



Developer Study Guide: An introduction to Bluetooth Low Energy Development

Creating a Bluetooth Low Energy application for Android

Version: 5.0.3

Last updated: 17th December 2018

Contents

REVISION HISTORY	6
INTRODUCTION.....	8
Overview	8
Terminology Reminder	9
Functional Requirements	10
Android Bluetooth Classes	11
GETTING STARTED	12
Permissions	16
Java Packages	17
Constants	17
EXERCISE 1 - SCANNING FOR PERIPHERAL DEVICES.....	18
1. Scanning and Scan Results Screen	18
2. Consuming Scanning Results	22
3. BleScanner Class	24
Class Constructor and Checking the Status of the Bluetooth Adapter	24
startScanning method	25
stopScanning method	26
ScanCallback.....	27
Scanning Getter / Setter Methods	27
4. MainActivity Class – Part 1	29
MainActivity Class variables.....	29
onCreate method	29
Scanning Screen Title	30
setText method.....	30
setScanState method	31
Remaining compilation errors	32

5. PeripheralControlActivity Class – Part 1	33
Create the new class	33
Class constants.....	33
AndroidManifest.xml update	34
6. MainActivity Class – Part 2	34
Responding to the Find Button	34
requestLocationPermission method.....	35
startScanning method.....	36
scanningStarted and scanningStopped.....	37
candidateBleDevice.....	37
7. Checkpoint: Test Scanning	38
8. BleScanner - Filtering	40
9. Summary of Exercise 1	41
EXERCISE 2 – COMMUNICATING WITH THE PERIPHERAL DEVICE	42
1. BleAdapterService Class – Part 1	42
Bluetooth-related objects	42
Creating a BluetoothAdapter Object	43
Service Binding.....	43
Activity and Service Communication	44
AndroidManifest.xml Service Entry	45
2. Peripheral Control Screen	45
activity_peripheral_control.xml.....	45
Service Connection	49
Creating a Handler for Service to Activity Communication	49
onClick Event Handlers	50
onCreate.....	51
onDestroy.....	52
Checkpoint: Select a Device	53
3. Connecting to the Peripheral Device	54
connect and disconnect methods.....	54
BluetoothGattCallback.....	55

The CONNECT button.....	56
Connected and Disconnect Events.....	57
Disconnect when back button is pressed	57
Checkpoint: Connecting to the Peripheral Device	58
4. Profile Check	58
Bluetooth procedures and our generally applicable design pattern	59
Service Discovery	59
Checkpoint: check service discovery and profile validation	62
5. Setting the Alert Level	62
Enable Buttons Only When Connected.....	62
Highlight Selected Alert Level	64
Reading and writing the Alert Level Bluetooth characteristic	64
Checkpoint: Test setting the Alert Level	70
6. Monitoring Signal Strength	71
readRemoteRssi	71
onReadRemoteRssi	71
Receiving RSSI values	72
RSSI and the UI.....	73
Checkpoint: Test RSSI monitoring.....	75
7. Immediate Alert	75
Enable the Make a Noise button	76
Triggering an immediate alert	76
Checkpoint: Test Immediate Alert	77
8. Sharing Proximity Data	77
The Sharing Switch.....	77
Writing Proximity Data to the Peripheral Device.....	78
Checkpoint: Test proximity sharing	79
9. Link Loss	81
AlarmManager	81
Adding an audio file to the project	83
Sounding the alarm on link loss	83
Stop alarm on reconnect.....	84

10. Temperature Monitoring	85
UI Changes	85
Switch Initialisation	86
Connection State Changes	87
Temperature Measurements	88
TEST YOUR APPLICATION	93
PROJECT STRUCTURE	94

Revision History

Version	Date	Author	Changes
1.0	1 st May 2013	Matchbox	Initial version
2.0	3rd September 2014	Martin Woolley, Bluetooth SIG	Make a Noise button now triggers the Immediate Alert service instead of a custom service. Switch added to enable/disable sharing of client proximity data with a new custom service called the Proximity Monitoring service.
3.0.0	7 th July 2016	Martin Woolley, Bluetooth SIG	Name change and version number increment to sync with changes to the Arduino lab for V3.0.
3.1.0	3 rd October	Martin Woolley, Bluetooth SIG	<p>Android code retired to the legacy folder and designated 'Android 4' as it uses the Android 4.4 APIs which have been deprecated.</p> <p>Android lab modified to be based on Android Studio instead of Eclipse.</p> <p>Code substantially refactored.</p> <p>Android lab and solution now uses the Android 5+ APIs and is compatible with Android 6+ with respect to the new permissions model.</p> <p>Key Android Bluetooth classes are now introduced and explained.</p> <p>New much more alarm-like alarm sound used.</p>
3.2.0	16 th December 2016	Martin Woolley, Bluetooth SIG	iOS Objective-C resources retired and replaced with new lab and solution written in Swift. No changes to the Android

			resources.
4.0.0	7 th July 2017	Martin Woolley, Bluetooth SIG	<p>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.</p> <p>Android lab text updated to improve description of recommended test steps in Checkpoint: Test setting the Alert Level.</p>
5.0.0	14 th December 2017	Martin Woolley, Bluetooth SIG	Added new use case involving temperature measurements and Bluetooth Indications.
5.0.2	15 th June 2018	Martin Woolley, Bluetooth SIG	Removed use of Bluetooth Developer Studio.
5.0.3	17 th December 2018	Martin Woolley Bluetooth SIG	Name changed to “Developer Study Guide: An introduction to Bluetooth Low Energy Development”

Introduction

Overview

This document is a step-by-step guide to creating a Bluetooth application for Android. The application is intended to be used with the Bluetooth peripheral device created in one of the server labs included in this study guide and so ideally, you should complete that lab first.

In this lab we will not cover any basics of Android programming or how to set up an Android development environment, nor is the resulting code necessarily suitable for production – the lab, tasks and code are designed to provide instruction in the principles and practice of developing Android applications which use Bluetooth Low Energy (LE) technology.

Objectives

This lab provides instructions to achieve the following:

- Request permissions from the user to perform Bluetooth ‘scanning’, if necessary.
- Scan for and discover a nearby Bluetooth peripheral device.
- Connect to a selected Bluetooth peripheral
- Communicate with the Bluetooth peripheral device, exercising the capabilities of that device’s Bluetooth profile. In this case, the peripheral device created in one of the server labs, acts as a specialised proximity device using a customised version of the Bluetooth standard Proximity profile. If you’re not clear what a profile is, then at least read the first few sections of the LE Basic Theory document.

System Requirements

- Android Studio with SDK Platforms 5.1.1 and 6.0 installed and any other required, external dependencies.

Equipment Requirements

- USB cable
- Android tablet or smartphone running Android 5.1 or better
- A device such as an Arduino or Raspberry Pi acting as the Bluetooth peripheral, running the solution to one of the server labs which are part of this study guide. See the Servers folder.

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 method or class 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**.

Compilation Errors

You will be building a full application in relatively small steps. At some stages, you'll have compilation errors. Don't worry, as generally speaking, this is expected and compilation errors will be dealt with before reaching the next testing checkpoint. The lab does not always explicitly tell you what to do with compilation errors. Simple issues such as needing to import a class are left to you, since you are expected to have at least a little experience of Java and Android programming. Other types of issue, such as calling a method we've not yet written, will get resolved later in the lab. As a guide, you should not have any compilation errors whenever you reach a "Checkpoint" testing step in the lab. If you do then you should check your code carefully against the previous lab steps to make sure you haven't made any mistakes and use your knowledge of Java to resolve simple issues.

Troubleshooting

If you encounter problems as you proceed, check the preceeding lab steps carefully to ensure you have followed the instructions correctly. If you're concerned you may not have your source code files in the right place, check the [Project Structure](#) section towards the end of the lab, where you'll find details of the files and folders your Android Studio project should have by the end of the exercise.

As a last resort, don't forget that the lab is packaged with a complete solution so you can always look at that and see if it helps determine what is wrong with your own solution.

Terminology Reminder

Our completed Bluetooth system will consist of two major parts; a device (#1) running some software which will display proximity information and allow users to initiate the flashing of lights and sounding of the buzzer in our custom electronic circuit and a device (#2) which is connected electrically to that circuit and which controls its state based on communication it receives over Bluetooth from the other device.

Device #1 will usually be referred to as a client or sometimes as a Central mode device. It will often be an application running on a smartphone but could be implemented some other way, such as in a web browser.

Device #2 will sometimes be referred to as a peripheral device, sometimes as a server and sometimes as a GAP Peripheral. Which is used will depend on the context. They all refer to the same thing and are all equally valid in a given context. Typically this device will be something like an Arduino or a Raspberry Pi.

"Client" and "Server" come from GATT. "Peripheral" and "Central" come from GAP. "Peripheral device" is informal. Re-read the LE Basic Theory guide if any of this is not making sense.

Functional Requirements

When finished, the application we will create in this lab will:

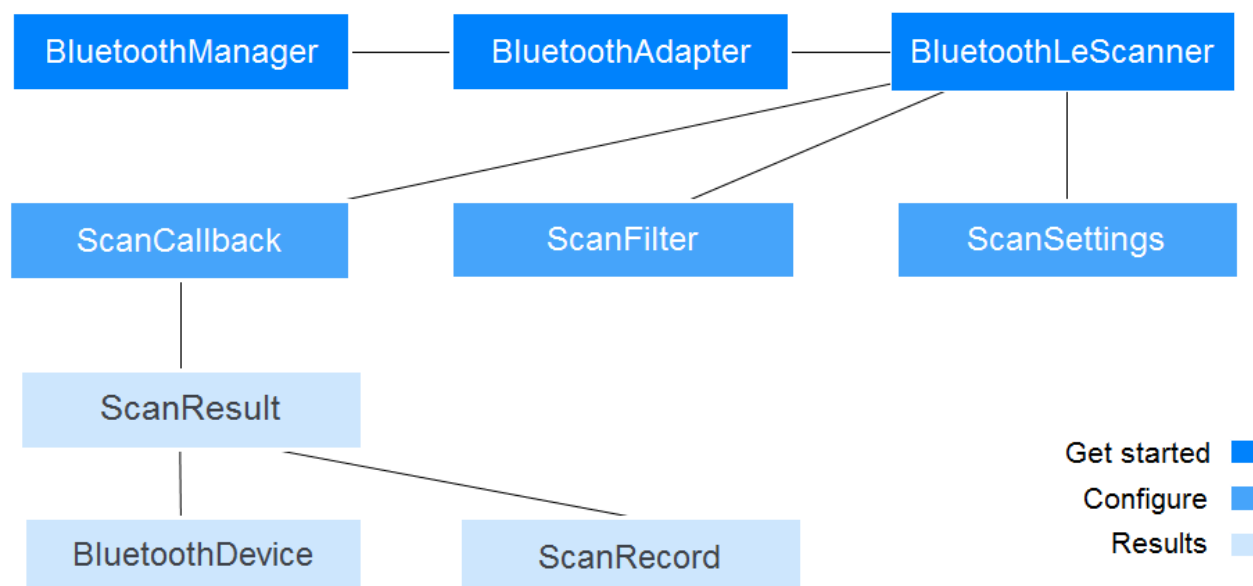
Feature		Expected Behaviour
1	Be able to scan for and discover devices running and advertising the custom profile we designed and implemented in the server labs of this starter study guide.	Clicking a button will initiate Bluetooth scanning. Filtering will ensure that only devices which are advertising with a device name of "BDSK" are selected and presented to the user in a UI list for selection.
2	Allow the user to establish a Bluetooth connection between their smartphone or tablet and the selected Bluetooth peripheral device.	Clicking a button will cause a Bluetooth connection to be established. The UI should indicate in a simple way that a connection has been created.
3	Allow the user to set an alert level of 0, 1 or 2 to be used by the peripheral when indicating that the Bluetooth link has been lost or when the user explicitly commands the peripheral to make a noise.	The UI will include three colour coded "alert level" buttons (0=green=low, 1=yellow=medium, 2=red=high). When clicked by the user, the smartphone application will write selected value of 0, 1 or 2 to the Link Loss Service's Alert Level characteristic.
4	Allow the user to instruct the peripheral that it should make a noise and flash its lights so that it can be easily located if lost or hidden.	The UI will include a button labelled "Make a noise". When clicked, the smartphone application will write the selected alert level value of 0, 1 or 2 to the peripheral's Immediate Alert Service's Alert Level characteristic.
5	Proximity Monitoring	The connected smartphone application will track its distance from the peripheral device using the received signal strength indicator (RSSI) and classify it as near (green), middle distance (yellow) or far away (red), indicated by a prominent, colour-coded rectangle on the main UI screen.
6	Proximity Sharing	The user must be able to enable or disable this feature using a suitable UI switch. When enabled, the RSSI and a distance classification of 1=near, 2=medium or 3=far will be periodically sent to the peripheral device by writing to the Client Proximity characteristic of the custom Proximity Monitoring service.
7	Link Lost / Disconnected	If the connection between smartphone and peripheral device is lost then the smartphone applicationshould make a noise for an extended period of time. The volume of the noise made should be loudest if the alert level set in feature (3) is 2, less loud if it was set to 1 and silent if it was set to 0. In all cases, the UI should indicate that the connection has been lost and re-enable the Connect button so that the user can attempt to reconnect.

8	Temperature monitoring	The user must be able to enable or disable temperature monitoring. Temperature measurements received from the connected peripheral should be displayed.
---	------------------------	---

Android Bluetooth Classes

We'll be using Java classes from Android's `android.bluetooth` package and `android.bluetooth.le` package. You should use the Android API documentation to familiarise yourself with classes we'll be using, as you encounter them in this lab. See <https://developer.android.com/reference/packages.html> for full details.

A short summary of key classes follows, however to help you get started.,

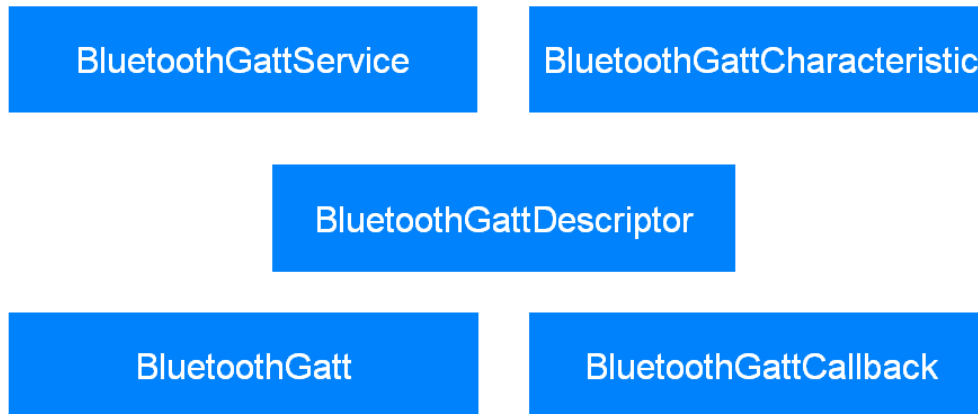


`BluetoothManager` is used to acquiring access to the Bluetooth adapter in your mobile device and for checking that Bluetooth is switched on.

`BluetoothAdapter` gives us access to a `BluetoothLeScanner` object which we use to find other Bluetooth devices in the environment, which are “advertising”.

We can configure the scanning which `BluetoothLeScanner` does and give it filter rules to apply so that only devices of interest are selected. This is the role of `ScanFilter` and `ScanSettings`.

`ScanCallback` is an abstract class we have to extend. It provides a mechanism for the `BluetoothLeScanner` to report its findings in terms of devices, represented by `BluetoothDevice` objects and `ScanRecords` which contain the Bluetooth advertising data which was received.



BluetoothGattService, **BluetoothGattCharacteristic** and **BluetoothGattDescriptor** represent GATT services, characteristics and descriptors from a GATT profile. If you're unfamiliar with these terms, please review the information contained within the LE Basic Theory document which serves as an introduction to basic Bluetooth LE technical concepts.


BluetoothGatt provides methods with which to initiate various Bluetooth procedure. **BluetoothGattCallback** must be extended to receive the results of such operations.

Getting Started

Your first task is to create a new Android Studio project to work within.

1. If you do not already have it installed and working, download and install Android Studio from <https://developer.android.com/studio/index.html>
2. Run Android Studio.
3. Create new project by selecting File > New > New Project or from the initial wizard dialogue. Complete the New Project dialogue as shown below.

Create New Project ✕

 **New Project**
Android Studio

Configure your new project

Application name:

Company Domain: [Edit](#)

Package name:

Project location: ...


Previous Next Cancel Finish

Note that the Project Location is not critical but you should enter values for the other fields exactly as shown.

4. Click Next and in the next dialogue, set the minimum SDK level to 22 (Android 5.1).

Create New Project

×

 Target Android Devices

Select the form factors your app will run on

Different platforms may require separate SDKs

☒ Phone and Tablet

Minimum SDK

Lower API levels target more devices, but have fewer features available.
By targeting API 22 and later, your app will run on approximately **24.1%** of the devices that are active on the Google Play Store.
[Help me choose](#)

☐ Wear

Minimum SDK

☐ TV

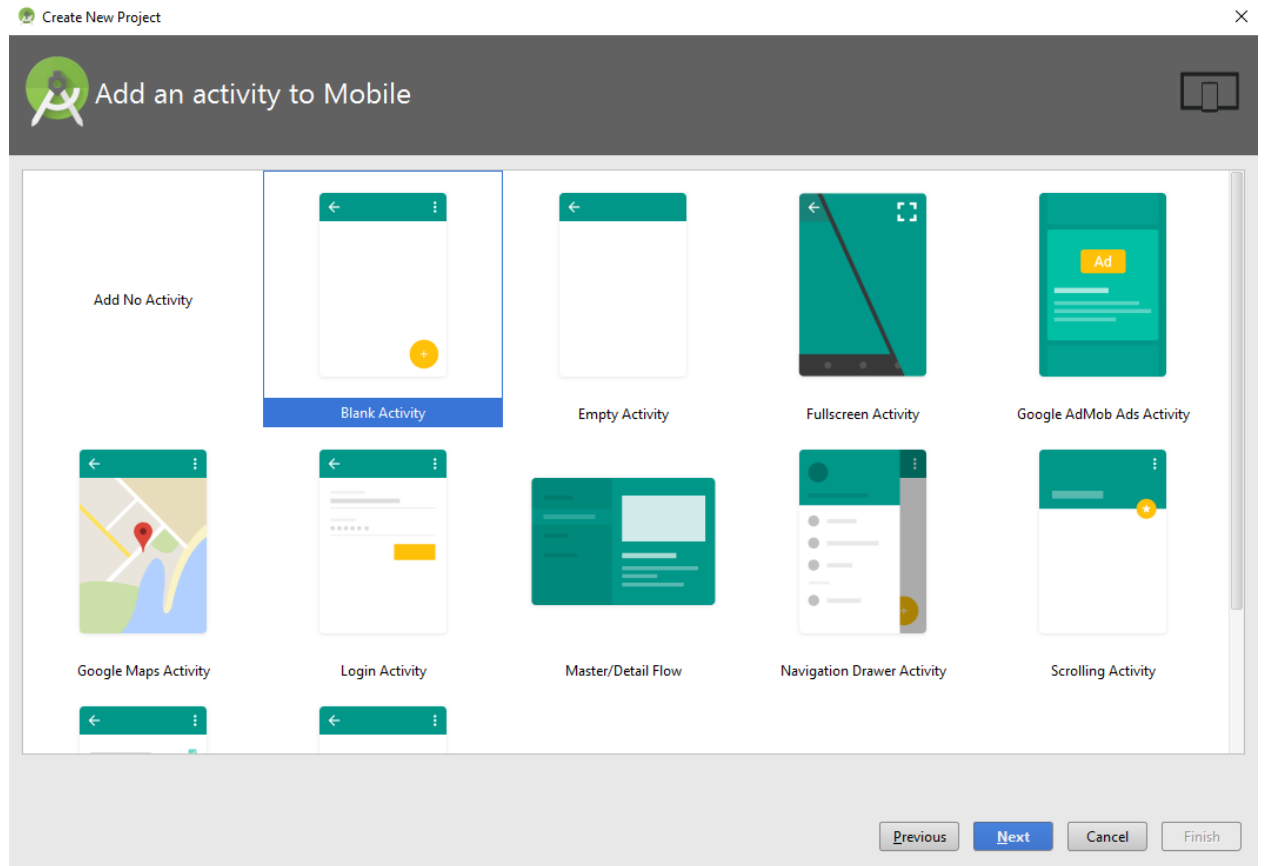
Minimum SDK

☐ Android Auto

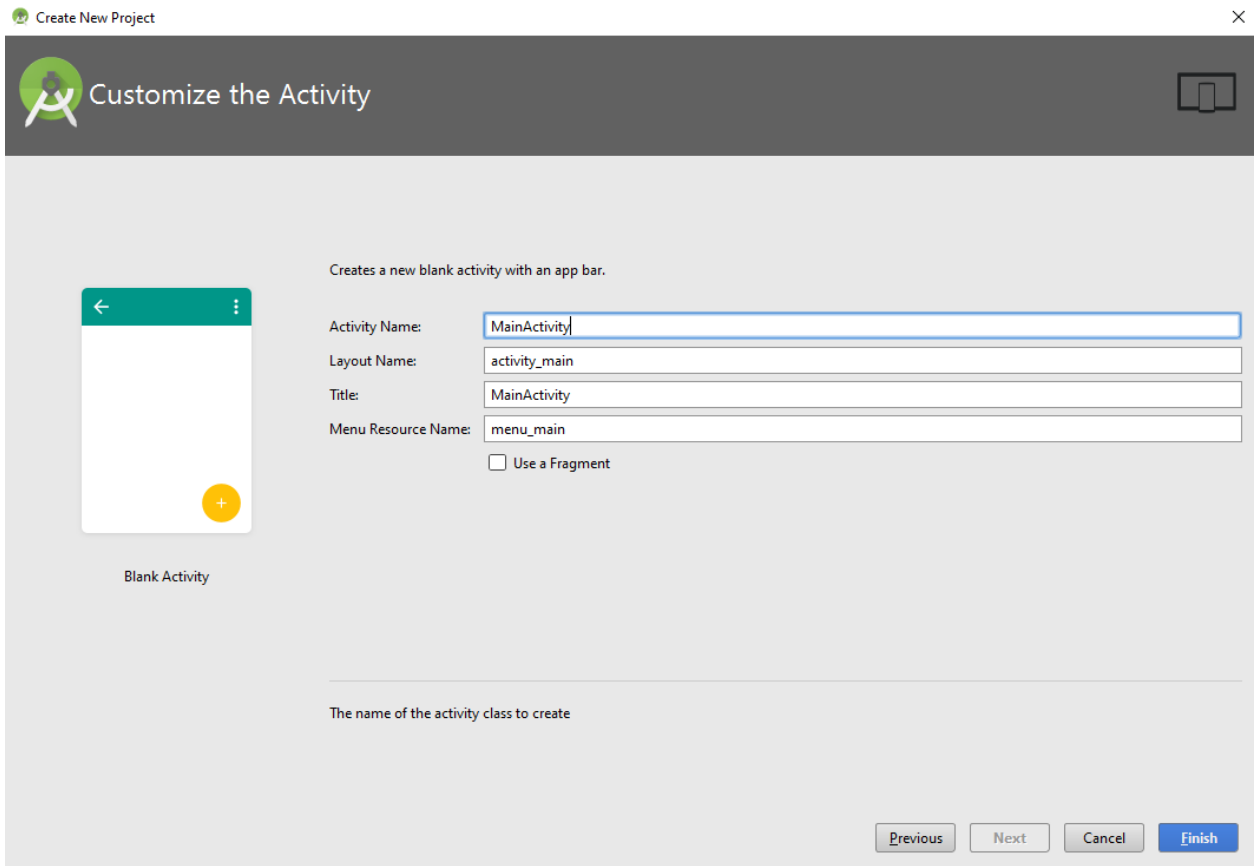
☐ Glass

Minimum SDK

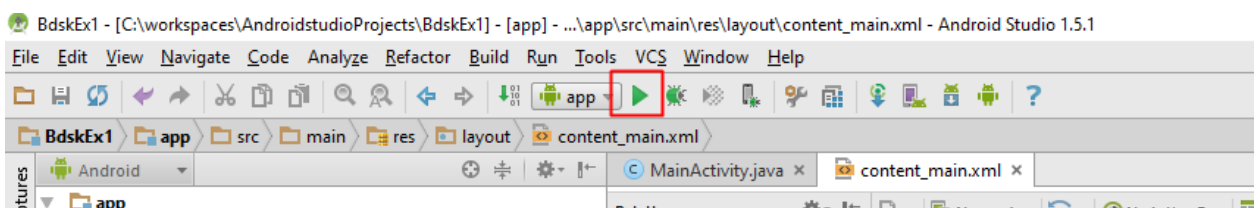
5. Select Next and retain the default values, as shown on the next dialogue.



6. Click Next and once again, retain the default values on the next and final dialogue in this sequence.



7. Click Finish
8. Check that your new project and development environment are working OK by plugging your Android device into your computer via USB and clicking the Run button. Your project should build, install onto your device and execute. This verifies that your development environment is fundamentally working.



Permissions

Add the following permissions underneath the 'manifest' element in AndroidManifest.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
```



```
package="com.bluetooth.mwoolley.microbitbledemo">
```

```
<uses-permission android:name="android.permission.BLUETOOTH" />
```

```
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
```

```
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
```

The BLUETOOTH permission says our application wants to use Bluetooth. The BLUETOOTH_ADMIN permission allows us to determine whether Bluetooth is switched on or off and if we want to, ask the user to switch it on. The 3rd of the permissions, ACCESS_COARSE_LOCATION is required if we want to scan for other devices. Scanning is regarded as a location determination procedure in terms of the Android permission system. Note that this permission was added in Android 6.

Java Packages

We'll use separate Java packages for those classes which are concerned with our UI and those which are concerned with Bluetooth. Create packages com.bluetooth.bdsk.ui and com.bluetooth.bdsk.bluetooth now. When done, move the MainActivity.java class which was created by the new project wizard into the ui package.

Constants

We'll use various constants in our application. Create a class called Constants.java in the default package and to start with, set its code to the following:

```
package com.bluetooth.bdsk;

public class Constants {

    public static final String TAG = "bdsk";

    public static final String FIND = "Find BDSK Devices";

    public static final String STOP_SCANNING = "Stop Scanning";

    public static final String SCANNING = "Scanning";

}
```

Exercise 1 - Scanning for peripheral devices

For this exercise you need to have your peripheral device, running the code created in one of the server labs (or the complete solution code which we have included). Our goal in this exercise is to give our Android application the ability to scan for and list devices it finds within range that are suitable for use from this application. We'll find out more about exactly what this means as we progress through the steps which follow.

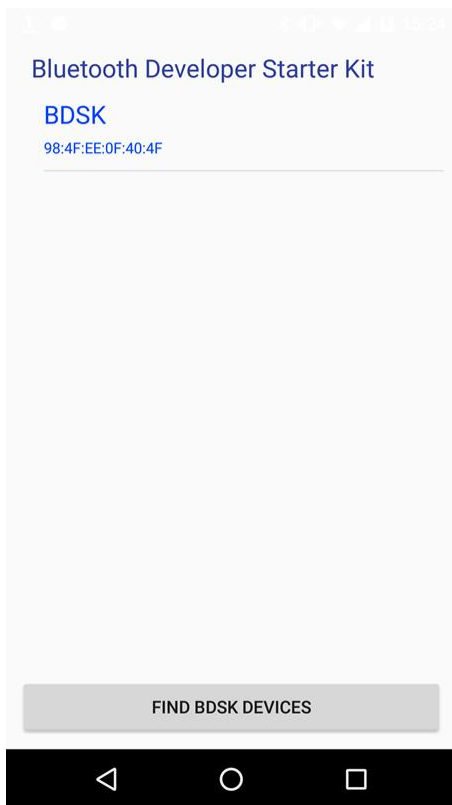
In order to use Bluetooth features in an application, we need to declare the several permissions as being required in the AndroidManifest.xml file. Android 6 changed the way these permissions are used, introducing a new permissions model, and we'll see how we deal with this for both Android 6+ and for Android 5, shortly.

1. Scanning and Scan Results Screen

Your next task is to create a screen which has a 'Scan' button, which when pressed by the user, will initiate Bluetooth scanning and will list any 'suitable' Bluetooth devices which are found. This is one part of the Bluetooth 'device discovery process'. For a description of this and other Bluetooth LE concepts, read the LE Basic Theory document.

We'll create this screen, using the MainActivity.java class as a basis. We'll proceed by modifying that class and by creating a screen definition which supports our requirements using Android's XML layout system, as usual.

Below is a screen shot of what the scanning /scan results screen should look like when the application is running.



Lets define the xml for this screen. Open `res\layout\activity_main.xml` and replace its contents with the below xml, and save the file.

Table 1 - activity_main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="10dp"
    android:paddingLeft="10dp"
    android:paddingRight="10dp"
    android:paddingTop="10dp"
    android:orientation="vertical"
    tools:context=".ui.MainActivity" >
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:gravity="left"
        android:paddingLeft="10dip"
        android:textSize="20dp"
```

```
        android:paddingBottom="10dp"
        android:textColor="@color/colorPrimaryDark"
        android:text="Bluetooth LE Developer Study Guide" />
    <ListView
        android:id="@+id/deviceList"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:paddingLeft="20dip"
        android:layout_weight="1" />
    <Button
        android:id="@+id/scanButton"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:enabled="true"
        android:onClick="onScan"
        android:text="" />
</LinearLayout>
```


activity_main.xml includes a ListView which will contain details of devices found and a Button which we'll use to trigger the scanning process. The ListView needs some XML which defines the structure of each member of the list as well, so we'll add that next.


Select File->New->XML->Layout file

Name the file **list_row** and retain the default root tag of 'LinearLayout'. Click Finish.

New Android Activity

×

Customize the Activity



Creates a new XML layout file.

Layout File Name:

Root Tag:

Name of the layout XML file.

Previous

Next

Cancel

Finish

Replace the contents of list_row.xml with the following xml, and save the file.

Table 2 - list_row.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:paddingBottom="5dp"
        android:textSize="20dp"
        android:textColor="#0040FF"
        android:text="" />

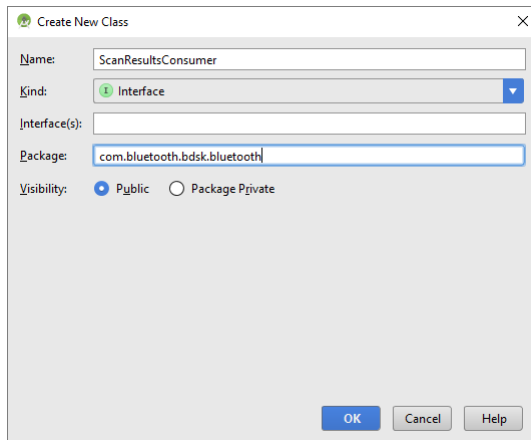
    <TextView
        android:id="@+id/bdaddr"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:paddingBottom="10dp"
        android:textSize="12dp"
        android:textColor="#0040FF"
        android:text="" />

</LinearLayout>
```

This is a linear layout with a textview which is used to display the device name in one row of the list row.

2. Consuming Scanning Results

Create a Java interface called ScanResultsConsumer in the com.bluetooth.bdsk.bluetooth package.



Replace its default content with the following code.

```
package com.bluetooth.bdsk.bluetooth;
import android.bluetooth.BluetoothDevice;
public interface ScanResultsConsumer {

    public void candidateBleDevice(BluetoothDevice device, byte[] scan_record, int rssi);
    public void scanningStarted();
    public void scanningStopped();

}
```

MainActivity.java needs to implement this interface so that it can receive and process data produced by the Bluetooth scanning process. Add the Implements clause to the MainActivity class declaration.

```
public class MainActivity extends AppCompatActivity implements ScanResultsConsumer {
```

Use the Android Studio auto-completion facility (select item in error and press Alt+Enter) to generate stub implementations of the three methods defined by this interface.

```
@Override
public void candidateBleDevice(BluetoothDevice device, byte[] scan_record, int rssi) {

}

@Override
public void scanningStarted() {

}
```

```
@Override  
public void scanningStopped() {  
  
}
```

3. BleScanner Class

We'll encapsulate scanning in a class called BleScanner. Create it now, in the bluetooth package. We'll add code one step at a time so we can scrutinise it and understand what it's doing, in "bite sized chunks". You will probably see compilation errors flagged as you complete some of the steps to create this class. At each intermediate step, this doesn't matter. Your BleScanner class should compile without errors once you have added all of the code which follows, however.

Class Constructor and Checking the Status of the Bluetooth Adapter

Start with the class containing the following code:

```
package com.bluetooth.bdsk.bluetooth;  
import android.bluetooth.BluetoothAdapter;  
import android.bluetooth.BluetoothManager;  
import android.bluetooth.le.BluetoothLeScanner;  
import android.bluetooth.le.ScanCallback;  
import android.bluetooth.le.ScanFilter;  
import android.bluetooth.le.ScanResult;  
import android.bluetooth.le.ScanSettings;  
import android.content.Context;  
import android.content.Intent;  
import android.os.Handler;  
import android.util.Log;  
import com.bluetooth.bdsk.Constants;  
import java.util.ArrayList;  
import java.util.List;  
  
public class BleScanner {  
    private BluetoothLeScanner scanner = null;  
    private BluetoothAdapter bluetooth_adapter = null;  
    private Handler handler = new Handler();  
    private ScanResultsConsumer scan_results_consumer;  
    private Context context;  
    private boolean scanning=false;  
    private String device_name_start="";  
  
    public BleScanner(Context context) {
```



```

        this.context = context;

        final BluetoothManager bluetoothManager = (BluetoothManager)
context.getSystemService(Context.BLUETOOTH_SERVICE);

        bluetooth_adapter = bluetoothManager.getAdapter();

        // check bluetooth is available and on
        if (bluetooth_adapter == null || !bluetooth_adapter.isEnabled()) {
            Log.d(Constants.TAG, "Bluetooth is NOT switched on");
            Intent enableBtIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
            enableBtIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
            context.startActivity(enableBtIntent);
        }
        Log.d(Constants.TAG, "Bluetooth is switched on");
    }
// end of class
}

```

This creates the class, declares various member variables which we'll need and creates the class constructor. The constructor takes a Context object as an argument so that we can use it to start an activity, which we need to do in the event that we find that Bluetooth is currently switched off.

The Android BluetoothManager class provides us with an instance of BluetoothAdapter. We use it to check whether or not Bluetooth is currently switched on and if it is not, prompt the user to enable it.

startScanning method

Next, we add methods which allow another object to initiate Bluetooth scanning. A public startScanning method which requires a time limit for scanning to be specified as an argument as well as an instance of our ScanResultsConsumer interface so that callbacks can be made to its methods during scanning.

```

    public void startScanning(final ScanResultsConsumer scan_results_consumer, long
stop_after_ms) {
        if (scanning) {
            Log.d(Constants.TAG, "Already scanning so ignoring startScanning request");
            return;
        }
        if (scanner == null) {
            scanner = bluetooth_adapter.getBluetoothLeScanner();
            Log.d(Constants.TAG, "Created BluetoothScanner object");
        }
        handler.postDelayed(new Runnable() {
            @Override
            public void run() {

```

```

        if (scanning) {
            Log.d(Constants.TAG, "Stopping scanning");
            scanner.stopScan(scan_callback);
            setScanning(false);
        }

    }, stop_after_ms);

    this.scan_results_consumer = scan_results_consumer;
    Log.d(Constants.TAG, "Scanning");
    List<ScanFilter> filters;
    filters = new ArrayList<ScanFilter>();
    ScanSettings settings = new
    ScanSettings.Builder().setScanMode(ScanSettings.SCAN_MODE_LOW_LATENCY).build();
    setScanning(true);
    scanner.startScan(filters, settings, scan_callback);
}

```

Note that we use a boolean to keep track of whether we're currently scanning or not. We obtain an instance of `BluetoothLeScanner` from the `BluetoothAdapter` class and it's this object that we'll use to start the Bluetooth scanning process off. Before we do though, we post a `Runnable` object for delayed execution to a `Handler`, using this technique to stop scanning after the specified time limit has been reached.

After setting up our deferred call to `stopScanning`, we set up a `ScanFilter` object and a `ScanSettings` object and use each in our call to the `startScan` method of the `BluetoothLeScanner` object. The `ScanFilter` list we create here is currently empty. We'll return to this later.

The `ScanSettings` object is using the `SCAN_MODE_LOW_LATENCY` setting and this means scanning will be fairly aggressive, meaning advertising packets will be collected quickly. Be careful with this setting as scanning in general is a fairly expensive procedure with respect to battery use. Android offers less aggressive scanning settings which may be more suitable for your purposes. We'll use it here though so that we get the most responsive user experience.

Note that we haven't yet created a `ScanCallback` object so you'll have compilation errors at this point.

stopScanning method

Add the following, simple method by which scanning can be terminated.

```

public void stopScanning() {
    setScanning(false);
    Log.d(Constants.TAG, "Stopping scanning");
    scanner.stopScan(scan_callback);
}

```

ScanCallback

Now let's create the ScanCallback object that our BluetoothLeScanner object needs. We'll create it as an inner class for convenience and implement the required onScanResult method right here. This method is going to be called every time the scanner collects a Bluetooth advertising packet which complies with our filtering criteria i.e. it includes DEVICE_NAME="BDSK". As you can see, we check our 'scanning' boolean in case scanning has just been terminated and then make a callback to the ScanResultsConsumer object which was provided to this class' constructor, passing the BluetoothDevice object that represents the remote device which emitted the advertising packet. We also pass the content of the advertising packet as a raw byte array and the received signal strength indicator. We don't use either of these items in the lab at present, but they might be useful to you, should you decide to take the lab further.

```
private ScanCallback scan_callback = new ScanCallback() {  
    public void onScanResult(int callbackType, final ScanResult result) {  
        if (!scanning) {  
            return;  
        }  
        scan_results_consumer.candidateBleDevice(result.getDevice(),  
result.getScanRecord().getBytes(), result.getRssi());  
    }  
};
```

Note that the ScanResultsConsumer is actually our MainActivity class, so the net effect so far is that we've configured and initiated scanning, with a specified time limit and we will be passing details of suitable devices found back to the MainActivity so that each distinct device discovered can be displayed to the user in our UI list.

All that's left is to add a couple of getter/setter methods.

Scanning Getter / Setter Methods

```
public boolean isScanning() {  
    return scanning;  
}  
  
void setScanning(boolean scanning) {  
    this.scanning = scanning;  
    if (!scanning) {  
        scan_results_consumer.scanningStopped();  
    } else {  
        scan_results_consumer.scanningStarted();  
    }  
}
```

Note that `setScanning` informs the `ScanResultsConsumer` object of changes in the scanning state. This is useful so that the UI can be updated accordingly.

We've now got a class which can perform scanning for us when we ask it to and which will provide details of any Bluetooth devices it finds to another object.

That other object will be our `MainActivity` object which underpins the scanning / scan results screen of our application.

4. MainActivity Class – Part 1

MainActivity Class variables

Inside the MainActivity class at the top add the following declarations, which we will use later.

```
private boolean ble_scanning = false;
private Handler handler = new Handler();
private ListAdapter ble_device_list_adapter;
private BleScanner ble_scanner;
private static final long SCAN_TIMEOUT = 5000;
private static final int REQUEST_LOCATION = 0;
private static String[] PERMISSIONS_LOCATION = {Manifest.permission.ACCESS_COARSE_LOCATION};
private boolean permissions_granted=false;
private int device_count=0;
private Toast toast;
static class ViewHolder {
    public TextView text;
    public TextView bdaddr;
}
```

We'll see how each of these class member variables are used as we progress with the remainder of the lab.

onCreate method

Update the onCreate() function in the MainActivity class with this code:

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    setButtonText();
    ble_device_list_adapter = new ListAdapter();
    ListView listView = (ListView) this.findViewById(R.id.deviceList);
    listView.setAdapter(ble_device_list_adapter);
    ble_scanner = new BleScanner(this.getApplicationContext());
    listView.setOnItemClickListener(new AdapterView.OnItemClickListener() {
        @Override
        public void onItemClick(AdapterView<?> parent, View view,
                                int position, long id) {
            if (ble_scanning) {
                setScanState(false);
            }
        }
    });
}
```

```

        ble_scanner.stopScanning();
    }
    BluetoothDevice device = ble_device_list_adapter.getDevice(position);
    if (toast != null) {
        toast.cancel();
    }
    Intent intent = new Intent(MainActivity.this,
                                PeripheralControlActivity.class);
    intent.putExtra(PeripheralControlActivity.EXTRA_NAME, device.getName());
    intent.putExtra(PeripheralControlActivity.EXTRA_ID, device.getAddress());
    startActivity(intent);
}
});
}

```

You'll have quite a few compilation errors at this point. Resolve any that simply require a class to be imported. We'll get to the others soon.

Scanning Screen Title

Edit `res/values/strings.xml` and add a screen title constant:

```

<resources>
    <string name="app_name">Bdsk</string>
    <string name="screen_title_main">Device List</string>
    <string name="action_settings">Settings</string>
</resources>

```

setButtonText method

Add the following function to `MainActivity`, which sets or resets the text showing on the button:

```

private void setButtonText() {
    String text="";
    text = Constants.FIND;
    final String button_text = text;
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            ((TextView)
MainActivity.this.findViewById(R.id.scanButton)).setText(button_text);
        }
    });
}

```

```
}
```

setScanState method

Add this method, which changes the text on the scan screen's button according to whether or not scanning is currently being performed.

```
private void setScanState(boolean value) {  
    ble_scanning = value;  
    ((Button) this.findViewById(R.id.scanButton)).setText(value ? Constants.STOP_SCANNING :  
    Constants.FIND);  
}
```

Now we'll add a list adapter that serves to store a list of BluetoothDevices that are found during scanning, and provides the data for the list view. The adapter will use the list_row.xml layout we added earlier and return that view with the name of the found device for the list to display in the row.

Add the code below to the end of the MainActivity class.

```
private class ListAdapter extends BaseAdapter {  
    private ArrayList<BluetoothDevice> ble_devices;  
    public ListAdapter() {  
        super();  
        ble_devices = new ArrayList<BluetoothDevice>();  
    }  
    public void addDevice(BluetoothDevice device) {  
        if (!ble_devices.contains(device)) {  
            ble_devices.add(device);  
        }  
    }  
    public boolean contains(BluetoothDevice device) {  
        return ble_devices.contains(device);  
    }  
    public BluetoothDevice getDevice(int position) {  
        return ble_devices.get(position);  
    }  
    public void clear() {  
        ble_devices.clear();  
    }  
    @Override  
    public int getCount() {  
        return ble_devices.size();  
    }  
}
```

```

@Override
public Object getItem(int i) {
    return ble_devices.get(i);
}

@Override
public long getItemId(int i) {
    return i;
}

@Override
public View getView(int i, View view, ViewGroup viewGroup) {
    ViewHolder viewHolder;
    if (view == null) {
        view = MainActivity.this.getLayoutInflater().inflate(R.layout.list_row,
null);

        viewHolder = new ViewHolder();
        viewHolder.text = (TextView) view.findViewById(R.id.textView);
        viewHolder.bdaddr = (TextView) view.findViewById(R.id.bdaddr);
        view.setTag(viewHolder);
    } else {
        viewHolder = (ViewHolder) view.getTag();
    }
    BluetoothDevice device = ble_devices.get(i);
    String deviceName = device.getName();
    if (deviceName != null && deviceName.length() > 0) {
        viewHolder.text.setText(deviceName);
    } else {
        viewHolder.text.setText("unknown device");
    }
    viewHolder.bdaddr.setText(device.getAddress());
    return view;
}
}

```

A device may or may not include its name in Bluetooth advertising packets. For those devices that do include a name for themselves, we display that name in the list entry. If there is no device name in the advertising packet, instead we display “unknown device”. In both cases we include the device’s address in its details.

Remaining compilation errors

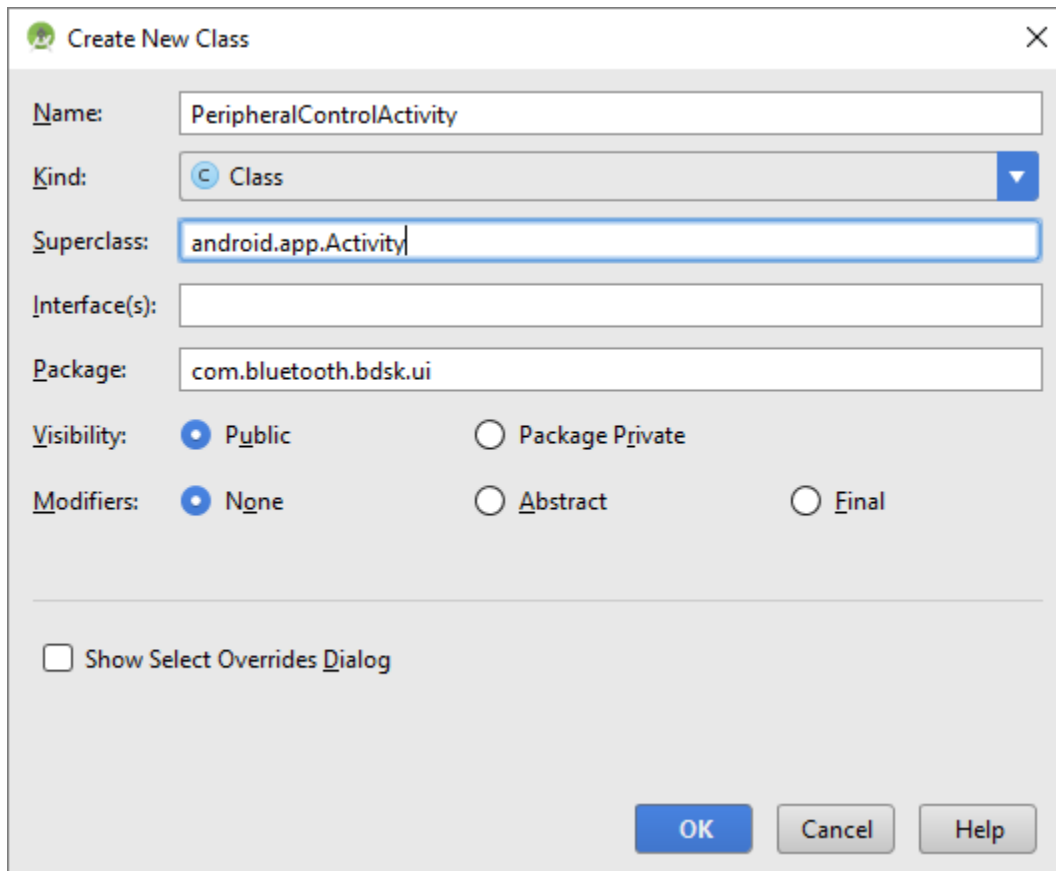
At this point only the following lines in MainActivity.java, in the onCreate method, should be indicating compilation errors and in each case this is simply due to the fact that we haven’t yet created the PeripheralControlActivity class.


```
Intent intent = new Intent(PeripheralControlActivity.this, PeripheralControlActivity.class);
intent.putExtra(PeripheralControlActivity.EXTRA_NAME, device.getName());
intent.putExtra(PeripheralControlActivity.EXTRA_ID, device.getAddress());
```

5. PeripheralControlActivity Class – Part 1

Create the new class

Create a class called PeripheralControlActivity.java in the com.bluetooth.bdsk.ui package and have it extend android.app.Activity.



Class constants

Add the two constants shown below so that your initial class definition is exactly as shown here:

```
package com.bluetooth.bdsk.ui;

import android.app.Activity;

public class PeripheralControlActivity extends Activity {
    public static final String EXTRA_NAME = "name";
    public static final String EXTRA_ID = "id";
}
```

```
}
```

AndroidManifest.xml update

Add a declaration for the new Activity to the manifest file:

```
<activity
    android:name=".ui.PeripheralControlActivity"
    android:label="@string/app_name"
    android:screenOrientation="portrait">
</activity>
```

6. MainActivity Class – Part 2

Responding to the Find Button

Next we need to add some code to the MainActivity class which will respond to the Find button being pressed. It will need to trigger Bluetooth scanning (unless we're already scanning) but before it can do so, it *may* need to request permissions from the user. This is only an issue if running on a device with Android 6 or a later version of the OS installed on it. When Android 6 came out, the permissions model changed so that some types of permissions are requested when they are first needed as opposed to when the application is initially installed. Once permissions have been granted, they stay granted so the user is only asked to grant permissions once.

