : temperature\_(GattCharacteristic::UUID\_TEMPERATURE\_CHAR),  humidity\_(GattCharacteristic::UUID\_HUMIDITY\_CHAR),  pressure\_(GattCharacteristic::UUID\_PRESSURE\_CHAR), led\_(LED1),  ledValue\_(0), ledChar\_(0xA000, &ledValue\_) {} |
| --- |

Here, we initialise the LED object to point to LED1 on the board, set the initial value to zero, and initialise the characteristic with a custom identifier, 0xA000.

Now that we have the basic controls in place, we can add the characteristic to the char table list:

| GattCharacteristic \*charTable[] = {&temperature\_, &humidity\_, &pressure\_, &ledChar\_}; |
| --- |

And finally implement the method that will actually control the LED:

| virtual void onDataWritten(const GattWriteCallbackParams &params) override {  if ((params.handle == ledChar\_.getValueHandle()) && (params.len == 1)) {  printf("New LED value: %x\r\n", \*(params.data));  ledValue\_ = \*(params.data);  led\_ = ledValue\_;  }  } |
| --- |

You will notice that this method is marked “override” because it overrides a method in the base class, and is called when new data is written from a client. In the method, we check that it is intended for the LED, and if so, update the LED value and state accordingly.

When you build and run your program, you should see the new characteristic appear, and when you write 1 (01) to it (using byte representation, not text) the LED on the board will turn on. Writing a 0 (00) will turn off the LED:

