# Mobile and Ubiquitous Computing Lab Sheets 2025

# Part One

# Arduino Nano 33 BLE and Flutter

## Contents

## Introduction

This set of exercises is intended to introduce you to Arduino, Flutter and Bluetooth. You will work through these in your group to familiarise yourself with the basic concepts. You are expected to conduct additional research into Arduino, Flutter, and Bluetooth so that your group can develop a functional prototype for this module.

We expect the functional prototype to be able to do the following:

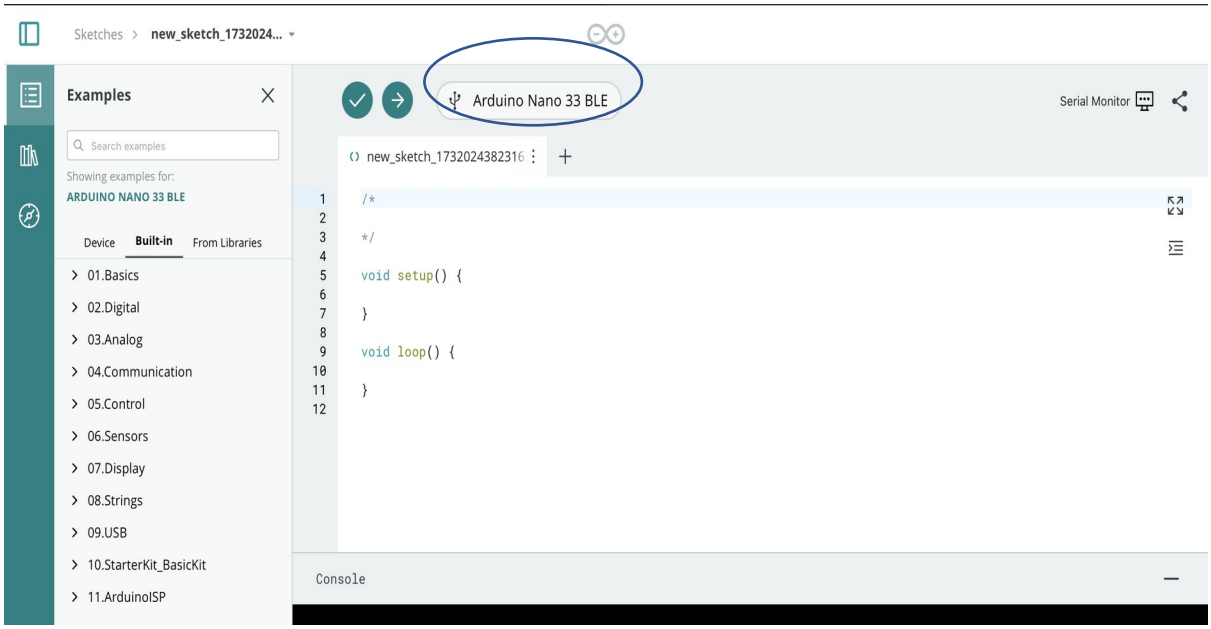
1. Capture accelerometer data from Arduino Nano 33 BLE.
2. Classify specific activities using these accelerometer data using TinyML (Note: this will be covered in Lab Sheets Part II).
3. Connect the Arduino Nano 33 BLE to a phone using Bluetooth and display sampled accelerometer values. We suggest that you sample the accelerometer values rather than stream these.
4. Connect the Arduino Nano 33 BLE to a phone using Bluetooth and display the recognised activity from TinyML. It would be good to include the activity and a confidence level (Note: this will be covered in Lab Sheets Part II).

We will provide more details on the requirements for the group activity later in the module. The expectation is that your group will develop a functional prototype to address points 1 to 4 above. The functional prototype will be reported using video to demonstrate how the prototype operates and to show it functioning. Your group will also upload Arduino, Flutter, TinyML and other code developed for the prototype.

We will provide each group with an Arduino Nano 33 BLE board. We suggest that each group uses an older model of an Android phone because these have developer options and security settings that are easier to work with. If you do not have access to an Android phone, then an older iPhone could also work. We provide some instructions on how to set Bluetooth permissions on an iPhone, but this might require additional research by your group.

## Exercise 1: Working with Arduino

You can create Arduino applications using the Integrated Development Environment (which can be downloaded from here: https://www.arduino.cc/en/software) or the Web Editor on Arduino Cloud. If you are using Mac Chromebook, you can only use the web Editor.




The Web Editor looks like the above figure. We have created a new sketch and added a blue circle around the device we are using. If you don't see this, plug the Arduino into your USB port and click on the button to discover it. To check that things are working, open '02. Digital' on the left of the screen, and then select '( ) BlinkWithoutDelay'. This will open a sketch in the main area of the screen. Click the arrow key (in the blue circle, next to the device button). This will load the code onto your Arduino (it will take a few seconds to complete). Once this has loaded, you will see the orange LED on the board (to the left of the USB input) flashing on and off.

We are using the Arduino Nano 33 BLE board for these labs. This is a small board containing a NINA B306 module, based on Nordic nRF52480 and containing an Arm® Cortex®-M4F and a 9-axis IMU. As the board diagram shows, this comes with pins that can be used for digital (D) or analogue (A) input. Note, the board can ONLY run on 3.3V (so ignore the pin labelled 5V). More detail can be found in the datasheet here: https://docs.arduino.cc/resources/datasheets/ABX00030-datasheet.pdf.

## Exercise 2: Reading the onboard Accelerometer on Arduino Nano 33 BLE

The datasheet tells us that the Inertial Measurement Unit (IMU) on the Nano 33 BLE is the LSM9DS1. We can install a library to manage this unit on the board.

On the web Editor, click on the library (books) icon. Then type LSMD9S1. Include this library.

In the Arduino IDE, go to ‘Tools’ > ‘Manage Libraries’ and type in LSM9DS1. Install this library.

On the Web Editor, click on the Sketch icon, go to the ‘From Libraries’ tab, and type in LSMD. Note: there is another library with LSM9DS1 in its name, but this is for the Adafruit device family. Expand the ‘ArduinoLSM9DS1 Library’ and select the code for ‘() SimpleAccelerometer’

In the ArduinoIDE will find sample code to use in ‘Files’ > ‘Examples’ > ‘Arduino – LSM9DS1’ > ‘Simple Accelerometer’.

Open this code and install it in on your board. Click on the blue circle with the arrow. Here is the complete code listing from the library.

```
/*
  Arduino LSM9DS1 - Simple Accelerometer

  This example reads the acceleration values from the LSM9DS1
  sensor and continuously prints them to the Serial Monitor
  or Serial Plotter.

  The circuit:
  - Arduino Nano 33 BLE Sense

  created 10 Jul 2019
  by Riccardo Rizzo

  This example code is in the public domain.
*/

#include <Arduino_LSM9DS1.h>

void setup() {
  Serial.begin(9600);
  while (!Serial);
  Serial.println("Started");

  if (!IMU.begin()) {
    Serial.println("Failed to initialize IMU!");
    while (1);
  }

  Serial.print("Accelerometer sample rate = ");
  Serial.print(IMU.accelerationSampleRate());
  Serial.println(" Hz");
  Serial.println();
  Serial.println("Acceleration in g's");
  Serial.println("X\tY\tZ");
}
```

In ‘void setup’, the board sets the rate at which data is exchanged over a serial port. The baud rate is in bits per second and the serial connection is via Universal Serial Bus (USB). Once a connection is established between the board and USB, a message is sent to say ‘Started’. Following this, a message is sent to indicate the Accelerometer sample rate, in Hertz.

```
void loop() {
  float x, y, z;

  if (IMU.accelerationAvailable()) {
    IMU.readAcceleration(x, y, z);

    Serial.print(x);
    Serial.print('\t');
    Serial.print(y);
    Serial.print('\t');
    Serial.println(z);
  }
}
```

In ‘void loop’, the board writes to the serial port a value for each accelerometer axis.

Once the program has compiled, installed and indexed, open the Serial Monitor (from ‘Tools’) and you should see rapidly scrolling list of x, y, z values from the accelerometer.

If you pick up the board and tilt it 90 degrees, the values in the list will change. You could change the delay – go to line 48 and between the two }, type the command:

delay (2000);

Then reload the code and open the Serial Monitor (this is the button on the top right of the Web Editor and opens a new browser tab, or is found under ‘Tools’ in the IDE). On the IDE, you should see, with a 2 second update:

```
Started
Accelerometer sample rate = 119.00 Hz

Acceleration in g's
X       Y       Z
-0.09   -0.84   0.53
-0.09   -0.84   0.53
-0.10   -0.83   0.53
-0.09   -0.83   0.54
-0.09   -0.84   0.53
-0.09   -0.83   0.53
-0.09   -0.84   0.53
-0.09   -0.84   0.53
-0.09   -0.84   0.53
```

The Serial Monitor on the Web Editor has 3 columns scrolling (for X, Y, Z).

As you move the accelerometer, so the values reported will change.

## Exercise 3: Interpreting the Accelerometer Values

Next, we might want the results to be a little more intuitive. So, we will replace the x,y,z readings with words describing the orientation of the board, e.g., pointing up, pointing down, etc.

This requires the x, y, z values to be interpreted relative to their change in orientation. So, we initialise variables for these.

Before the void setup, add the following lines:

```
#define MIN_TILT 5 //minimum change in orientation for reading
#define DELAY 500 //sample every 500ms

float x, y, z;
int angleX = 0;
int angleY = 0;
unsigned long previousTime = 0;
```

The void loop is modified to define the orientation of the accelerometer. First, remove the first line in the void loop, i.e., float x,y,z as this is now moved to the top of the code. In this code, if the values do not exceed the MIN_TILT then there is no reading on the serial monitor.

```
void loop() {
  unsigned long currentTime = millis();

  if (IMU.accelerationAvailable() && (currentTime - previousTime >= DELAY)) {
    previousTime = currentTime;

    IMU.readAcceleration(x, y, z);

    //Calculate orientation in degrees

  angleX = atan2(x, sqrt(y * y + z * z)) * 180 / PI;
  angleY = atan2(y, sqrt(x * x + z * z)) * 180 / PI;

    // Determine the orientation of the board based on angleX and angleY

  if (angleX > MIN_TILT) {  //  up
     Serial.print("Pointing up ");
    Serial.print(angleX);
          Serial.println(" degrees");
    } else if (angleX < -MIN_TILT) {  //  down
      Serial.print("Pointing down ");
      Serial.print(-angleX);
      Serial.println(" degrees");
    }

    if (angleY > MIN_TILT) {  //  left
      Serial.print("Pointing left ");
      Serial.print(angleY);
      Serial.println(" degrees");
    } else if (angleY < -MIN_TILT) {  //  right
      Serial.print("Pointing right ");
     Serial.print(-angleY);
          Serial.println(" degrees");
               }
    }
  }
```

This code is very slightly modified from the example provided in:

https://docs.arduino.cc/tutorials/nano-33-ble/imu-accelerometer

You will notice that the readings require the Arduino orientation to be fairly close to up, down, left, or right – where you have the board tilted between these, you will have readings that oscillate between up and left, for example.

A further change to the code could include an action to perform if the board is in a specific orientation. For example, the onboard LED (pin 13) could be turned on when the board is pointing 'up' and turned off in any other orientation.

In this case, declare the pin for the LED using the following statement (insert after the #define lines):

const int PIN = 13;

and then, after each Serial.println(" degrees"); and digitalWrite(PIN, LOW) - but use HIGH for the pointing up function.

From this, the onboard (orange) LED will turn on only when the board is pointing up.

You could imagine that, rather than merely turning the LED on, the function could be used to drive a motor or other actuator. You could look at the 'servo' or 'stepper' code examples in the libraries to see how these motors are controlled.

Question: How might you use the accelerometer data to count steps when walking or running?

## Exercise 4: Working with Flutter

Flutter is an open-source framework that Google have developed to allow you to compile apps to different platforms from a single code.

In these labs we provide advice on using Flutter from the IntelliJ integrated development environment, although the Flutter code ought to work equally well from Android Studio, XCode, or Visual Studio Code.

Install Flutter from here: https://docs.flutter.dev/get-started/install. This will save to your Downloads folder. It is a good idea to move this into another folder (but not the Desktop), so I suggest putting this into your Documents folder (unless you want to create a new folder for it).

- Assuming that you already have IntelliJ IDEA Community edition, which is free from www.jetbrains.com, installed on your computer.
- Open IntelliJ IDEA and, on the left panel, select 'Plugins'.
- Open 'Marketplace' and type flutter into the search bar.
- Select the Flutter plugin and click install.
- Click Yes when prompted to begin install and then restart (if required). This installation should also include the Dart language that flutter uses.

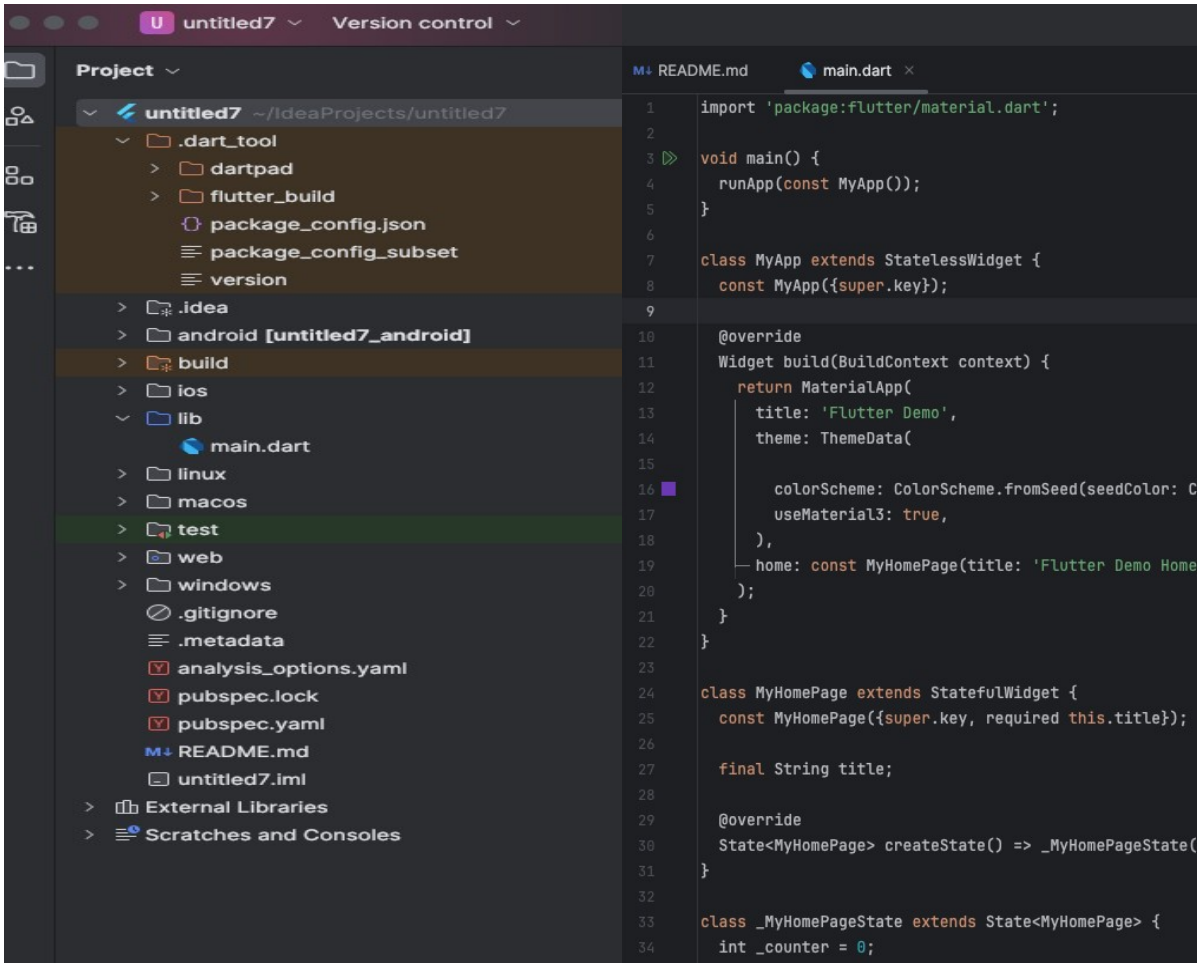
The same process (i.e., obtaining flutter from marketplace) should also work with Android Studio.

First Steps
We will use the demo app, which looks like this:


From the IntelliJ home page, select ‘New Project’.

Make sure that you have selected ‘Flutter’ on the list of Generators (on the left panel).
This will take you to a page that requires you to specific the Flutter SDK path. This will be the path to Flutter (e.g., in your Documents). You can, on Mac, get this by opening Documents, finding the Flutter folder, and right-clicking on this to find ‘Get info’, and then copy the path from ‘Where:’

Give the project a name and click ‘Next’. This will take a couple of minutes to open (don’t panic) and the screen will show the Project page with a ‘README.md’ and ‘main.dart’ panels. If you don’t see a ‘main.dart’ panel, scroll down to ‘lib’ (on the left panel) and expand this.




The code, in ‘main.dart’ builds the app from the ‘material.dart’ package by defining a BuildContext (from line 45 onwards) that allow a counter (line 33) to increment each time the + button (line 68) is clicked. This code combines a set of widgets (which is the term Flutter and Dart use to refer to objects) that are organised in a widget tree.

A brief explanation of this code:

- All Flutter apps make use of the ‘Material’ style-guide. This contains many User Interface design solutions (in the form of ‘widgets’) that influence layout, e.g., how the screen can be divided into columns and rows, how containers can be positioned on the screen, how text can be presented etc. To make use of this style guide we first import the Material package. There is a related package that has additional elements for iOS (called ‘cupertino.dart’) but Material works on iphones so we will use this as the basic package for these labs.
- All widgets have their own build method that is defined in a BuildContext. This could be a default ‘context’ or relate to a specific ‘theme’.
- The void main ( ) defines the code that is to be run. You can see here (and in the later examples) how this code feels familiar to other languages that you might know – and the developers at Google have combined elements from a few languages into Dart. In this example, the code call a class called ‘MyApp’.
- The class ‘MyApp’ is the application that we are writing. It draws on the StatelessWidget definition in Material. Widgets can be ‘stateless’ (in that they do not

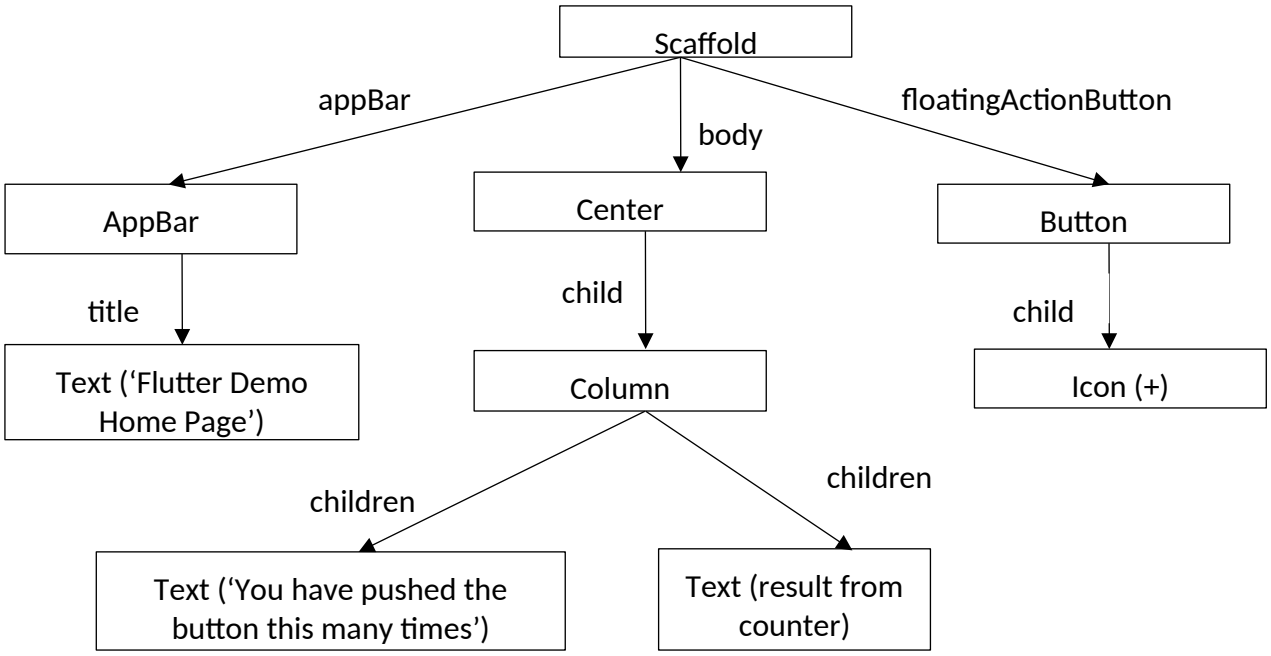
change at run time) or ‘stateful’ (in that they can be updated or modified at run time). Each Stateful Widget has a State object to maintain it.

- A Stateless Widget is immutable; once created, it cannot be modified, e.g., icons or text. Stateless Widgets extend generic StatelessWidget classes and have the method in the widget:

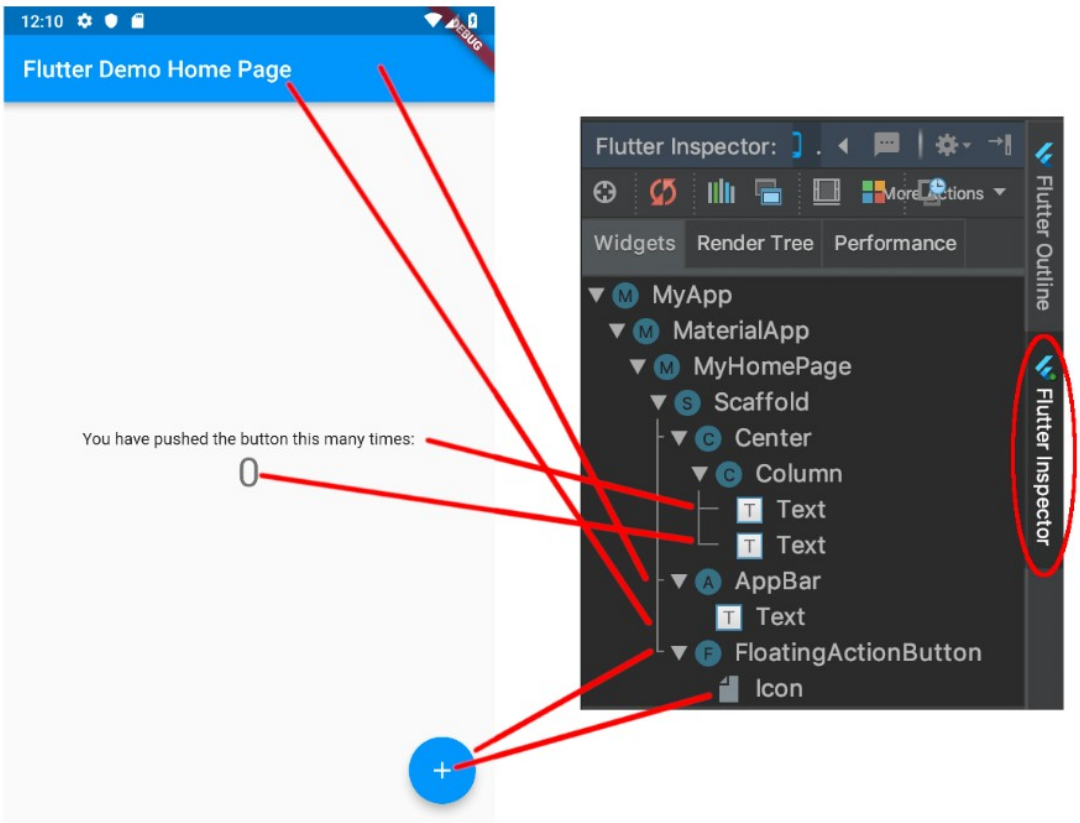
```
class StaticWidget extends StatelessWidget {
@override
Widget build(BuildContent context) {
return MaterialApp(
home: StateWidget (),
    );
  }
}
```

- The demo is are using the functionality of Material’s StatelessWidget for the layout.
- For the counter, we want to modify the content, so this uses a Stateful widget.
- The root of the app is the Scaffold (which is the basic structure of our User Interface) on line 46. One useful feature of a Scaffold is that it adapts to the device parameters on which the UI will be displayed and render the content differently according to these parameters.
- The AppBar is the title bar for your app and will be placed at the top of the UI. We can modify it position but for the moment it will appear in the top left.
- Below the AppBar we define a column that will have text centred on the UI.

For this demo app, the widget tree can be drawn like this:

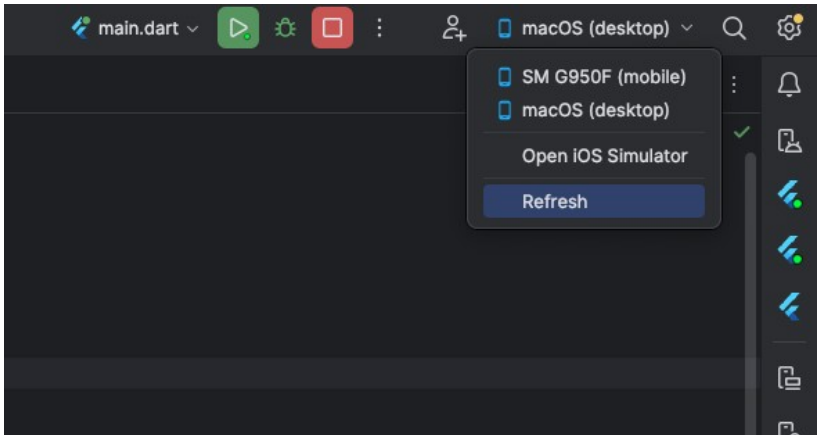



You can also use Flutter Inspector to create a hierarchical text listing of the code in the form of a widget tree:




From the drop-down menu on the top-right of the screen (above the 'main.dart' panel) to select the destination for the app, select 'macOS (desktop)' (on a Windows laptop the option

will be ‘Chrome(web)’) and then click the green arrow to compile and load the code to a desktop screen:




Connect your phone to the USB port. With the phone in debug mode (see Appendix 1), this should appear on the options. If the above figure, I have connected a Samsung Galaxy 9. You might need to turn the debug mode on and off in order for the phone to appear in the drop-down menu. Once the phone is available, click the green arrow and the app will install on your phone. It will appear as icon with the Flutter logo and the project name. Open this and the app will be running on your phone.

## Exercise 5: Basic Flutter layouts

In this lab, we will look at the arrangement of Widgets on a screen. Figure 1 shows the final design, running on an iOS emulator (using XCode on a Mac)


In terms of layout, this has:

1. A title at the top of the screen
2. An image that is in a single column…
3. A block of header text – in a single column…
4. A block of filler text – wrapped in a single column…
5. Three buttons, spaced in a row…

Question: Sketch the Widget tree for this layout. You can check this later by using Flutter Inspector from the completed code.

Create a new Flutter project. This will open with a default script in main.dart (the button pressing example described in the previous exercise). I find it easier to delete this and start from a blank page, even though we will be reusing some of the current commands.

Create the basic app by giving it a name, a title and begin the layout as a stateless widget:

```
import 'package:flutter/material.dart';


        void main() {
                runApp(const MyApp());
        }

        class MyApp extends StatelessWidget {
          const MyApp({super.key});

          @override

         Widget build(BuildContext context) {
                return MaterialApp(
                        title: ‘Layout demo’,
                                home: Scaffold(
                                        appBar:AppBar(
                                        title: Text(‘Welcome to the School of Computer
                                        Science’),
                `               ), //AppBar
                                Body: Center( ),
                                 ), //Scaffold
                ); //MaterialApp
        }
}
```

Next, we are going to develop the Body of the layout. Inside the brackets ( ) after Body: Center add the following:

```
child: Column(
children: [
Container(
padding: EdgeInsets.all(32),
child: Column(
crossAxisAlignment: CrossAxisAlignment.start,
children: [
Text(
University of Birmingham',
style: TextStyle(fontWeight: FontWeight.bold),
),
```

```
Text(
'Birmingham, UK',
style: TextStyle(color: Colors.grey[500]),
),
],
),
)
],
)
),
),
);
}
}
```

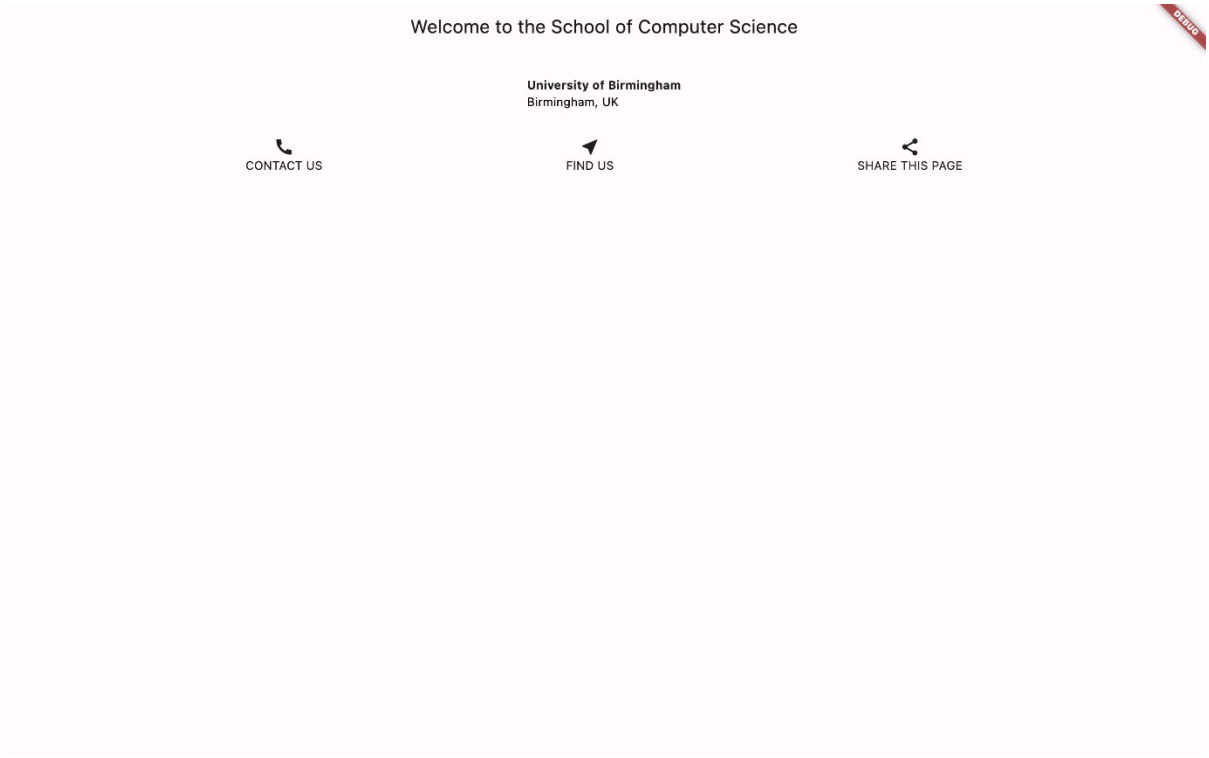
This adds a Container which is indented from the screen edge and which arranges in contents in a column.  In this case, the container has two children, which are two lines of text.

Next, we are going to add the three action buttons in a row (although in this exercise we are not assigning any functionality to the buttons)

```
Row(
                mainAxisAlignment: MainAxisAlignment.spaceEvenly,
                children: [
                  Column(
                    children: [
                      Icon(Icons.call),
                      Text('CONTACT US'),
        ],
        ),
```

You can add additional children for buttons for find us and share this page using the Icons.near_me and Icons.share.  Hint: copy and paste from Column to ],),

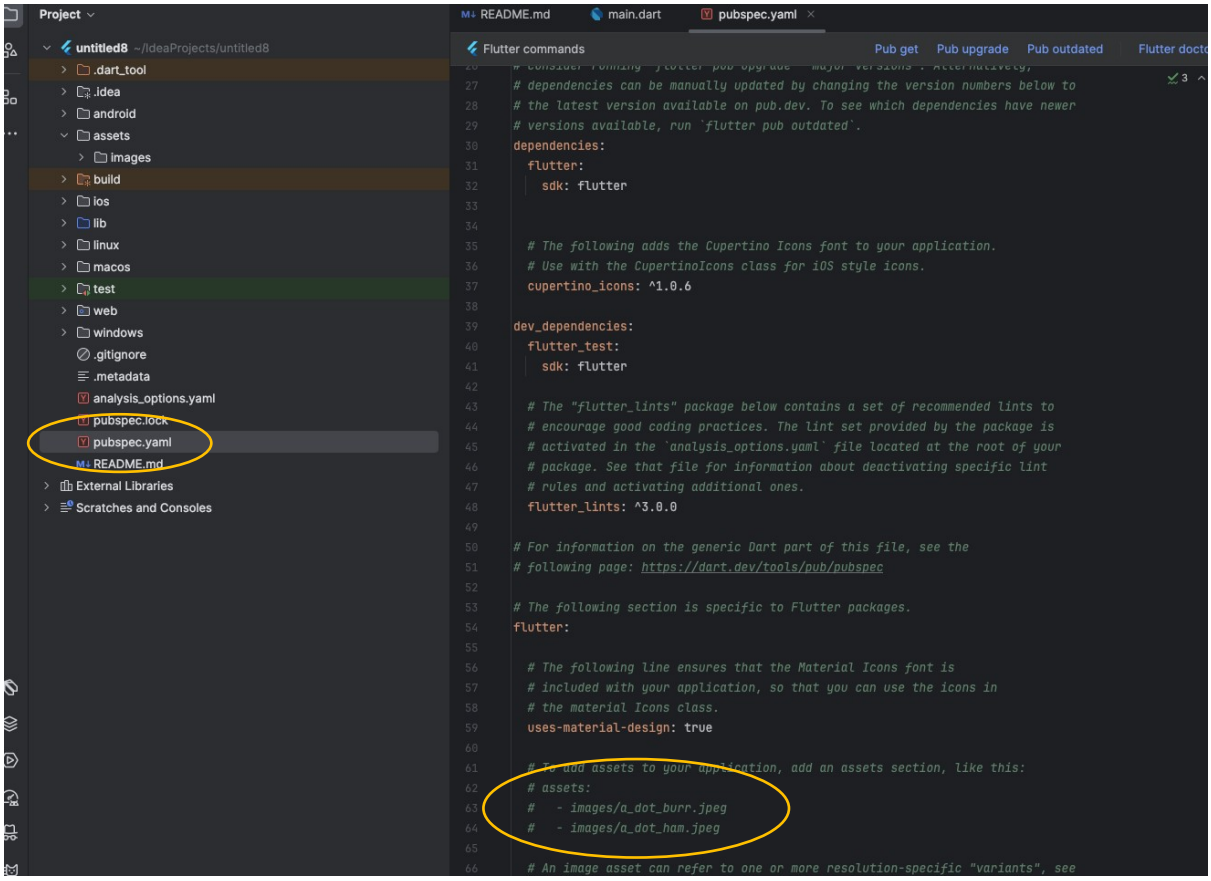
So far, if you send the page to the macOS (desktop) you should have a screen that looks like this:




Next we are going to add the image.  To add an Image, you need to put this into the root directory of the project.
Click on the project name and select new > Directory from the File / New menu
Call this ‘assets’ and inside this, add another directory, called ‘images’.
I copied the image from here: https://www.birmingham.ac.uk/schools/computer-science/visit%20us and saved it to my downloads.  You might want to edit the image prior to adding it to your project.

Drag the image into your ‘images’ directory and agree that this can be refactored.

Open the pubspec.yaml (towards to the lower part of the left panel)




i. Click on ‘assets’ and backspace until the text is orange. This should be 2 whitespaces from the border.
ii. Right-click on your image, and copy the path (you might need to copy the absolute path).
iii. Paste the image path - this needs to be 4 whitespaces from the edge of the screen
iv. Click Pub upgrade
v. In the body of the code, update the reference to the image (line 91) with the path to your image.
vi. Click Get Dependencies.
vii. Upload the project to the macOS(desktop) or Chrome(web) or to your phone.

Add the following code (after the widget for the Text section):

```
Row (
        mainAxisAlignment: MainAxisAlignment.spaceEvenly,
children: [
Image.asset(paste absolute image path,
width: 300,
height: 150,)
],
),
```

Finally, we add the block of text. After the button row, add this:

```
Padding(
Padding: EdgeInsets.all(20.),
Child: Container(
Width: 600,
Child:
Text('The School of Computer Science at the University of Birmingham was ranked 3rd in the 2022 UK REF.' 'It offers Undergraduate and Postgraduate programmes across a range of topics including AI and data science, computer security, and human-computer interaction.',
softWrap: true),
),
),
```

## Exercise 6: Creating a Radial Display in Flutter

In this example, we will build a radial display using the ‘sleek circular slider’ library in Flutter. The reason for adding this is that you could use this display to show the sampled accelerometer values when you connect the Arduino to the phone.

First add and depend on the library:

```
dependencies:
  sleek_circular_slider: ^2.0.1
```

Next import the library and define the layout:

```
import 'package:flutter/material.dart';
import 'package:sleek_circular_slider/sleek_circular_slider.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      // Define the app's theme
      theme: ThemeData(
        primarySwatch: Colors.green, // Set the app's primary theme color
      ),
      debugShowCheckedModeBanner: false, // Remove debug banner
      home: CircularSliderWidget(),
    );
  }
}
```

Then we will define the ‘circular slider’ as a Class. This defines the layout of the screen and inserts a circular display that can be interacted with by moving it with your finger.

```
class CircularSliderWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Sleek Circular Slider Example'),
      ),
      body: Center(
        child: SleekCircularSlider(
          initialValue: 50, // Initial value
          max: 100, // Maximum value
          appearance: CircularSliderAppearance(
            customColors: CustomSliderColors(
```

```
            progressBarColors: [Colors.blue], // Customize progress bar colors
            trackColor: Colors.grey, // Customize track color
            shadowColor: Colors.green, // Customize shadow color
            shadowMaxOpacity: 0.2, // Set shadow maximum opacity
          ),
          customWidths: CustomSliderWidths(
            progressBarWidth: 12, // Set progress bar width
            trackWidth: 12, // Set track width
            shadowWidth: 20, // Set shadow width
          ),
          size: 200, // Set the slider's size
          startAngle: 150, // Set the starting angle
          angleRange: 240, // Set the angle range
          infoProperties: InfoProperties(
            // Customize label style
            mainLabelStyle: TextStyle(fontSize: 24, color: Colors.blue),
            modifier: (double value) {
              // Display value as a percentage
              return '${value.toStringAsFixed(0)}%';
            },
          ),
          spinnerMode: false, // Disable spinner mode
          animationEnabled: true, // Enable animation
        ),
        onChange: (double value) {
          // Handle value change here
        },
      ),
    ),
  );
 }
}
```

## Exercise 7: Using BLE with Arduino

For this, you need to find the BLE library for Arduino. This contains several code examples in the ArduinoBLE library. We will be using the Peripheral libraries under this.

In BLE, devices are either Central (server) or Peripheral (client). That is, rather than one-to-one UART connections in traditional Bluetooth, BLE has peripheral devices that can publish data and clients that can read the published data.

The example code begins by calling the library required (i.e., ArduinoBLE) and then creates a service, which is called 'ledService' and has a 16-bit identifier. Within this service, there are characteristics which can be read or written. These also have 16-bit identifiers.

The devices provide Service (set of capabilities). Each service has a Unique Identification code (UUID). By convention, UUIDs are 128-bit values. For example, the 'led service' UUID is: "19B10010-E8F2-537E-4F6C-D104768A1214". You will recognise that this is written in hexadecimal. A full set of identifiers can be found here:

https://www.bluetooth.com/wp-content/uploads/Files/Specification/HTML/Assigned_Numbers/out/en/Assigned_Numbers.pdf?v=1732021426065.

Some apps, such as the 'battery monitor' we will look at in the next exercise, use a 16-bit version is easier to write. The relation between the two is: 128-bit value = 16-bit value * 2^96 + BluetoothBaseUUID. The BluetoothBaseUUID is 00000000-0000-1000-8000-00805F9B34FB. So, removing this from the 'battery service' gives 0000180F. If you look at section 3.4 of the pdf from this link, you can see a table of Services by Name. The 'battery service' has a UUID of 0x180F. Each service contains characteristics, which is the device's capabilities. In section 3.8 of the pdf, there is a table of Characteristics. Here, the 'battery level' is characteristic 0x2A19. In BLE, the General Attribute Profile (GATT) allows peripherals to have multiple services, where each service can consist of multiple characteristics. The Central device will read (or write or be notified by) the services to which it subscribes.

In the Arduino code, the void setup defines the baud rate (as we did in the accelerometer example) and defines which pins are to be used by the app. In this version, the on-board LED (connected to pin 13) will be turned off or on, and a button can be connected to pin 4 to toggle the LED status. Following this, the initialization routine checks that a BLE connection has been established. You will need to open the Serial Monitor to check this.

```
*
  Button LED

  This example creates a Bluetooth® Low Energy peripheral with service that contains a
  characteristic to control an LED and another characteristic that
  represents the state of the button.

  The circuit:
  - Arduino MKR WiFi 1010, Arduino Uno WiFi Rev2 board, Arduino Nano 33 IoT,
    Arduino Nano 33 BLE, or Arduino Nano 33 BLE Sense board.
  - Button connected to pin 4

  You can use a generic Bluetooth® Low Energy central app, like LightBlue (iOS and Android) or
  nRF Connect (Android), to interact with the services and characteristics
  created in this sketch.
```

```
  This example code is in the public domain.
*/

#include <ArduinoBLE.h>

const int ledPin = LED_BUILTIN; // set ledPin to on-board LED
const int buttonPin = 4; // set buttonPin to digital pin 4

BLEService ledService("19B10010-E8F2-537E-4F6C-D104768A1214"); // create service

// create switch characteristic and allow remote device to read and write
BLEByteCharacteristic ledCharacteristic("19B10011-E8F2-537E-4F6C-D104768A1214", BLERead |
BLEWrite);
// create button characteristic and allow remote device to get notifications
BLEByteCharacteristic buttonCharacteristic("19B10012-E8F2-537E-4F6C-D104768A1214", BLERead
| BLENotify);

void setup() {
  Serial.begin(9600);
  while (!Serial);

  pinMode(ledPin, OUTPUT); // use the LED as an output
  pinMode(buttonPin, INPUT); // use button pin as an input

  // begin initialization
  if (!BLE.begin()) {
    Serial.println("starting Bluetooth® Low Energy module failed!");

    while (1);
  }

  // set the local name peripheral advertises
  BLE.setLocalName("ButtonLED");
  // set the UUID for the service this peripheral advertises:
  BLE.setAdvertisedService(ledService);

  // add the characteristics to the service
  ledService.addCharacteristic(ledCharacteristic);
  ledService.addCharacteristic(buttonCharacteristic);

  // add the service
  BLE.addService(ledService);

  ledCharacteristic.writeValue(0);
  buttonCharacteristic.writeValue(0);

  // start advertising
  BLE.advertise();

  Serial.println("Bluetooth® device active, waiting for connections...");
}

void loop() {
  // poll for Bluetooth® Low Energy events
  BLE.poll();

  // read the current button pin state
  char buttonValue = digitalRead(buttonPin);

  // has the value changed since the last read
  bool buttonChanged = (buttonCharacteristic.value() != buttonValue);

  if (buttonChanged) {
    // button state changed, update characteristics
    ledCharacteristic.writeValue(buttonValue);
```

```
    buttonCharacteristic.writeValue(buttonValue);
  }

  if (ledCharacteristic.written() || buttonChanged) {
    // update LED, either central has written to characteristic or button state has changed
    if (ledCharacteristic.value()) {
      Serial.println("LED on");
      digitalWrite(ledPin, HIGH);
    } else {
      Serial.println("LED off");
      digitalWrite(ledPin, LOW);
    }
  }
}
```

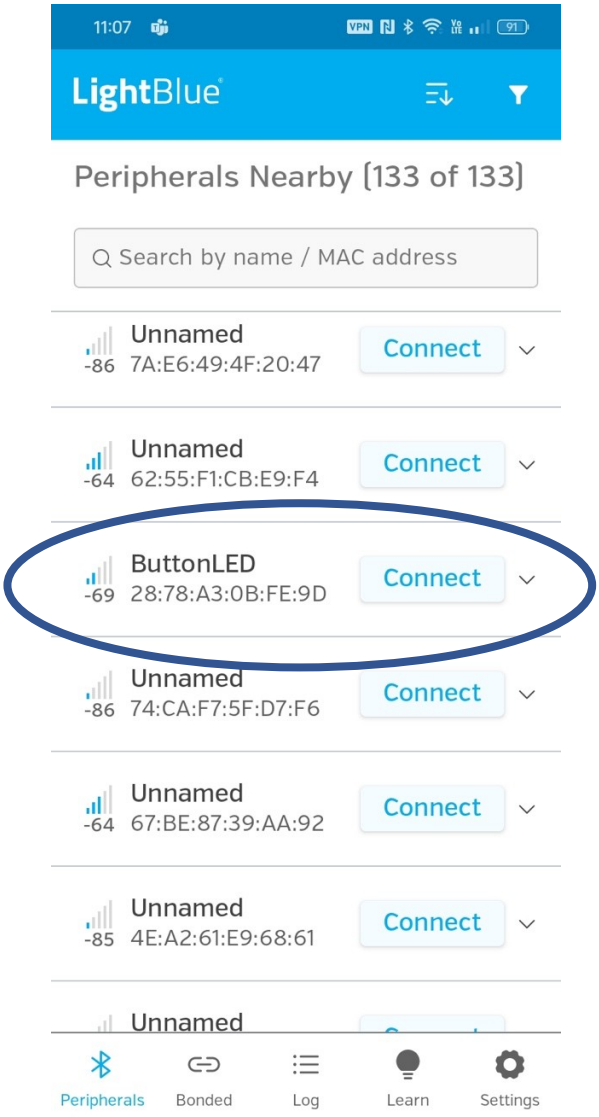
The code then defines the Arduino board as a peripheral device with the name 'ButtonLED' and sets this to advertise the ledService that had previously been identified. This means that any client device will be able to use this service and modify the characteristics, once connected.

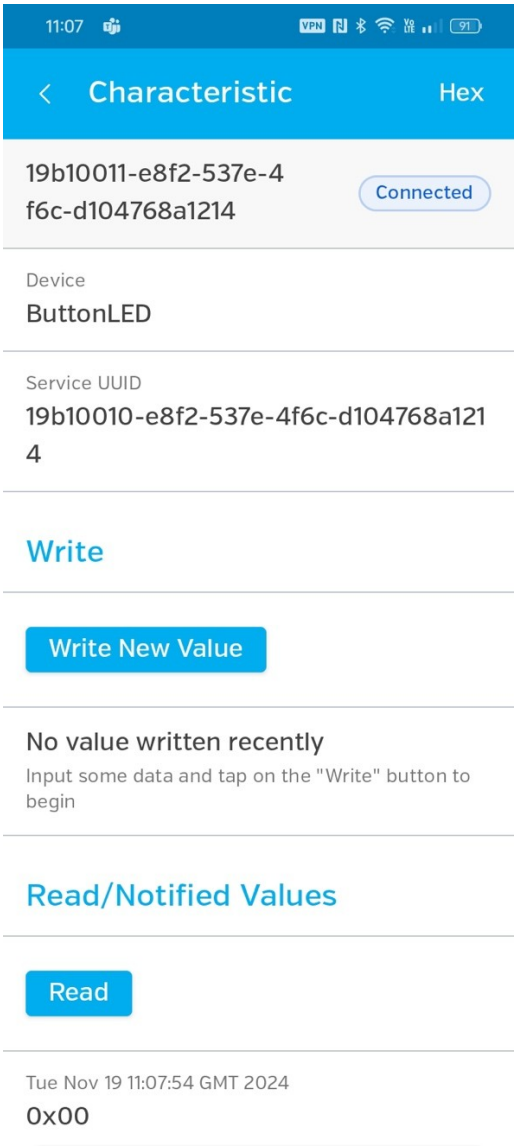
Open 'Serial Monitor' (from the Tools menu) and the prompt will say 'Bluetooth device active, waiting for connections…'

The void loop begins by polling for possible clients. To establish a connection, install the LightBlue app on your phone. You can find that in most app stores.

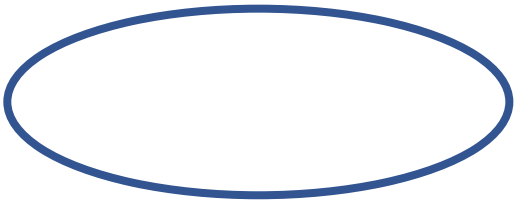
Running LightBlue will allow you to establish a connection with the app. The Serial Monitor will now showing scrolling read-out of the Battery Level %. The LightBlue app pages look like the following:

1. In a list of devices that have been discovered by the LightBlue app. Scroll these until you find the device name we are looking for 'BatteryMonitor'. Click on connect.
2. Scroll to 'Battery Service' and click on 'Properties: Read, Notify'
3. Click on 'Read'. This will present the reading of the battery level at the last update prior to you pressing 'read'.


11:07
LightBlue
Peripherals Nearby (133 of 133)
Search by name / MAC address
Unnamed
-86 7A:E6:49:4F:20:47
Connect
Unnamed
-64 62:55:F1:CB:E9:F4
Connect
ButtonLED
-69 28:78:A3:0B:FE:9D
Connect
Unnamed
-86 74:CA:F7:5F:D7:F6
Connect
Unnamed
-64 67:BE:87:39:AA:92
Connect
Unnamed
-85 4E:A2:61:E9:68:61
Connect
Unnamed
Peripherals
Bonded
Log
Learn
Settings


11:07
Characteristic
Hex
19b10011-e8f2-537e-4f6c-d104768a1214
Connected
Device
ButtonLED
Service UUID
19b10010-e8f2-537e-4f6c-d104768a1214
Write
Write New Value
No value written recently
Input some data and tap on the "Write" button to begin
Read/Notified Values
Read
Tue Nov 19 11:07:54 GMT 2024
0x00

The update rate is quite high and we can reduce this simply by adding a ‘delay(2000);’ command before the final } in the battery monitor code. You’ll notice that the update (on the Serial Monitor) is about once a second (but the timing is not consistent).



## Exercise 8: Flutter app to discover Bluetooth devices

For this exercise, the Arduino is the peripheral (with name ‘BatteryMonitor’) and will broadcast the Battery Level using the service UUID ‘180F’ and characteristic ‘2A19’. The Flutter app will connect to a device with this name and use this service. The output is a value in the centre of the phone screen that changes as the battery level reading is updated.

‘BatteryMonitor’ (ArduinoIDE: File > Examples > ArduinoBLE > Peripheral > BatteryMonitor) code is listed below. As with the ButtonLED example, the code defines service and characteristics. In this example, updates are acquired automatically (on the basis of sampling at time intervals).

```
/*
  Battery Monitor

  This example creates a Bluetooth® Low Energy peripheral with the standard battery service and
  level characteristic. The A0 pin is used to calculate the battery level.

  The circuit:
  - Arduino MKR WiFi 1010, Arduino Uno WiFi Rev2 board, Arduino Nano 33 IoT,
    Arduino Nano 33 BLE, or Arduino Nano 33 BLE Sense board.

  You can use a generic Bluetooth® Low Energy central app, like LightBlue (iOS and Android) or
  nRF Connect (Android), to interact with the services and characteristics
  created in this sketch.

  This example code is in the public domain.
*/

#include <ArduinoBLE.h>

 // Bluetooth® Low Energy Battery Service
BLEService batteryService("180F");

// Bluetooth® Low Energy Battery Level Characteristic
BLEUnsignedCharCharacteristic batteryLevelChar("2A19",  // standard 16-bit characteristic UUID
    BLERead | BLENotify); // remote clients will be able to get notifications if this characteristic
changes

int oldBatteryLevel = 0;  // last battery level reading from analog input
long previousMillis = 0;  // last time the battery level was checked, in ms

void setup() {
  Serial.begin(9600);    // initialize serial communication
  while (!Serial);

  pinMode(LED_BUILTIN, OUTPUT); // initialize the built-in LED pin to indicate when a central is
connected

  // begin initialization
  if (!BLE.begin()) {
    Serial.println("starting BLE failed!");

    while (1);
  }

  /* Set a local name for the Bluetooth® Low Energy device
     This name will appear in advertising packets
     and can be used by remote devices to identify this Bluetooth® Low Energy device
     The name can be changed but maybe be truncated based on space left in advertisement packet
  */
  BLE.setLocalName("BatteryMonitor");
```

```
  BLE.setAdvertisedService(batteryService); // add the service UUID
  batteryService.addCharacteristic(batteryLevelChar); // add the battery level characteristic
  BLE.addService(batteryService); // Add the battery service
  batteryLevelChar.writeValue(oldBatteryLevel); // set initial value for this characteristic

  /* Start advertising Bluetooth® Low Energy.  It will start continuously transmitting Bluetooth®
Low Energy
     advertising packets and will be visible to remote Bluetooth® Low Energy central devices
     until it receives a new connection */

  // start advertising
  BLE.advertise();

  Serial.println("Bluetooth® device active, waiting for connections...");
}

void loop() {
  // wait for a Bluetooth® Low Energy central
  BLEDevice central = BLE.central();

  // if a central is connected to the peripheral:
  if (central) {
    Serial.print("Connected to central: ");
    // print the central's BT address:
    Serial.println(central.address());
    // turn on the LED to indicate the connection:
    digitalWrite(LED_BUILTIN, HIGH);

    // check the battery level every 200ms
    // while the central is connected:
    while (central.connected()) {
      long currentMillis = millis();
      // if 200ms have passed, check the battery level:
      if (currentMillis - previousMillis >= 200) {
        previousMillis = currentMillis;
        updateBatteryLevel();
      }
    }
    // when the central disconnects, turn off the LED:
    digitalWrite(LED_BUILTIN, LOW);
    Serial.print("Disconnected from central: ");
    Serial.println(central.address());
  }
}

void updateBatteryLevel() {
  /* Read the current voltage level on the A0 analog input pin.
     This is used here to simulate the charge level of a battery.
  */
  int battery = analogRead(A0);
  int batteryLevel = map(battery, 0, 1023, 0, 100);

  if (batteryLevel != oldBatteryLevel) {      // if the battery level has changed
    Serial.print("Battery Level % is now: "); // print it
    Serial.println(batteryLevel);
    batteryLevelChar.writeValue(batteryLevel);  // and update the battery level characteristic
    oldBatteryLevel = batteryLevel;           // save the level for next comparison
  }
}
```

This exercise is using the 'flutter blue' library, and the example is taken from Kevin Bernard's 2020 Medium post here: https://medium.com/@kvinblondelbernard/how-to-communicate-through-ble-using-flutter-b3d299308d85

Flutter does not natively support Bluetooth so you need to work with additional libraries. Not all of these are well maintained, so you might need to hunt for ones that are up to date. An alternative is flutter_reactive (see: ) or flutter_blue_plus (see: ).

To work with any additional library, you need (as a minimum) to update the pubspecyaml, to modify permissions in the AndroidManifest, and to define the minimum sdk version in the build.gradle files. You can, from this brief description, appreciate that Flutter combines several ?? and ensuring that update is consistent across all of these can be challenging.

Create a new Flutter project. Let's call it 'flutterscan'. Ensure that you are writing this using kotlin and gradle (and not Java).

Update the dependencies in flutterscan / pubspecyaml (which is towards the bottom of the list of files in the project.

```
dependencies:
  flutter:
    sdk: flutter

  # The following adds the Cupertino Icons font to your application.
  # Use with the CupertinoIcons class for iOS style icons.
  cupertino_icons: ^1.0.6
  flutter_blue_plus: ^1.32.1
  permission_handler: ^11.3.1
```

Update the sdk version to use in flutterscan/android/app/src/build.gradle:

```
minSdkVersion flutter.minSdkVersion = 21
```

Update permissions in flutterscan /android / app / src / main / AndroidManifest.xml:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android">

    <uses-permission android:name="android.permission.INTERNET"/>
    <uses-permission android:name="android.permission.BLUETOOTH" />
    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
    <uses-permission android:name="android.permission.BLUETOOTH_CONNECT" />
    <uses-permission android:name="android.permission.BLUETOOTH_SCAN" />
    <uses-permission android:name="android.permission.BLUETOOTH_ADVERTISE"
/>
    <uses-permission
android:name="android.permission.ACCESS_COARSE_LOCATION"/>
    <uses-permission
android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

Note: if you are developing the app to run on iOS, you need to update the settings in flutterscan / ios / Runner / Info.plist:

```
<key>NSBluetoothAlwaysUsageDescription</key>
<string>Need BLE permission</string>
```

```
<key>NSBluetoothPeripheralUsageDescription</key>
<string>Need BLE permission</string>
<key>NSLocationAlwaysAndWhenInUseUsageDescription</key>
<string>Need Location permission</string>
<key>NSLocationAlwaysUsageDescription</key>
<string>Need Location permission</string>
<key>NSLocationWhenInUseUsageDescription</key>
<string>Need Location permission</string>
```

Replace the 'main.dart' content to import the flutter_blue library and permission handler. Then modify the code to scanning for Bluetooth devices and add these to a device list. The device list will include devices that are currently connected as well as devices that are not connected.

From the displayed list of devices, you can scroll to a device that you are seeking. In this case, you could look for the Arduino that you have named 'ButtonLED'. Once you find this, 'connect' to it. This gives you access to the characteristics offered by that device. You can read, write, or notify these.

The first third of the code imports the packages and defines the functionality of the Bluetooth scanner, and the next two thirds of the code define the user interface layout and behaviour.

We begin by importing the libraries to use:

```
import 'dart:convert';

import 'package:flutter/material.dart';
import 'package:flutter/services.dart';
import 'package:flutter_blue_plus/flutter_blue_plus.dart';
import 'package:permission_handler/permission_handler.dart';
```

Next, we define the void main( ) function.

```
void main() => runApp(const MyApp());

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) => MaterialApp(
    title: 'BLE Demo',
    theme: ThemeData(
      primarySwatch: Colors.blue,
    ),
    home: MyHomePage(title: 'Flutter BLE Demo'),
  );
}
```

This will prepare the display to show list of devices that have been discovered in a BLE scan.

```
class MyHomePageState extends State<MyHomePage> {
  final _writeController = TextEditingController();
  BluetoothDevice? _connectedDevice;
  List<BluetoothService> _services = [];

  _addDeviceTolist(final BluetoothDevice device) {
    if (!widget.devicesList.contains(device)) {
      setState(() {
        widget.devicesList.add(device);
      });
    }
  }
```

We define a function to initialise scanning.

```
_initBluetooth() async {
    var subscription = FlutterBluePlus.onScanResults.listen(
        (results) {
      if (results.isNotEmpty) {
        for (ScanResult result in results) {
          _addDeviceTolist(result.device);
        }
      }
    },
    onError: (e) => ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(
        content: Text(e.toString()),
      ),
    ),
  );

  FlutterBluePlus.cancelWhenScanComplete(subscription);

  await FlutterBluePlus.adapterState.where((val) => val == BluetoothAdapterState.on).first;

  await FlutterBluePlus.startScan();

  await FlutterBluePlus.isScanning.where((val) => val == false).first;
  FlutterBluePlus.connectedDevices.map((device) {
    _addDeviceTolist(device);
  });
}

  @override
  void initState() {
    () async {
      var status = await Permission.location.status;
      if (status.isDenied) {
        final status = await Permission.location.request();
        if (status.isGranted || status.isLimited) {
```

```
        _initBluetooth();
      }
    } else if (status.isGranted || status.isLimited) {
      _initBluetooth();
    }

    if (await Permission.location.status.isPermanentlyDenied) {
      openAppSettings();
    }
  }();
  super.initState();
 }
```

Then add any new devices to the list.

```
ListView _buildListViewOfDevices() {
  List<Widget> containers = <Widget>[];
  for (BluetoothDevice device in widget.devicesList) {
    containers.add(
      SizedBox(
        height: 50,
        child: Row(
          children: <Widget>[
            Expanded(
              child: Column(
                children: <Widget>[
                  Text(device.platformName == '' ? '(unknown device)' : device.advName),
                  Text(device.remoteId.toString()),
                ],
              ),
            ),
            TextButton(
              child: const Text(
                'Connect',
                style: TextStyle(color: Colors.black),
              ),
              onPressed: () async {
                FlutterBluePlus.stopScan();
                try {
                  await device.connect();
                } on PlatformException catch (e) {
                  if (e.code != 'already_connected') {
                    rethrow;
                  }
                } finally {
                  _services = await device.discoverServices();
                }
                setState(() {
                  _connectedDevice = device;
```

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

  return ListView(
    padding: const EdgeInsets.all(8),
    children: <Widget>[
      ...containers,
    ],
  );
}
```

We add a set of buttons to allow connected devices to be interacted with.

```
List<ButtonTheme> _buildReadWriteNotifyButton(BluetoothCharacteristic characteristic) {
  List<ButtonTheme> buttons = <ButtonTheme>[];

  if (characteristic.properties.read) {
    buttons.add(
      ButtonTheme(
        minWidth: 10,
        height: 20,
        child: Padding(
          padding: const EdgeInsets.symmetric(horizontal: 4),
          child: TextButton(
            child: const Text('READ', style: TextStyle(color: Colors.black)),
            onPressed: () async {
              var sub = characteristic.lastValueStream.listen((value) {
                setState(() {
                  widget.readValues[characteristic.uuid] = value;
                });
              });
              await characteristic.read();
              sub.cancel();
            },
          ),
        ),
      ),
    );
  }
  if (characteristic.properties.write) {
    buttons.add(
      ButtonTheme(
        minWidth: 10,
        height: 20,
        child: Padding(
```

```
          padding: const EdgeInsets.symmetric(horizontal: 4),
          child: ElevatedButton(
            child: const Text('WRITE', style: TextStyle(color: Colors.black)),
            onPressed: () async {
              await showDialog(
                  context: context,
                  builder: (BuildContext context) {
                    return AlertDialog(
                      title: const Text("Write"),
                      content: Row(
                        children: <Widget>[
                          Expanded(
                            child: TextField(
                              controller: _writeController,
                            ),
                          ),
                        ],
                      ),
                      actions: <Widget>[
                        TextButton(
                          child: const Text("Send"),
                          onPressed: () {
                            characteristic.write(utf8.encode(_writeController.value.text));
                            Navigator.pop(context);
                          },
                        ),
                        TextButton(
                          child: const Text("Cancel"),
                          onPressed: () {
                            Navigator.pop(context);
                          },
                        ),
                      ],
                    );
                  });
            },
          ),
        ),
      ),
    ),
  );
}
if (characteristic.properties.notify) {
  buttons.add(
    ButtonTheme(
      minWidth: 10,
      height: 20,
      child: Padding(
        padding: const EdgeInsets.symmetric(horizontal: 4),
        child: ElevatedButton(
          child: const Text('NOTIFY', style: TextStyle(color: Colors.black)),
```

```
            onPressed: () async {
              characteristic.lastValueStream.listen((value) {
                setState(() {
                  widget.readValues[characteristic.uuid] = value;
                });
              });
              await characteristic.setNotifyValue(true);
            },
          ),
        ),
      ),
    );
  }

  return buttons;
}
```

And we also check for devices that have previously been connected.

```
ListView _buildConnectDeviceView() {
  List<Widget> containers = <Widget>[];

  for (BluetoothService service in _services) {
    List<Widget> characteristicsWidget = <Widget>[];

    for (BluetoothCharacteristic characteristic in service.characteristics) {
      characteristicsWidget.add(
        Align(
          alignment: Alignment.centerLeft,
          child: Column(
            children: <Widget>[
              Row(
                children: <Widget>[
                  Text(characteristic.uuid.toString(), style: const TextStyle(fontWeight:
FontWeight.bold)),
                ],
              ),
              Row(
                children: <Widget>[
                  ..._buildReadWriteNotifyButton(characteristic),
                ],
              ),
              Row(
                children: <Widget>[
                  Expanded(child: Text('Value: ${widget.readValues[characteristic.uuid]}')),
                ],
              ),
              const Divider(),
            ],
          ),
```

```
          ),
        );
      }
      containers.add(
        ExpansionTile(title: Text(service.uuid.toString()), children: characteristicsWidget),
      );
    }

    return ListView(
      padding: const EdgeInsets.all(8),
      children: <Widget>[
        ...containers,
      ],
    );
  }

  ListView _buildView() {
    if (_connectedDevice != null) {
      return _buildConnectDeviceView();
    }
    return _buildListViewOfDevices();
  }

  @override
  Widget build(BuildContext context) => Scaffold(
    appBar: AppBar(
      title: Text(widget.title),
    ),
    body: _buildView(),
  );
}
```