Add the following code to respond to the Find button:

```
public void onScan(View view) {
    if (!ble_scanner.isScanning()) {
        device_count=0;
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
            if (checkSelfPermission(Manifest.permission.ACCESS_COARSE_LOCATION)
                != PackageManager.PERMISSION_GRANTED) {
                permissions_granted = false;
                requestLocationPermission();
            } else {
                Log.i(Constants.TAG, "Location permission has already been granted. Starting
scanning.");
                permissions_granted = true;
            }
        } else {
            // the ACCESS_COARSE_LOCATION permission did not exist before M so....
            permissions_granted = true;
        }
    }
}
```

```

        startScanning();
    } else {
        ble_scanner.stopScanning();
    }
}

```

Note: We haven't written the `startScanning()` method yet so you can safely ignore compilation errors relating to our calls to `startScanning()` for now.

The condition involving `Build.VERSION.SDK_INT` is checking the version of Android we're running on and if it's greater than or equal to "M" (Android 6), a call to the `requestLocationPermission` method (which we have not yet written), is made.

requestLocationPermission method

Add the following code to handle requesting permission to perform scanning from the user:

```

    private void requestLocationPermission() {
        Log.i(Constants.TAG, "Location permission has NOT yet been granted. Requesting permission.");
        if (ActivityCompat.shouldShowRequestPermissionRationale(this, Manifest.permission.ACCESS_COARSE_LOCATION)) {
            Log.i(Constants.TAG, "Displaying location permission rationale to provide additional context.");
            final AlertDialog.Builder builder = new AlertDialog.Builder(this);
            builder.setTitle("Permission Required");
            builder.setMessage("Please grant Location access so this application can perform Bluetooth scanning");
            builder.setPositiveButton(android.R.string.ok, null);
            builder.setOnDismissListener(new DialogInterface.OnDismissListener() {
                public void onDismiss(DialogInterface dialog) {
                    Log.d(Constants.TAG, "Requesting permissions after explanation");
                    ActivityCompat.requestPermissions(MainActivity.this, new String[]{Manifest.permission.ACCESS_COARSE_LOCATION}, REQUEST_LOCATION);
                }
            });
            builder.show();
        } else {
            ActivityCompat.requestPermissions(this, new String[]{Manifest.permission.ACCESS_COARSE_LOCATION}, REQUEST_LOCATION);
        }
    }

    @Override
    public void onRequestPermissionsResult(int requestCode, @NonNull String[] permissions, @NonNull int[] grantResults) {
        if (requestCode == REQUEST_LOCATION) {

```

```

        Log.i(Constants.TAG, "Received response for location permission request.");
        // Check if the only required permission has been granted
        if (grantResults.length == 1 && grantResults[0] == PackageManager.PERMISSION_GRANTED)
        {
            // Location permission has been granted
            Log.i(Constants.TAG, "Location permission has now been granted. Scanning.....");
            permissions_granted = true;
            if (ble_scanner.isScanning()) {
                startScanning();
            }
        }else{
            Log.i(Constants.TAG, "Location permission was NOT granted.");
        }
    } else {
        super.onRequestPermissionsResult(requestCode, permissions, grantResults);
    }
}

private void simpleToast(String message, int duration) {
    toast = Toast.makeText(this, message, duration);
    toast.setGravity(Gravity.CENTER, 0, 0);
    toast.show();
}
}

```

Note: When prompted to import the AlertDialog class, you may find there is more than one class with this name available to you. Make sure you opt to import “android.app.AlertDialog”.

startScanning method

Add the following method to the MainActivity class.

```

private void startScanning() {
    if (permissions_granted) {
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                ble_device_list_adapter.clear();
                ble_device_list_adapter.notifyDataSetChanged();
            }
        });
        simpleToast(Constants.SCANNING, 2000);
        ble_scanner.startScanning(this, SCAN_TIMEOUT);
    } else {
        Log.i(Constants.TAG, "Permission to perform Bluetooth scanning was not yet granted");
    }
}

```

```
}  
  
}
```

As you can see, this method checks that permissions have been granted, clears the UI device list and then tells the BleScanner object to start scanning.

scanningStarted and scanningStopped

The BleScanner object will tell our MainActivity object whenever it starts to perform scanning or stops scanning by calling the corresponding methods of the ScanResultsConsumer interface which MainActivity implements. Complete those methods now:

```
@Override  
public void scanningStarted() {  
    setScanState(true);  
}  
  
@Override  
public void scanningStopped() {  
    if (toast != null) {  
        toast.cancel();  
    }  
    setScanState(false);  
}
```

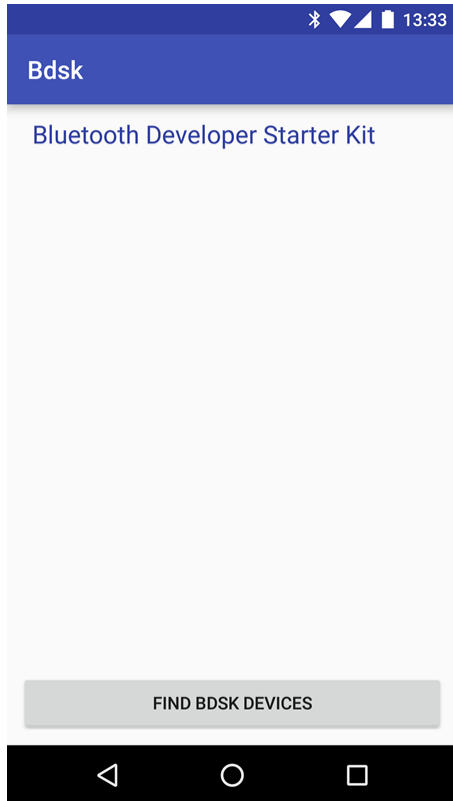
candidateBleDevice

Update the candidateBleDevice method so that any details passed to it by the BleScanner object are stored in the ListAdapter and are shown on the UI if not already there.

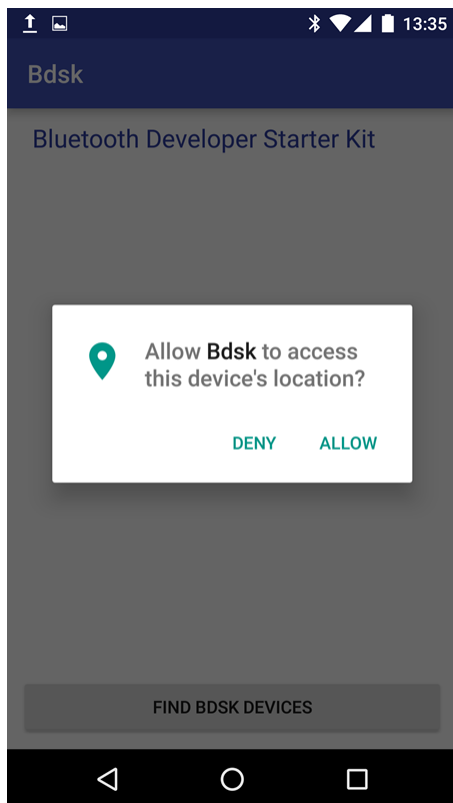
```
@Override  
public void candidateBleDevice(final BluetoothDevice device, byte[] scan_record, int rssi) {  
    runOnUiThread(new Runnable() {  
        @Override  
        public void run() {  
            ble_device_list_adapter.addDevice(device);  
            ble_device_list_adapter.notifyDataSetChanged();  
            device_count++;  
        }  
    });  
}
```

7. Checkpoint: Test Scanning

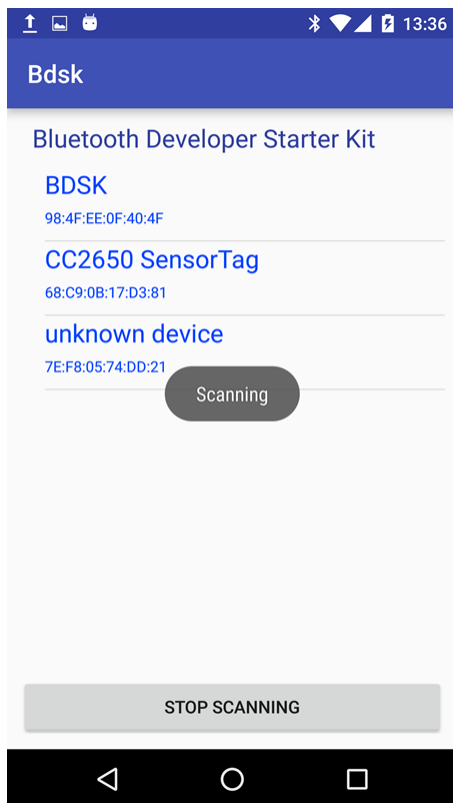
We're now in a position to test the application and verify that it will scan for and list Bluetooth devices it finds. Build and run your application on a physical Android device (not emulator) now. You should see a screen like this appear:



Click the FIND BDSK DEVICES button to initiate scanning. If your device is running Android 6 or later, you should see a dialogue asking if “access to this device’s location” should be allowed or denied.



If you see this dialogue, select ALLOW and then select the FIND... button again. Scanning should now commence and you should see details of all Bluetooth low energy devices which are advertising within range of your Android device listed, perhaps like this:



8. BleScanner - Filtering

At this stage, our application scans for and lists **all** Bluetooth peripherals it finds. Ideally, it would be much better if it only displayed BDSK devices since these are the only devices our application will work with. Luckily, the Android APIs make that easy using the `ScanFilter` class.

Modify the `startScanning` method in `BleScanner` so that it creates a `ScanFilter` object and uses it when scanning, like this:

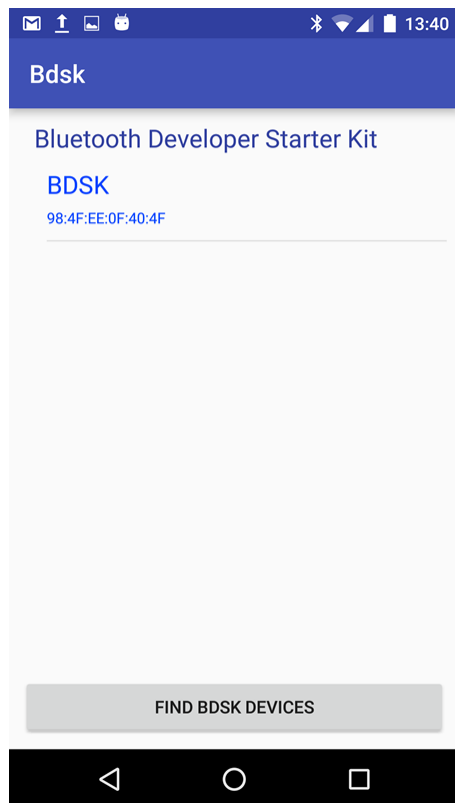
```
List<ScanFilter> filters;
filters = new ArrayList<ScanFilter>();
ScanFilter filter = new ScanFilter.Builder().setDeviceName("BDSK").build();
filters.add(filter);

ScanSettings settings = new
ScanSettings.Builder().setScanMode(ScanSettings.SCAN_MODE_LOW_LATENCY).build();

setScanning(true);
scanner.startScan(filters, settings, scan_callback);
```

The `ScanFilter` object created here, ensures that only devices that are advertising with a device name of "BDSK" will be passed to the `MainActivity` class for listing on the UI.

Build and test again and this time, if your peripheral device is plugged in and advertising correctly, you'll see:



9. Summary of Exercise 1

And that is the first exercise done. To recap, in this exercise we performed these tasks:

1. Created a blank Android project
2. Added a list view and a scan button
3. Implemented the list view functionality
4. Implemented the Bluetooth scanning and filtering functionality and populated the list with relevant devices.

Exercise 2 – Communicating with the Peripheral Device

In the next part of this lab we will add a new screen which will allow the user to interact with the peripheral device over Bluetooth in various ways. We'll implement the new screen as another Android Activity and we'll implement any required Bluetooth operations in an Android service. This is a convenient technique which lessens the chance of our Bluetooth related objects getting lost due to the Android lifecycle terminating or recycling objects.

See <https://developer.android.com/reference/android/app/Service.html> for more information on Android services.

We'll start by adding the Android service with some of the methods it will need to have to support the functionality our new Activity will include.

1. BleAdapterService Class – Part 1

Add a new class called "BleAdapterService" to the com.bluetooth.bdsk.bluetooth package. Over the course of this lab, we'll add methods to it which will act as a high level API for Bluetooth interactions with the peripheral device.

Bluetooth-related objects

To begin with, just replace all the code in the new Service class with the the following code:

```
package com.bluetooth.bdsk.bluetooth;

import android.app.Service;
import android.bluetooth.BluetoothAdapter;
import android.bluetooth.BluetoothDevice;
import android.bluetooth.BluetoothGatt;
import android.bluetooth.BluetoothGattDescriptor;
import android.bluetooth.BluetoothManager;
import android.content.Intent;
import android.content.Context;
import android.os.Handler;
import android.os.Binder;
import android.os.IBinder;
import android.support.annotation.Nullable;
import android.os.Bundle;
import android.os.Message;

public class BleAdapterService extends Service {
    private BluetoothAdapter bluetooth_adapter;
    private BluetoothGatt bluetooth_gatt;
    private BluetoothManager bluetooth_manager;
    private Handler activity_handler = null;
```

```

private BluetoothDevice device;
private BluetoothGattDescriptor descriptor;
private boolean connected = false;
private boolean alarm_playing = false;

public boolean isConnected() {
    return connected;
}
}

```

You will have one or two compilation errors at this point. We'll resolve them shortly.

Creating a BluetoothAdapter Object

To allow Bluetooth operations to be performed, amongst other things, we need a BluetoothAdapter object which can be obtained from a BluetoothManager object. We'll acquire a BluetoothAdapter object when the BleAdapterService object is created. Add the following code to override the service's onCreate method:

```

@Override
public void onCreate() {
    if (bluetooth_manager == null) {
        bluetooth_manager = (BluetoothManager) getSystemService(Context.BLUETOOTH_SERVICE);
        if (bluetooth_manager == null) {
            return;
        }
    }
    bluetooth_adapter = bluetooth_manager.getAdapter();
    if (bluetooth_adapter == null) {
        return;
    }
}
}

```

When we create a BleAdapterService object, this code will execute.

Service Binding

For an Activity to be able to use an Android Service it must be able to "bind" to it. Add the following code directly under the alarm_playing variable declaration:

```

private final IBinder binder = new LocalBinder();
public class LocalBinder extends Binder {
    public BleAdapterService getService() {

```

```

        return BleAdapterService.this;
    }
}

@Override
public IBinder onBind(Intent intent) {
    return binder;
}

@Override
public boolean onUnbind(Intent intent) {
    return super.onUnbind(intent);
}

```

The public LocalBinder object will be used by our new Activity.

Activity and Service Communication

We will use a message Handler object to communicate from the BleAdapterService object to the Activity which uses its services. Add the following constant declarations underneath the alarm_playing variable declaration:

```

// messages sent back to activity
public static final int GATT_CONNECTED = 1;
public static final int GATT_DISCONNECT = 2;
public static final int GATT_SERVICES_DISCOVERED = 3;
public static final int GATT_CHARACTERISTIC_READ = 4;
public static final int GATT_CHARACTERISTIC_WRITTEN = 5;
public static final int GATT_REMOTE_RSSI = 6;
public static final int MESSAGE = 7;
public static final int NOTIFICATION_OR_INDICATION_RECEIVED = 8;

// message parms
public static final String PARCEL_DESCRIPTOR_UUID = "DESCRIPTOR_UUID";
public static final String PARCEL_CHARACTERISTIC_UUID = "CHARACTERISTIC_UUID";
public static final String PARCEL_SERVICE_UUID = "SERVICE_UUID";
public static final String PARCEL_VALUE = "VALUE";
public static final String PARCEL_RSSI = "RSSI";
public static final String PARCEL_TEXT = "TEXT";

```

Add the following method, which will allow an Activity to register the Handler object it will expect events messages to be communicated via:

```

// set activity the will receive the messages

```

```
public void setActivityHandler(Handler handler) {  
    activity_handler = handler;  
}
```

And add the following method, which will allow the service to send text messages to the activity, which we'll display on the screen:

```
private void sendConsoleMessage(String text) {  
    Message msg = Message.obtain(activity_handler, MESSAGE);  
    Bundle data = new Bundle();  
    data.putString(PARCEL_TEXT, text);  
    msg.setData(data);  
    msg.sendToTarget();  
}
```

AndroidManifest.xml Service Entry

The android manifest needs to have our Android service declared in it. Add the following inside the `<application></application>` element:

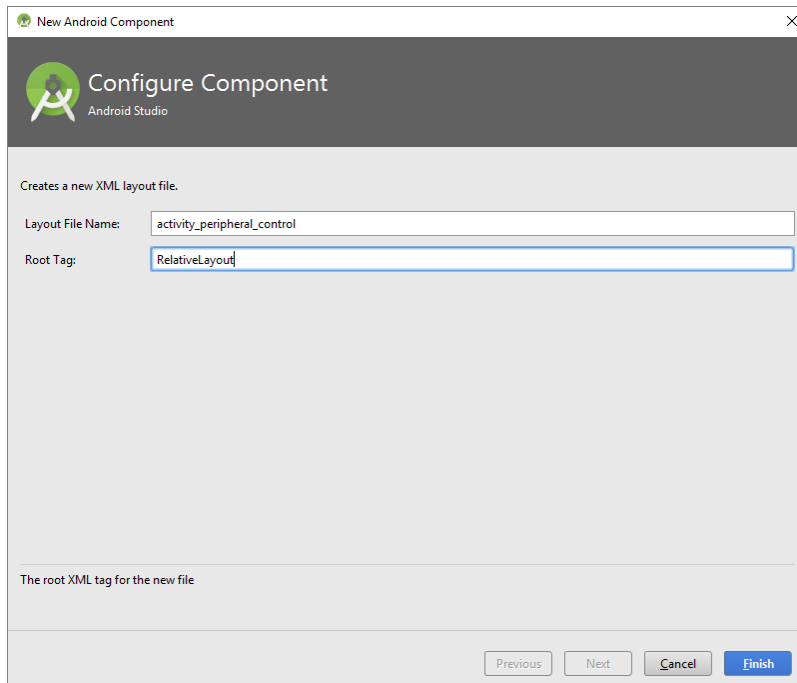
```
<service  
    android:name=".bluetooth.BleAdapterService"  
    android:enabled="true" />  
</application>
```

2. Peripheral Control Screen

Now we'll create a new screen which will serve as the main user interface for our interactions with the peripheral device. This screen will consist of an Android Activity and some XML which defines the screen layout.

activity_peripheral_control.xml

Create a new XML layout file called "activity_peripheral_control.xml" and paste the following XML below into it.



```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/nameTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="15dp"
        android:text="name" />

    <TextView
        android:id="@+id/rssiTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/nameTextView"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="16dp"
        android:text="RSSI" />

    <LinearLayout
        android:id="@+id/rectangle"
```

```

        android:layout_width="200dp"
        android:layout_height="60dp"
        android:layout_below="@+id/rssiTextView"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="21dp"
        android:background="#FF0000"
        android:orientation="vertical"
        android:padding="10dp" >
</LinearLayout>

<Button
    android:id="@+id/lowButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignBottom="@+id/midButton"
    android:layout_marginRight="21dp"
    android:layout_toLeftOf="@+id/midButton"
    android:onClick="onLow"
    android:text="Low" />

<Button
    android:id="@+id/midButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@+id/rectangle"
    android:layout_centerHorizontal="true"
    android:layout_marginTop="21dp"
    android:onClick="onMid"
    android:text="Mid" />

<Button
    android:id="@+id/highButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignBaseline="@+id/midButton"
    android:layout_alignBottom="@+id/midButton"
    android:layout_marginLeft="28dp"
    android:layout_toRightOf="@+id/midButton"
    android:onClick="onHigh"
    android:text="High" />

<Button

```

```

        android:id="@+id/connectButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:layout_centerHorizontal="true"
        android:layout_marginBottom="26dp"
        android:onClick="onConnect"
        android:text="Connect" />

<Button
    android:id="@+id/noiseButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_above="@+id/connectButton"
    android:layout_centerHorizontal="true"
    android:onClick="onNoise"
    android:text="Make a noise" />

<TextView
    android:id="@+id/msgTextView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_above="@+id/noiseButton"
    android:layout_centerHorizontal="true"
    android:layout_marginTop="16dp"
    android:text="" />

<Switch
    android:id="@+id/switch1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignLeft="@+id/rectangle"
    android:layout_alignRight="@+id/rectangle"
    android:layout_below="@+id/lowButton"
    android:layout_marginTop="26dp"
    android:text="Share" />

</RelativeLayout>

```

We've already created the Activity class which goes with this XML, `PeripheralControlActivity`. There's not much code in the class at this stage though and much of what we'll work through in the remainder of

this lab will involve adding code to this Activity and to the BleAdapterService class, whose services our activity will need.

Service Connection

Our new Activity needs to be able to use the BleAdapterService as a kind of Bluetooth API. To do so, it needs to form a “service connection” and when connected to the Android service, obtain an instance of the BleAdapterService for subsequent use. Don’t confuse “Android service” with “Bluetooth GATT service” by the way!

Add the following code to PeripheralControlActivity create a service connection and to handle service connection and disconnection:

```
package com.bluetooth.bdsk.ui;
import android.app.Activity;
import android.content.ComponentName;
import android.content.ServiceConnection;
import android.os.IBinder;
import com.bluetooth.bdsk.bluetooth.BleAdapterService;
public class PeripheralControlActivity extends Activity {
    public static final String EXTRA_NAME = "name";
    public static final String EXTRA_ID = "id";

    private BleAdapterService bluetooth_le_adapter;
    private final ServiceConnection service_connection = new ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName componentName, IBinder service) {
            bluetooth_le_adapter = ((BleAdapterService.LocalBinder) service).getService();
            bluetooth_le_adapter.setActivityHandler(message_handler);
        }
        @Override
        public void onServiceDisconnected(ComponentName componentName) {
            bluetooth_le_adapter = null;
        }
    };
}
```

When the activity connects to the Android service, it obtains an instance of the BleAdapterService object and tells the adapter to use an instance of a Handler class for communication. We haven’t yet created the Handler so let’s do that now.

Creating a Handler for Service to Activity Communication

Add this code to the Activity:

```

@SuppressLint("HandlerLeak")
private Handler message_handler = new Handler() {
    @Override
    public void handleMessage(Message msg) {
        Bundle bundle;
        String service_uuid = "";
        String characteristic_uuid = "";
        byte[] b = null;
        // message handling logic
        switch (msg.what) {
            case BleAdapterService.MESSAGE:
                bundle = msg.getData();
                String text = bundle.getString(BleAdapterService.PARCEL_TEXT);
                showMsg(text);
                break;
        }
    }
};

```

We'll add further case blocks to the switch statement to deal with different types of message passed to the handler from the BleAdapterService later. The MESSAGE type is simply used to receive text to display to the user from the BleAdapterService and will help us monitor activity as we develop the remainder of our code.

onClick Event Handlers

You'll notice in the XML layout file you just created that there are a number of elements which use the onClick property. Let's add empty event handler methods to our Java code to link to these onClick entries. We'll complete them as we progress through the lab. In PeriperalControlActivity.java, add the following:

```

public void onLow(View view) {
}
public void onMid(View view) {
}
public void onHigh(View view) {
}
public void onNoise(View view) {
}

```

onCreate

We need to implement the Activity onCreate method so that we can set the View to the layout XML for this Activity, retrieve parameters passed to the Activity from the previous, MainActivity, initialise the UI and connect to the BleAdapterService Android service.

Here's the code you need to start with, starting with some class variables which you need to add to your declarations section inside the PeripheralControlActivity class:

```
private String device_name;
private String device_address;
private Timer mTimer;
private boolean sound_alarm_on_disconnect = false;
private int alert_level;
private boolean back_requested = false;
private boolean share_with_server = false;
private Switch share_switch;
```

And here's the initial onCreate method plus the showMsg method we've not yet created. We'll add to this method later on, but for now this is all we need:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_peripheral_control);

    // read intent data
    final Intent intent = getIntent();
    device_name = intent.getStringExtra(EXTRA_NAME);
    device_address = intent.getStringExtra(EXTRA_ID);

    // show the device name
    ((TextView) this.findViewById(R.id.nameTextView)).setText("Device : "+device_name+"["+device_address+"]");

    // disable the noise button
    ((Button) PeripheralControlActivity.this.findViewById(R.id.noiseButton))
        .setEnabled(false);

    // disable the LOW/MID/HIGH alert level selection buttons
    ((Button) this.findViewById(R.id.lowButton)).setEnabled(false);
    ((Button) this.findViewById(R.id.midButton)).setEnabled(false);
```

```

        ((Button) this.findViewById(R.id.highButton)).setEnabled(false);

        share_switch = (Switch) this.findViewById(R.id.switch1);
        share_switch.setEnabled(false);
        share_switch.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {
            public void onCheckedChanged(CompoundButton buttonView,
                                         boolean isChecked) {
                // we'll complete this later
            }
        });

        // connect to the Bluetooth adapter service
        Intent gattServiceIntent = new Intent(this, BleAdapterService.class);
        bindService(gattServiceIntent, service_connection, BIND_AUTO_CREATE);
        showMsg("READY");
    }

    private void showMsg(final String msg) {
        Log.d(Constants.TAG, msg);
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                ((TextView) findViewById(R.id.msgTextView)).setText(msg);
            }
        });
    }
}

```

onDestroy

Add the following method which will be automatically called when the Activity is destroyed as part of the standard Android lifecycle:

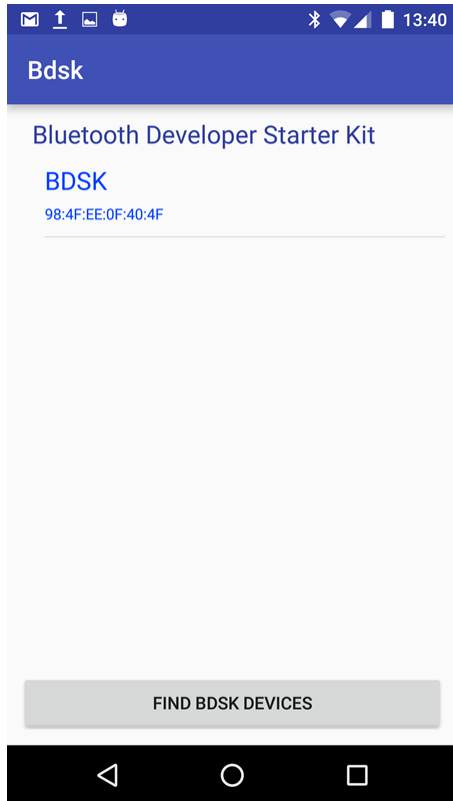
```

@Override
protected void onDestroy() {
    super.onDestroy();
    unbindService(service_connection);
    bluetooth_le_adapter = null;
}

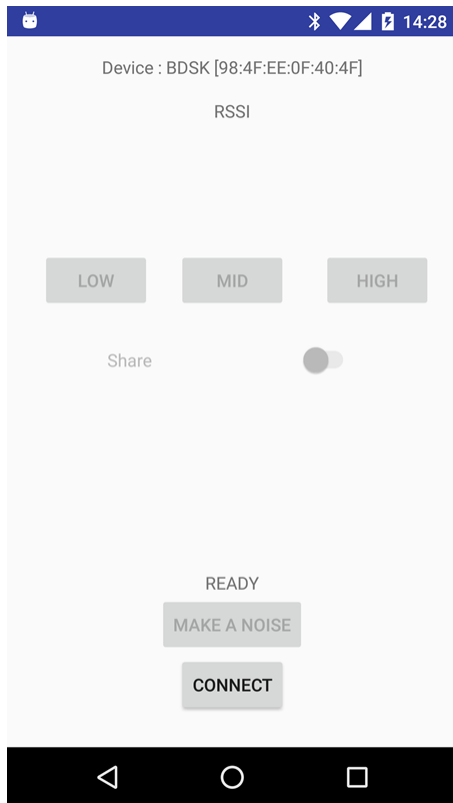
```

Checkpoint: Select a Device

Build and install your application on an Android device. With your peripheral device plugged in and code running on it, launch the Android application and click the FIND BDSK DEVICES button. You should see your device listed with the name “BDSK” and it’s Bluetooth device address listed under it, like this:



Select the listed BDSK device and your application should navigate to the PeripheralControlActivity screen and look very much like this:



3. Connecting to the Peripheral Device

Next, we'll add code to handle the **CONNECT** button being clicked.

connect and disconnect methods

Clicking the **CONNECT** button must cause a Bluetooth connection to be established between the smartphone and the selected peripheral device. All Bluetooth operations, including connecting and disconnecting, are to be implemented in the **BleAdapterService** class.

Add the following methods to the **BleAdapterService** class:

```
// connect to the device
public boolean connect(final String address) {

    if (bluetooth_adapter == null || address == null) {
        sendConsoleMessage("connect: bluetooth_adapter=null");
        return false;
    }

    device = bluetooth_adapter.getRemoteDevice(address);
    if (device == null) {
```

```

        sendConsoleMessage("connect: device=null");
        return false;
    }

    bluetooth_gatt = device.connectGatt(this, false, gatt_callback);
    return true;
}

// disconnect from device
public void disconnect() {
    sendConsoleMessage("disconnecting");
    if (bluetooth_adapter == null || bluetooth_gatt == null) {
        sendConsoleMessage("disconnect: bluetooth_adapter|bluetooth_gatt null");
        return;
    }
    if (bluetooth_gatt != null) {
        bluetooth_gatt.disconnect();
    }
}
}

```

At this point you'll have a compilation error relating to the **gatt_callback** object reference. Let's fix that now.

BluetoothGattCallback

The Bluetooth operations which we'll implement in the `BleAdapterService` class and which will be used by our `PeripheralControlActivity` are all asynchronous operations. This means that after initiating an operation, our code will not block while it waits for the result, but will continue executing in the same thread of execution. Later on (usually milliseconds), after communication over Bluetooth with the peripheral device has completed, we'll receive the result of the operation via a call back. In the Android Bluetooth APIs, to receive such call backs, we need to extend an abstract class called `BluetoothGattCallback` and override any methods which we're interested in.

Add the following code to `BleAdapterService`:

```

private final BluetoothGattCallback gatt_callback = new BluetoothGattCallback() {
    @Override
    public void onConnectionStateChange(BluetoothGatt gatt, int status,
                                        int newState) {
        Log.d(Constants.TAG, "onConnectionStateChange: status=" + status);
        if (newState == BluetoothProfile.STATE_CONNECTED) {
            Log.d(Constants.TAG, "onConnectionStateChange: CONNECTED");
            connected = true;
            Message msg = Message.obtain(activity_handler, GATT_CONNECTED);
            msg.sendToTarget();

```

```

    } else if (newState == BluetoothProfile.STATE_DISCONNECTED) {
        Log.d(Constants.TAG, "onConnectionStateChange: DISCONNECTED");
        connected = false;
        Message msg = Message.obtain(activity_handler, GATT_DISCONNECT);
        msg.sendToTarget();
        if (bluetooth_gatt != null) {
            Log.d(Constants.TAG, "Closing and destroying BluetoothGatt object");
            bluetooth_gatt.close();
            bluetooth_gatt = null;
        }
    }
}
};

```

We've created a sub-class of `BluetoothGattCallback` and we've overridden the `onConnectionStateChange` method. This method will be called whenever we establish a connection to another device or when a connection is broken. In our code, all we do in either of these events is inform the associated Activity using the message handler we set up for communication purposes.

Note: when a connection is lost, it's important to close the `BluetoothGatt` object and ensure it is garbage collected by setting it to null.

The CONNECT button

Add an `onConnect` method to `PeripheralControlActivity` with the following code:

```

public void onConnect(View view) {
    showMsg("onConnect");
    if (bluetooth_le_adapter != null) {
        if (bluetooth_le_adapter.connect(device_address)) {
            ((Button) PeripheralControlActivity.this
                .findViewById(R.id.connectButton)).setEnabled(false);
        } else {
            showMsg("onConnect: failed to connect");
        }
    } else {
        showMsg("onConnect: bluetooth_le_adapter=null");
    }
}
}

```

Now, clicking the CONNECT button will ask the `BleAdapterService` to initiate connecting to the selected peripheral device. Remember, this is an asynchronous operation though. This Activity will be informed when the connection has been established via a message arriving in our handler object.

Connected and Disconnect Events

Let's add some code to handle both connection and disconnection events now. Update your Handler code so that it looks like this:

```
private Handler message_handler = new Handler() {
    @Override
    public void handleMessage(Message msg) {
        Bundle bundle;
        String service_uuid = "";
        String characteristic_uuid = "";
        byte[] b = null;

        switch (msg.what) {
            case BleAdapterService.MESSAGE:
                bundle = msg.getData();
                String text = bundle.getString(BleAdapterService.PARCEL_TEXT);
                showMsg(text);
                break;

            case BleAdapterService.GATT_CONNECTED:
                ((Button) PeripheralControlActivity.this
                    .findViewById(R.id.connectButton)).setEnabled(false);

                // we're connected
                showMsg("CONNECTED");
                break;

            case BleAdapterService.GATT_DISCONNECT:
                ((Button) PeripheralControlActivity.this
                    .findViewById(R.id.connectButton)).setEnabled(true);

                // we're disconnected
                showMsg("DISCONNECTED");
                break;
        }
    }
};
```

Disconnect when back button is pressed

If we're connected to the peripheral device when the user presses the back button, we need to respond by first disconnecting and then allowing the default response to pressing the back button to be taken. Add this function to PeripheralControlActivity. It will be automatically called.

```

public void onBackPressed() {
    Log.d(Constants.TAG, "onBackPressed");
    back_requested = true;
    if (bluetooth_le_adapter.isConnected()) {
        try {
            bluetooth_le_adapter.disconnect();
        } catch (Exception e) {
        }
    } else {
        finish();
    }
}

```

This method actually prevents the user from exiting the current screen as things stand. It requests that we disconnect from Bluetooth but until we receive a message from BleAdapterService to say disconnection has been accomplished, we can't yet exit. Update the handler case for disconnection so that it takes into account the possibility that the user has pressed the back button and completes the process of exiting the current screen:

```

case BleAdapterService.GATT_DISCONNECT:
    ((Button) PeripheralControlActivity.this
        .findViewById(R.id.connectButton)).setEnabled(true);

    // we're disconnected
    showMsg("DISCONNECTED");
    if (back_requested) {
        PeripheralControlActivity.this.finish();
    }
    break;

```

Checkpoint: Connecting to the Peripheral Device

Test the application again but this time, after scanning for and selecting the BDSK device, click the CONNECT button. The CONNECT button should get greyed out and the word "CONNECTED" should appear above it.

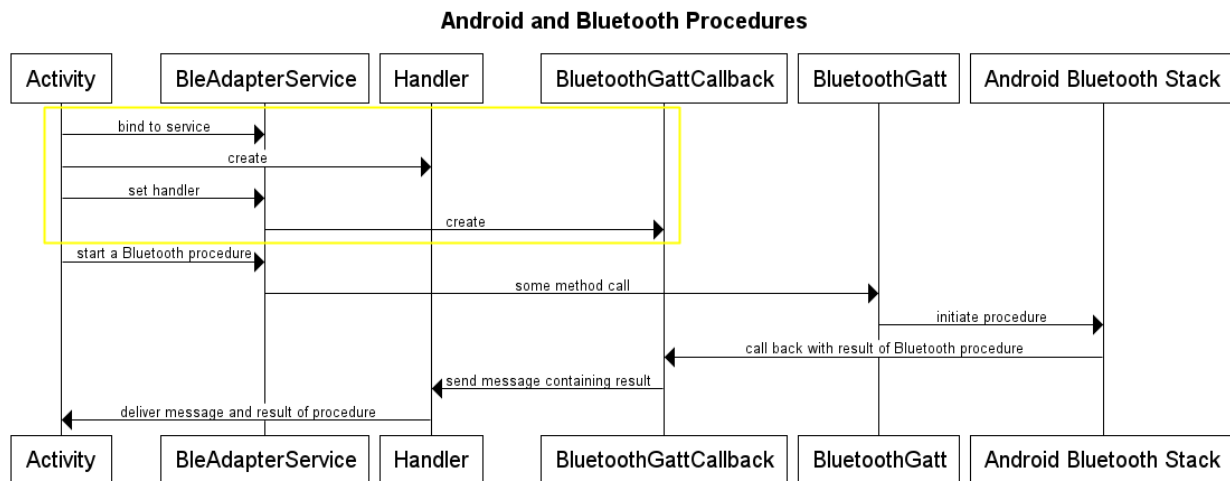
4. Profile Check

Does the device we just connected to implement the expected profile? We can check this, to a certain extent, after connecting to the selected device. A profile specification describes many aspects of a device's required behaviour, including but not limited to the specification of its GATT structure in terms of GATT services, characteristics and descriptors. We can't check all of them programmatically but we can certainly ask the device what services and characteristics it has. For our lab exercise, we'll restrict ourselves to determining what services the device implements and if we find it has the link loss service, immediate alert service, TX power service and our custom proximity monitoring service then we'll

conclude we're connected to a compatible device. To do so, we'll use a technique called "service discovery".

Bluetooth procedures and our generally applicable design pattern

Service discovery is a Bluetooth procedure so we'll follow the same design pattern we used with connect/disconnect event handling. We'll add a public method to the `BleAdapterService` class which can be used by an Activity to initiate the procedure. When the procedure has completed, the `BleAdapterService`'s `BluetoothGattCallback` object will receive a call back and we'll respond to it by passing a message to the Activity using the `Handler` class. This is the general pattern we'll use for all of our Bluetooth procedures. Here it is, depicted as a UML sequence diagram



The interactions bounded by the yellow box are performed once only. After that, we execute a series of Bluetooth procedures such as connecting, discovering services, writing to characteristics and so on and in all cases, essentially the same design pattern applies.

With that in mind, let's proceed with the service discovery code.

Service Discovery

As a first step, add the following constants to `BleAdapterService`. We'll check the services we discover against the service UUIDs defined here and we'll use the characteristic UUIDs later.

```
public static String IMMEDIATE_ALERT_SERVICE_UUID = "00001802-0000-1000-8000-00805F9B34FB";
public static String LINK_LOSS_SERVICE_UUID      = "00001803-0000-1000-8000-00805F9B34FB";
public static String TX_POWER_SERVICE_UUID       = "00001804-0000-1000-8000-00805F9B34FB";
public static String PROXIMITY_MONITORING_SERVICE_UUID = "3E099910-293F-11E4-93BD-AFD0FE6D1DFD";
public static String HEALTH_THERMOMETER_SERVICE_UUID = "00001809-0000-1000-8000-00805F9B34FB";

// service characteristics
public static String ALERT_LEVEL_CHARACTERISTIC    = "00002A06-0000-1000-8000-
```

```

00805F9B34FB";

    public static String CLIENT_PROXIMITY_CHARACTERISTIC = "3E099911-293F-11E4-93BD-
AFD0FE6D1DFD";

    public static String TEMPERATURE_MEASUREMENT_CHARACTERISTIC = "00002A1C-0000-1000-8000-
00805F9B34FB";

    public static String CLIENT_CHARACTERISTIC_CONFIG = "00002902-0000-1000-8000-00805f9b34fb";

```

Now add the following public methods to BleAdapterService:

```

public void discoverServices() {
    if (bluetooth_adapter == null || bluetooth_gatt == null) {
        return;
    }
    Log.d(Constants.TAG, "Discovering GATT services");
    bluetooth_gatt.discoverServices();
}

public List<BluetoothGattService> getSupportedGattServices() {
    if (bluetooth_gatt == null)
        return null;
    return bluetooth_gatt.getServices();
}

```

And override onServicesDiscovered in the BluetoothGattCallback class defined within BleAdapterService:

```

private final BluetoothGattCallback gatt_callback = new BluetoothGattCallback() {
    @Override
    public void onServicesDiscovered(BluetoothGatt gatt, int status) {
        sendConsoleMessage("Services Discovered");
        Message msg = Message.obtain(activity_handler,
            GATT_SERVICES_DISCOVERED);
        msg.sendToTarget();
    }
}

```

In PeripheralControlActivity, update the message handler to include a case in the switch statement for handling the GATT_SERVICES_DISCOVERED message:

```

case BleAdapterService.GATT_SERVICES_DISCOVERED:

    // validate services and if ok....

```

```

        List<BluetoothGattService> slist =
bluetooth_le_adapter.getSupportedGattServices();

        boolean link_loss_present=false;
        boolean immediate_alert_present=false;
        boolean tx_power_present=false;
        boolean proximity_monitoring_present=false;
        boolean health_thermometer_present = false;

        for (BluetoothGattService svc : slist) {
            Log.d(Constants.TAG, "UUID=" + svc.getUuid().toString().toUpperCase() + "
INSTANCE=" + svc.getInstanceId());

            if
(svc.getUuid().toString().equalsIgnoreCase(BleAdapterService.LINK_LOSS_SERVICE_UUID)) {
                link_loss_present = true;
                continue;
            }
            if
(svc.getUuid().toString().equalsIgnoreCase(BleAdapterService.IMMEDIATE_ALERT_SERVICE_UUID)) {
                immediate_alert_present = true;
                continue;
            }
            if
(svc.getUuid().toString().equalsIgnoreCase(BleAdapterService.TX_POWER_SERVICE_UUID)) {
                tx_power_present = true;
                continue;
            }
            if
(svc.getUuid().toString().equalsIgnoreCase(BleAdapterService.PROXIMITY_MONITORING_SERVICE_UUID))
{
                proximity_monitoring_present = true;
                continue;
            }
            if
(svc.getUuid().toString().equalsIgnoreCase(BleAdapterService.HEALTH_THERMOMETER_SERVICE_UUID)) {
                health_thermometer_present = true;
                continue;
            }

        }

        if (link_loss_present && immediate_alert_present && tx_power_present &&
proximity_monitoring_present && health_thermometer_present) {
            showMsg("Device has expected services");

            // show the rssi distance colored rectangle
            ((LinearLayout) PeripheralControlActivity.this

```

```

        .findViewById(R.id.rectangle))
        .setVisibility(View.VISIBLE);
        // enable the LOW/MID/HIGH alert level selection buttons
        ((Button)
PeripheralControlActivity.this.findViewById(R.id.lowButton)).setEnabled(true);
        ((Button)
PeripheralControlActivity.this.findViewById(R.id.midButton)).setEnabled(true);
        ((Button)
PeripheralControlActivity.this.findViewById(R.id.highButton)).setEnabled(true);
    } else {
        showMsg("Device does not have expected GATT services");
    }
    break;

```

and finally, modify the code which handles GATT_CONNECTED messages in the PeripheralControlActivity Handler class so that it initiates the service discovery procedure.

```

case BleAdapterService.GATT_CONNECTED:
    ((Button) PeripheralControlActivity.this
        .findViewById(R.id.connectButton)).setEnabled(false);
    // we're connected
    showMsg("CONNECTED");
    bluetooth_le_adapter.discoverServices();
    break;

```

Checkpoint: check service discovery and profile validation

Build and test on your Android device. After connecting to your peripheral device, you should see the message “Device has expected services” displayed above the MAKE A NOISE button.

5. Setting the Alert Level

Your next job is to implement the ability to set the alert level to a value of 0, 1 or 2 using the three LOW, MID, and HIGH buttons. We’ll want the UI to reflect the selected alert level and of course we need to communicate the selection to the connected peripheral device over Bluetooth. Let’s start with the UI related behaviours.

Enable Buttons Only When Connected

We’ll disable the LOW/MID/HIGH buttons until we have connected over Bluetooth. To achieve this, update the message handler so that GATT_CONNECTED and GATT_DISCONNECT events enable or disable these buttons, as required:

```

private Handler message_handler = new Handler() {
    @Override
    public void handleMessage(Message msg) {
        Bundle bundle;
        String service_uuid = "";
        String characteristic_uuid = "";
        byte[] b = null;

        switch (msg.what) {
            case BleAdapterService.GATT_CONNECTED:
                ((Button) PeripheralControlActivity.this
                    .findViewById(R.id.connectButton)).setEnabled(false);
                // we're connected
                showMsg("CONNECTED");
                // enable the LOW/MID/HIGH alert level selection buttons
                ((Button)
PeripheralControlActivity.this.findViewById(R.id.lowButton)).setEnabled(true);
                ((Button)
PeripheralControlActivity.this.findViewById(R.id.midButton)).setEnabled(true);
                ((Button)
PeripheralControlActivity.this.findViewById(R.id.highButton)).setEnabled(true);
                bluetooth_le_adapter.discoverServices();
                break;

            case BleAdapterService.GATT_DISCONNECT:
                ((Button) PeripheralControlActivity.this
                    .findViewById(R.id.connectButton)).setEnabled(true);
                // we're disconnected
                showMsg("DISCONNECTED");
                // disable the LOW/MID/HIGH alert level selection buttons
                ((Button)
PeripheralControlActivity.this.findViewById(R.id.lowButton)).setEnabled(false);
                ((Button)
PeripheralControlActivity.this.findViewById(R.id.midButton)).setEnabled(false);
                ((Button)
PeripheralControlActivity.this.findViewById(R.id.highButton)).setEnabled(false);
                if (back_requested) {
                    PeripheralControlActivity.this.finish();
                }
                break;

            case BleAdapterService.GATT_REMOTE_RSSI:
                bundle = msg.getData();
                int rssi = bundle.getInt(BleAdapterService.PARCEL_RSSI);

```

```

PeripheralControlActivity.this.updateRssi(rssi);
break;
    }
}
};

```

Highlight Selected Alert Level

When the user clicks one of the LOW, MID or HIGH buttons and the associated characteristic has been successfully updated over Bluetooth, we'll set the colour of the text of selected button and set an internal variable to the selected alert level. Add the following method, which we'll call from various places later:

```

private void setAlertLevel(int alert_level) {
    this.alert_level = alert_level;
    ((Button) this.findViewById(R.id.lowButton)).setTextColor(Color.parseColor("#000000")); ;
    ((Button) this.findViewById(R.id.midButton)).setTextColor(Color.parseColor("#000000")); ;
    ((Button) this.findViewById(R.id.highButton)).setTextColor(Color.parseColor("#000000")); ;
;
    switch (alert_level) {
        case 0:
            ((Button)
this.findViewById(R.id.lowButton)).setTextColor(Color.parseColor("#FF0000")); ;
            break;
        case 1:
            ((Button)
this.findViewById(R.id.midButton)).setTextColor(Color.parseColor("#FF0000")); ;
            break;
        case 2:
            ((Button)
this.findViewById(R.id.highButton)).setTextColor(Color.parseColor("#FF0000")); ;
            break;
    }
}

```

Reading and writing the Alert Level Bluetooth characteristic

During our overall initialisation of the PeripheralControlActivity, we now need to read the Link Loss service's Alert Level characteristic and use the value we obtain to initialise the alert_level variable in the PeripheralControlActivity and set the appropriate LOW/MID/HIGH button to have red text to indicate that is the level which is currently selected. We also need to write to the same characteristic whenever the user clicks one of these buttons.

To support characteristic reads and writes, we'll first add a couple of new public methods to `BleAdapterService` and update the `BluetoothGattCallback` object to pass the result of these operations to the Activity as messages delivered via the Handler, using our usual design pattern.

Add the following two methods to `BleAdapterService`. These are the public methods which the Activity will call to initiate reading from or writing to a characteristic, specified using a service UUID and a characteristic UUID.

```
public boolean readCharacteristic(String serviceUuid,
                                String characteristicUuid) {
    Log.d(Constants.TAG, "readCharacteristic:"+characteristicUuid+" of service "
+serviceUuid);
    if (bluetooth_adapter == null || bluetooth_gatt == null) {
        sendConsoleMessage("readCharacteristic: bluetooth_adapter|bluetooth_gatt null");
        return false;
    }

    BluetoothGattService gattService = bluetooth_gatt
        .getService(java.util.UUID.fromString(serviceUuid));
    if (gattService == null) {
        sendConsoleMessage("readCharacteristic: gattService null");
        return false;
    }
    BluetoothGattCharacteristic gattChar = gattService
        .getCharacteristic(java.util.UUID.fromString(characteristicUuid));
    if (gattChar == null) {
        sendConsoleMessage("readCharacteristic: gattChar null");
        return false;
    }
    return bluetooth_gatt.readCharacteristic(gattChar);
}

public boolean writeCharacteristic(String serviceUuid,
                                   String characteristicUuid, byte[] value) {

    Log.d(Constants.TAG, "writeCharacteristic:"+characteristicUuid+" of service "
+serviceUuid);
    if (bluetooth_adapter == null || bluetooth_gatt == null) {
        sendConsoleMessage("writeCharacteristic: bluetooth_adapter|bluetooth_gatt null");
        return false;
    }

    BluetoothGattService gattService = bluetooth_gatt
```

```

        .getService(java.util.UUID.fromString(serviceUuid));

        if (gattService == null) {
            sendConsoleMessage("writeCharacteristic: gattService null");
            return false;
        }

        BluetoothGattCharacteristic gattChar = gattService
            .getCharacteristic(java.util.UUID.fromString(characteristicUuid));

        if (gattChar == null) {
            sendConsoleMessage("writeCharacteristic: gattChar null");
            return false;
        }

        gattChar.setValue(value);

        return bluetooth_gatt.writeCharacteristic(gattChar);

    }

```

Add the next two methods to the BluetoothGattCallback class defined within BleAdapterService. These methods will be called when Bluetooth read and write procedures have completed and they'll pass results to the Activity using our message handler object.

```

@Override
public void onCharacteristicRead(BluetoothGatt gatt,
                                BluetoothGattCharacteristic characteristic, int status)
{
    if (status == BluetoothGatt.GATT_SUCCESS) {
        Bundle bundle = new Bundle();

        bundle.putString(PARCEL_CHARACTERISTIC_UUID, characteristic.getUuid()
            .toString());

        bundle.putString(PARCEL_SERVICE_UUID,
            characteristic.getService().getUuid().toString());

        bundle.putByteArray(PARCEL_VALUE, characteristic.getValue());

        Message msg = Message.obtain(activity_handler,
            GATT_CHARACTERISTIC_READ);

        msg.setData(bundle);

        msg.sendToTarget();

    } else {
        Log.d(Constants.TAG, "failed to read
            characteristic:"+characteristic.getUuid().toString()+" of service
            "+characteristic.getService().getUuid().toString()+" : status="+status);

        sendConsoleMessage("characteristic read err:"+status);

    }
}

```

```

        public void onCharacteristicWrite(BluetoothGatt gatt,
                                           BluetoothGattCharacteristic characteristic, int status)
        {
            Log.d(Constants.TAG, "onCharacteristicWrite");
            if (status == BluetoothGatt.GATT_SUCCESS) {
                Bundle bundle = new Bundle();
                bundle.putString(PARCEL_CHARACTERISTIC_UUID,
                                characteristic.getUuid().toString());
                bundle.putString(PARCEL_SERVICE_UUID,
                                characteristic.getService().getUuid().toString());
                bundle.putByteArray(PARCEL_VALUE, characteristic.getValue());
                Message msg = Message.obtain(activity_handler, GATT_CHARACTERISTIC_WRITTEN);
                msg.setData(bundle);
                msg.sendToTarget();
            } else {
                sendConsoleMessage("characteristic write err:" + status);
            }
        }
    };

```

Now let's make changes to the PeripheralControlActivity to make use of the new read and write capabilities we've just added to BleAdapterService to acquire the current Link Loss Alert Level value or change it.

After service discovery has completed and we've validated the services on the device, we'll read the alert level characteristic. Change the case statement for handling completion of service discovery to add the highlighted code:

```

        case BleAdapterService.GATT_SERVICES_DISCOVERED:

            // validate services and if ok....
            List<BluetoothGattService> slist =
            bluetooth_le_adapter.getSupportedGattServices();

            boolean link_loss_present=false;
            boolean immediate_alert_present=false;
            boolean tx_power_present=false;
            boolean proximity_monitoring_present=false;

            for (BluetoothGattService svc : slist) {
                Log.d(Constants.TAG, "UUID=" + svc.getUuid().toString().toUpperCase() + "
            INSTANCE=" + svc.getInstanceId());
                if
            (svc.getUuid().toString().equalsIgnoreCase(BleAdapterService.LINK_LOSS_SERVICE_UUID)) {
                    link_loss_present = true;

```

```

        continue;
    }
    if
(svc.getUuid().toString().equalsIgnoreCase(BleAdapterService.IMMEDIATE_ALERT_SERVICE_UUID)) {
        immediate_alert_present = true;
        continue;
    }
    if
(svc.getUuid().toString().equalsIgnoreCase(BleAdapterService.TX_POWER_SERVICE_UUID)) {
        tx_power_present = true;
        continue;
    }
    if
(svc.getUuid().toString().equalsIgnoreCase(BleAdapterService.PROXIMITY_MONITORING_SERVICE_UUID))
{
        proximity_monitoring_present = true;
        continue;
    }
}

    if (link_loss_present && immediate_alert_present && tx_power_present &&
proximity_monitoring_present) {
        showMsg("Device has expected services");

        // show the rssi distance colored rectangle
        ((LinearLayout) PeripheralControlActivity.this
            .findViewById(R.id.rectangle))
            .setVisibility(View.VISIBLE);
        // enable the LOW/MID/HIGH alert level selection buttons
        ((Button)
PeripheralControlActivity.this.findViewById(R.id.lowButton)).setEnabled(true);
        ((Button)
PeripheralControlActivity.this.findViewById(R.id.midButton)).setEnabled(true);
        ((Button)
PeripheralControlActivity.this.findViewById(R.id.highButton)).setEnabled(true);

        bluetooth_le_adapter.readCharacteristic(
            BleAdapterService.LINK_LOSS_SERVICE_UUID,
            BleAdapterService.ALERT_LEVEL_CHARACTERISTIC);
    } else {
        showMsg("Device does not have expected GATT services");
    }
    break;

```

Now, update the message handler in `PeripheralControlActivity` so it handles characteristic read and write messages from `BleAdapterService` by adding the following case blocks to the switch statement.

```
        case BleAdapterService.GATT_CHARACTERISTIC_READ:
            bundle = msg.getData();
            Log.d(Constants.TAG, "Service=" +
                bundle.get(BleAdapterService.PARCEL_SERVICE_UUID).toString().toUpperCase() + " Characteristic=" +
                bundle.get(BleAdapterService.PARCEL_CHARACTERISTIC_UUID).toString().toUpperCase());
            if (bundle.get(BleAdapterService.PARCEL_CHARACTERISTIC_UUID).toString()
                .toUpperCase().equals(BleAdapterService.ALERT_LEVEL_CHARACTERISTIC)
                && bundle.get(BleAdapterService.PARCEL_SERVICE_UUID).toString()
                .toUpperCase().equals(BleAdapterService.LINK_LOSS_SERVICE_UUID)) {
                b = bundle.getByteArray(BleAdapterService.PARCEL_VALUE);
                if (b.length > 0) {
                    PeripheralControlActivity.this.setAlertLevel((int) b[0]);
                }
            }
            break;
        case BleAdapterService.GATT_CHARACTERISTIC_WRITTEN:
            bundle = msg.getData();
            if (bundle.get(BleAdapterService.PARCEL_CHARACTERISTIC_UUID).toString()
                .toUpperCase().equals(BleAdapterService.ALERT_LEVEL_CHARACTERISTIC)
                && bundle.get(BleAdapterService.PARCEL_SERVICE_UUID).toString()
                .toUpperCase().equals(BleAdapterService.LINK_LOSS_SERVICE_UUID)) {
                b = bundle.getByteArray(BleAdapterService.PARCEL_VALUE);
            }
            break;
```

Add the following constants to the `Constants` class:

```
public static final byte [] ALERT_LEVEL_LOW = { (byte) 0x00};
public static final byte [] ALERT_LEVEL_MID = { (byte) 0x01};
public static final byte [] ALERT_LEVEL_HIGH = { (byte) 0x02};
```

And replace your `onLow`, `onMid` and `onHigh` methods with the following code, which as you can see, uses our new `BleAdapterService` methods to initiate writing to the alert level characteristic of the link loss service:

```
public void onLow(View view) {
    bluetooth_le_adapter.writeCharacteristic(
        BleAdapterService.LINK_LOSS_SERVICE_UUID,
        BleAdapterService.ALERT_LEVEL_CHARACTERISTIC, Constants.ALERT_LEVEL_LOW
```

```

    );
}

public void onMid(View view) {
    bluetooth_le_adapter.writeCharacteristic(
        BleAdapterService.LINK_LOSS_SERVICE_UUID,
        BleAdapterService.ALERT_LEVEL_CHARACTERISTIC, Constants.ALERT_LEVEL_MID
    );
}

public void onHigh(View view) {
    bluetooth_le_adapter.writeCharacteristic(
        BleAdapterService.LINK_LOSS_SERVICE_UUID,
        BleAdapterService.ALERT_LEVEL_CHARACTERISTIC, Constants.ALERT_LEVEL_HIGH
    );
}

```

Finally, when our message handler is informed that the write operation completed successfully, we need to update the UI to indicate the new alert_level selection. Change the GATT_CHARACTERISTIC_WRITTEN case in the handler switch statement to make a call to setAlertLevel.

```

case BleAdapterService.GATT_CHARACTERISTIC_WRITTEN:
    bundle = msg.getData();
    if (bundle.get(BleAdapterService.PARCEL_CHARACTERISTIC_UUID).toString()
        .toUpperCase().equals(BleAdapterService.ALERT_LEVEL_CHARACTERISTIC)
        && bundle.get(BleAdapterService.PARCEL_SERVICE_UUID).toString()
        .toUpperCase().equals(BleAdapterService.LINK_LOSS_SERVICE_UUID)) {
        b = bundle.getByteArray(BleAdapterService.PARCEL_VALUE);
        if (b.length > 0) {
            PeripheralControlActivity.this.setAlertLevel((int) b[0]);
        }
    }
    break;

```

Checkpoint: Test setting the Alert Level

Build, install and test. Try the following sequence of tests:

1. Scan, select and connect, select LOW.
 - o The LOW button should become highlighted in red to indicate the current alert level has been set to “LOW” (0). If you have the appropriate circuit connected to your peripheral device you should see the green LED flash by way of acknowledgement.

2. Go back and disconnect. Select and connect again.
 - o The LOW button should be highlighted red to indicate the current alert level read back from the peripheral device is “LOW” (0).
3. Select MID.
 - o The MID button should be highlighted in red to indicate the current alert level been set to “MID” (1). If you have the appropriate circuit connected to your peripheral device you should see the yellow LED flash by way of acknowledgement.
4. Go back and disconnect. Select and connect again.
 - o The MID button should be highlighted red to indicate the current alert level read back from the peripheral device is “MID” (1).
5. Scan, select and connect, select HIGH.
 - o The HIGH button should be highlighted in red to indicate the current alert level read back from the peripheral device is “HIGH” (2). If you have the appropriate circuit connected to your peripheral device you should see the red LED flash by way of acknowledgement.
6. Go back and disconnect. Select and connect again.
7. The HIGH button should be highlighted red to indicate the current alert level read back from the peripheral device is “HIGH” (2).

6. Monitoring Signal Strength

Next we'll initiate sampling the signal strength periodically and we'll classify the signal strength as high, medium or low and draw a coloured block in yellow, green or red accordingly. Note that the signal strength is termed the RSSI or Received Signal Strength Indicator.

readRemoteRssi

First we'll add some code to the **BleAdapterService** class:

```
public void readRemoteRssi() {  
    if (bluetooth_adapter == null || bluetooth_gatt == null) {  
        return;  
    }  
    bluetooth_gatt.readRemoteRssi();  
}
```

The Activity will call this method.

onReadRemoteRssi

Next let's add code to our BluetoothGattCallback object which handles receiving RSSI values:

```

@Override
public void onReadRemoteRssi(BluetoothGatt gatt, int rssi, int status) {
    if (status == BluetoothGatt.GATT_SUCCESS) {
        sendConsoleMessage("RSSI read OK");
        Bundle bundle = new Bundle();
        bundle.putInt(PARCEL_RSSI, rssi);
        Message msg = Message
            .obtain(activity_handler, GATT_REMOTE_RSSI);
        msg.setData(bundle);
        msg.sendToTarget();
    } else {
        sendConsoleMessage("RSSI read err:"+status);
    }
}

```

Receiving RSSI values

Note that the sampled RSSI value is sent to our Activity using the message handler so our next job is to change the PeripheralControlActivity to receive and use messages of this type. We also need to initiate sampling the RSSI value periodically, using a timer when we have successfully connected to the peripheral device.

Update the message handler switch statement in PeripheralControlActivity as shown:

```

case BleAdapterService.GATT_CHARACTERISTIC_READ:
    bundle = msg.getData();
    Log.d(Constants.TAG, "Service=" +
bundle.get(BleAdapterService.PARCEL_SERVICE_UUID).toString().toUpperCase() + " Characteristic=" +
bundle.get(BleAdapterService.PARCEL_CHARACTERISTIC_UUID).toString().toUpperCase());
    if (bundle.get(BleAdapterService.PARCEL_CHARACTERISTIC_UUID).toString()
        .equals(BleAdapterService.ALERT_LEVEL_CHARACTERISTIC)
        && bundle.get(BleAdapterService.PARCEL_SERVICE_UUID).toString()
        .equals(BleAdapterService.LINK_LOSS_SERVICE_UUID)) {
        b = bundle.getByteArray(BleAdapterService.PARCEL_VALUE);
        if (b.length > 0) {
            PeripheralControlActivity.this.setAlertLevel((int) b[0]);
            // show the rssi distance colored rectangle
            ((LinearLayout) PeripheralControlActivity.this
                .findViewById(R.id.rectangle))
                .setVisibility(View.VISIBLE);
            // start off the rssi reading timer
            startReadRssiTimer();
        }
    }
}

```



```

        }
        break;
        case BleAdapterService.GATT_REMOTE_RSSI:
            bundle = msg.getData();
            int rssi = bundle.getInt(BleAdapterService.PARCEL_RSSI);
            PeripheralControlActivity.this.updateRssi(rssi);
            break;
    }
}
};

```

Here, we've made the RSSI rectangle visible and started the RSSI polling timer after our initialisation read of the Link Loss service's alert level characteristic has completed. We've also added a case block to handle RSSI values passed from the BleAdapterService.

RSSI and the UI

In the onCreate method, before the code which disables the noise button, add the following code to initialise the RSSI category indicator rectangle:

```

// hide the coloured rectangle used to show green/amber/red rssi
// distance
((LinearLayout) this.findViewById(R.id.rectangle))
    .setVisibility(View.INVISIBLE);

```

Add the following methods:

```

private void startReadRssiTimer() {
    mTimer = new Timer();
    mTimer.schedule(new TimerTask() {
        @Override
        public void run() {
            bluetooth_le_adapter.readRemoteRssi();
        }
    }, 0, 2000);
}

private void stopTimer() {
    if (mTimer != null) {
        mTimer.cancel();
        mTimer = null;
    }
}

```

```

private void updateRssi(int rssi) {
    ((TextView) findViewById(R.id.rssiTextView)).setText("RSSI = "
        + Integer.toString(rssi));
    LinearLayout layout = ((LinearLayout) PeripheralControlActivity.this
        .findViewById(R.id.rectangle));
    byte proximity_band = 3;
    if (rssi < -80) {
        layout.setBackgroundColor(0xFFFF0000);
    } else if (rssi < -50) {
        layout.setBackgroundColor(0xFFFF8A01);
        proximity_band = 2;
    } else {
        layout.setBackgroundColor(0xFF00FF00);
        proximity_band = 1;
    }
    layout.invalidate();
}

```

These methods initiate regular sampling of the RSSI, stop the sampling process and update the UI with RSSI values, respectively.

Add a call to `stopTimer()` to the `onDestroy()` method and when handling a `GATT_DISCONNECT` event. Also, make the RSSI rectangle invisible when disconnected.

```

@Override
protected void onDestroy() {
    super.onDestroy();
    stopTimer();
    unbindService(service_connection);
    bluetooth_le_adapter = null;
}

```

```

case BleAdapterService.GATT_DISCONNECT:
    ((Button) PeripheralControlActivity.this
        .findViewById(R.id.connectButton)).setEnabled(true);
    // we're disconnected
    showMsg("DISCONNECTED");
    // hide the rssi distance colored rectangle
    ((LinearLayout) PeripheralControlActivity.this
        .findViewById(R.id.rectangle))
        .setVisibility(View.INVISIBLE);

```

```

        // disable the LOW/MID/HIGH alert level selection buttons
        ((Button)
PeripheralControlActivity.this.findViewById(R.id.lowButton)).setEnabled(false);

        ((Button)
PeripheralControlActivity.this.findViewById(R.id.midButton)).setEnabled(false);

        ((Button)
PeripheralControlActivity.this.findViewById(R.id.highButton)).setEnabled(false);

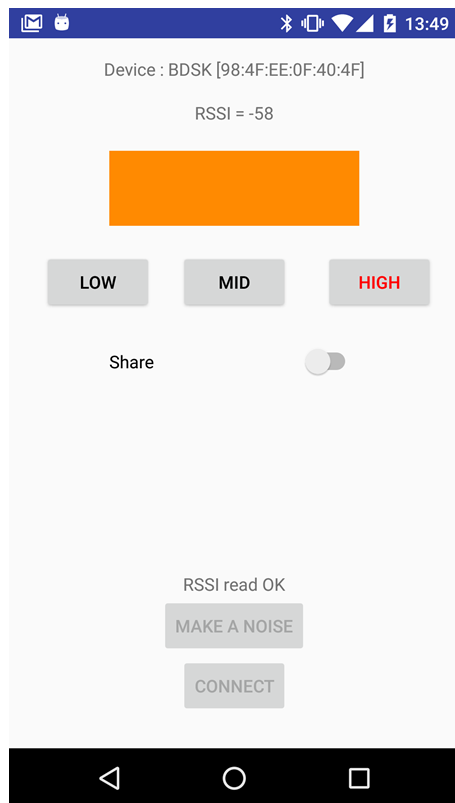
        // stop the rssi reading timer
        stopTimer();

        if (back_requested) {
            PeripheralControlActivity.this.finish();
        }

```

Checkpoint: Test RSSI monitoring

Build and test your application. After connecting to the peripheral device you should see RSSI values displayed near the top of the screen and a coloured rectangle appear. Try moving nearer or further away from the peripheral device and watch the RSSI value change and occasionally, the rectangle change colour.



7. Immediate Alert

When the user clicks the “MAKE A NOISE” button, we need to write the current value of our `alert_level` variable to the Alert Level characteristic which belongs to the Immediate Alert service. This will cause our peripheral device to flash a light of a colour which corresponds to the selected alert level and make a noise if the alert level is greater than zero. If we write a value of zero then if there’s already an alert in progress it should be silenced.

Luckily, `BleAdapterService` already has the ability to write to characteristics and deliver the result to our Activity so we don’t have any work to do there.

Enable the Make a Noise button

When connected, we need the MAKE A NOISE button to be enabled. Modify the `GATT_CONNECTED` message handler case in `PeripheralControlActivity` as shown:

```
case BleAdapterService.GATT_CONNECTED:
    ((Button) PeripheralControlActivity.this
        .findViewById(R.id.connectButton)).setEnabled(false);
    ((Button) PeripheralControlActivity.this
        .findViewById(R.id.lowButton)).setEnabled(true);
    ((Button) PeripheralControlActivity.this
        .findViewById(R.id.midButton)).setEnabled(true);
    ((Button) PeripheralControlActivity.this
        .findViewById(R.id.highButton)).setEnabled(true);
    ((Button) PeripheralControlActivity.this.findViewById(R.id.noiseButton))
        .setEnabled(true);
    // we're connected
    showMsg("CONNECTED");
    bluetooth_le_adapter.discoverServices();
    break;
```

Triggering an immediate alert

Now all we need to do is write to the Immediate Alert service’s Alert Level characteristic whenever the MAKE A NOISE button is clicked. Replace the `onNoise` method with the following to achieve this:

```
public void onNoise(View view) {
    byte [] al = new byte[1];
    al[0] = (byte) alert_level;
    bluetooth_le_adapter.writeCharacteristic(
        BleAdapterService.IMMEDIATE_ALERT_SERVICE_UUID,
        BleAdapterService.ALERT_LEVEL_CHARACTERISTIC, al
    );
}
```

Checkpoint: Test Immediate Alert

Build, install and run your application. After connecting to the peripheral device, select LOW and then click the MAKE A NOISE button. Repeat this test but preceding MAKE A NOISE with selecting MID and then HIGH. All lights on the peripheral device's attached circuit will flash, an increasing number of times depending on which alert level has been selected. With MID or HIGH selected, you should also hear the buzzer make a noise.

8. Sharing Proximity Data

Our UI has a switch labelled "Share". When this switch is set to the "on" position, our application must send RSSI values and a number which classifies distance from the peripheral device (1=near, 2=middle distance, 3=far away) to the peripheral device by writing to the Client Proximity characteristic of the Proximity Monitoring service. Let's implement that now.

The Sharing Switch

Earlier on, in the onCreate method, we created a Switch object. Complete it's implementation now, by adding the highlighted code:

```
share_switch = (Switch) this.findViewById(R.id.switch1);
share_switch.setEnabled(false);
share_switch.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {
    public void onCheckedChanged(CompoundButton buttonView,
                                boolean isChecked) {
        if (bluetooth_le_adapter != null) {
            share_with_server = isChecked;
            if (!isChecked && bluetooth_le_adapter.isConnected()) {
                showMsg("Switched off sharing proximity data");
                // write 0,0 to cause peripheral device to switch off all LEDs
                if (bluetooth_le_adapter.writeCharacteristic(
                    BleAdapterService.PROXIMITY_MONITORING_SERVICE_UUID,
                    BleAdapterService.CLIENT_PROXIMITY_CHARACTERISTIC,
                    new byte[] { 0 , 0 })) {
                } else {
                    showMsg("Failed to inform peripheral device sharing has been
disabled");
                }
            }
        }
    }
});
```

Enable the share switch when we have a connection and disable it when disconnected by updating the message handler case blocks as shown:

```
case BleAdapterService.GATT_CONNECTED:
    ((Button) PeripheralControlActivity.this
        .findViewById(R.id.connectButton)).setEnabled(false);
    ((Button) PeripheralControlActivity.this.findViewById(R.id.noiseButton))
        .setEnabled(true);
    share_switch.setEnabled(true);
    // we're connected
    showMsg("CONNECTED");
    bluetooth_le_adapter.discoverServices();
    break;

case BleAdapterService.GATT_DISCONNECT:
    ((Button) PeripheralControlActivity.this
        .findViewById(R.id.connectButton)).setEnabled(true);
    // we're disconnected
    showMsg("DISCONNECTED");
    // hide the rssi distance colored rectangle
    ((LinearLayout) PeripheralControlActivity.this
        .findViewById(R.id.rectangle))
        .setVisibility(View.INVISIBLE);
    share_switch.setEnabled(false);
    // disable the LOW/MID/HIGH alert level selection buttons
    ((Button)
PeripheralControlActivity.this.findViewById(R.id.lowButton)).setEnabled(false);
    ((Button)
PeripheralControlActivity.this.findViewById(R.id.midButton)).setEnabled(false);
    ((Button)
PeripheralControlActivity.this.findViewById(R.id.highButton)).setEnabled(false);
    // stop the rssi reading timer
    stopTimer();
    if (back_requested) {
        PeripheralControlActivity.this.finish();
    }
    break;
```

Writing Proximity Data to the Peripheral Device

Our final step is to write proximity data, each time it's available to us, to the peripheral device Client Proximity characteristic. Update the updateRssi method as shown:

```

private void updateRssi(int rssi) {
    ((TextView) findViewById(R.id.rssiTextView)).setText("RSSI = "
        + Integer.toString(rssi));
    LinearLayout layout = ((LinearLayout) PeripheralControlActivity.this
        .findViewById(R.id.rectangle));

    byte proximity_band = 3;
    if (rssi < -80) {
        layout.setBackgroundColor(0xFFFF0000);
    } else if (rssi < -50) {
        layout.setBackgroundColor(0xFFFF8A01);
        proximity_band = 2;
    } else {
        layout.setBackgroundColor(0xFF00FF00);
        proximity_band = 1;
    }
    layout.invalidate();

    if (share_with_server) {
        if (bluetooth_le_adapter.writeCharacteristic(
            BleAdapterService.PROXIMITY_MONITORING_SERVICE_UUID,
            BleAdapterService.CLIENT_PROXIMITY_CHARACTERISTIC,
            new byte[] { proximity_band, (byte) rssi })) {
            showMsg("proximity data shared: proximity_band:" + proximity_band + ",rssi:" +
rssi);
        } else {
            showMsg("Failed to share proximity data");
        }
    }
}

```

Checkpoint: Test proximity sharing

Test by toggling the Share switch after connecting to the peripheral device. With sharing enabled you should see RSSI values displayed on the LCD display or the peripheral device's console, depending on what you used for your peripheral device. The LED of the same colour as our UI's RSSI rectangle should also be lit.



9. Link Loss

The penultimate part of this lab is to implement the link loss procedure. When our Bluetooth connection is lost, we need to have the smartphone sound an alarm if the `alert_level` variable has a value of greater than zero.

AlarmManager

We'll implement all of the code which deals with making or stopping sound in a new singleton class called `AlarmManager`. Create a new package called "com.bluetooth.bdsk.audio" and then create the new class and copy the following code into it:

```
package com.bluetooth.bdsk.audio;

import android.content.res.AssetFileDescriptor;
import android.media.MediaPlayer;
import android.util.Log;

import com.bluetooth.bdsk.Constants;
import com.bluetooth.bdsk.R;

import java.io.IOException;

public class AlarmManager {

    private static AlarmManager instance;
    public MediaPlayer mp;

    private AlarmManager() {
    }

    public static synchronized AlarmManager getInstance() {
        if (instance == null) {
            instance = new AlarmManager();
        }
        return instance;
    }

    public boolean alarmIsSounding() {
        boolean already_playing=false;
        if (mp != null) {
            try {
                already_playing = mp.isPlaying();
            }
        }
    }
}
```

```

        } catch (IllegalStateException e) {
            // means we're not playing since this exception is thrown is isPlaying() is
            // called before MediaPlayer has been initialised or after it has been released
        }
    }
    return already_playing;
}

public void soundAlarm(AssetFileDescriptor afd) {
    Log.d(Constants.TAG, "playSound");
    if (!alarmIsSounding()) {
        mp = new MediaPlayer();
        mp.setOnCompletionListener(new MediaPlayer.OnCompletionListener() {
            @Override
            public void onCompletion(MediaPlayer mp) {
                Log.d(Constants.TAG, "playSound - alarm playback has completed");
                mp.reset();
                mp.release();
            }
        });
        try {
            mp.setDataSource(afd.getFileDescriptor(), afd.getStartOffset(),
                afd.getLength());
            mp.prepare();
            mp.start();
            afd.close();
        } catch (IllegalArgumentException e) {
            Log.d(Constants.TAG, "playSound - IllegalArgumentException: " + e.getMessage());
        } catch (IllegalStateException e) {
            Log.d(Constants.TAG, "playSound - IllegalStateException: " + e.getMessage());
        } catch (IOException e) {
            Log.d(Constants.TAG, "playSound - IOException: " + e.getMessage());
        }
    } else {
        Log.d(Constants.TAG, "playSound - already playing alarm");
    }
}

public void stopAlarm() {
    if (mp != null && mp.isPlaying()) {
        mp.stop();
        mp.reset();
    }
}

```

```

        mp.release();
    }
}

```

This class uses Android’s MediaPlayer to play a sound file which we’ll provide in a call from our Activity. We’ve also included methods which let us stop an alarm and one to check whether or not we’re already sounding the alarm.

Adding an audio file to the project

In the “source” folder of the Android lab, you’ll find a file called “alarm.wav”. Create a directory called “raw” in your Android Studio’s app\src\main\res folder and copy the alarm.wav file into it. You should see the new file reflected in the explorer view of Android Studio.

Sounding the alarm on link loss

Now let’s use the AlarmManager class from our Activity.

In PeripheralControlActivity, update the case block for GATT_DISCONNECT to sound the alarm if the user has set the alarm level to greater than zero:

```

        case BleAdapterService.GATT_DISCONNECT:
            ((Button) PeripheralControlActivity.this
                .findViewById(R.id.connectButton)).setEnabled(true);
            // we're disconnected
            showMsg("DISCONNECTED");
            // hide the rssi distance colored rectangle
            ((LinearLayout) PeripheralControlActivity.this
                .findViewById(R.id.rectangle))
                .setVisibility(View.INVISIBLE);
            share_switch.setEnabled(false);
            // disable the LOW/MID/HIGH alert level selection buttons
            ((Button)
PeripheralControlActivity.this.findViewById(R.id.lowButton)).setEnabled(false);
            ((Button)
PeripheralControlActivity.this.findViewById(R.id.midButton)).setEnabled(false);
            ((Button)
PeripheralControlActivity.this.findViewById(R.id.highButton)).setEnabled(false);
            // stop the rssi reading timer
            stopTimer();
            if (alert_level > 0) {
                AlarmManager.getInstance().soundAlarm(getResources().openRawResourceFd(R.raw.alarm));
            }

```

```
        if (back_requested) {  
            PeripheralControlActivity.this.finish();  
        }  
        break;
```

Stop alarm on reconnect

If we're playing an alarm when the user reconnects, we want to stop the alarm. To achieve this, update the case block for GATT_CONNECTED as shown:

```
        case BleAdapterService.GATT_CONNECTED:  
            ((Button) PeripheralControlActivity.this  
                .findViewById(R.id.connectButton)).setEnabled(false);  
            ((Button) PeripheralControlActivity.this.findViewById(R.id.noiseButton))  
                .setEnabled(true);  
            share_switch.setEnabled(true);  
            // we're connected  
            showMsg("CONNECTED");  
            AlarmManager am = AlarmManager.getInstance();  
            Log.d(Constants.TAG, "alarmIsSounding=" + am.alarmIsSounding());  
            if (am.alarmIsSounding()) {  
                Log.d(Constants.TAG, "Stopping alarm");  
                am.stopAlarm();  
            }  
            bluetooth_le_adapter.discoverServices();  
            break;
```

10. Temperature Monitoring

The user needs to be able to enable or disable temperature monitoring, which will exploit Bluetooth indications sent from the Temperature Measurement characteristic when enabled. Check the LE Basic Theory document if you need to refresh your memory about notifications and indications.

From an Android point of view, notifications and indications are dealt with in *almost* exactly the same way. There's no need to acknowledge indications. The Android stack will do that for you. The only noticable difference in your code concerns the value you write to the remote device's client characteristic configuration descriptor to switch indications on rather than notifications. Android provides predefined constants for you to use though, so that's easy. You'll see this shortly.

UI Changes

Let's start by adding the following code to the activity_peripheral_control.xml file.

```
<Switch
    android:id="@+id/switch1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignLeft="@+id/rectangle"
    android:layout_alignRight="@+id/rectangle"
    android:layout_below="@+id/lowButton"
    android:layout_marginTop="26dp"
    android:text="Share" />

<Switch
    android:id="@+id/switch2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignLeft="@+id/rectangle"
    android:layout_alignRight="@+id/rectangle"
    android:layout_below="@+id/switch1"
    android:layout_marginTop="26dp"
    android:text="Monitor Temperature" />

<TextView
    android:id="@+id/temperatureLabel"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@+id/switch2"
    android:layout_centerHorizontal="true"
    android:layout_marginTop="60dp"
    android:text="Temperature"
```

```

        android:textStyle="bold"/>

        <TextView
            android:id="@+id/temperatureValue"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_below="@+id/temperatureLabel"
            android:layout_centerHorizontal="true"
            android:layout_marginTop="10dp"
            android:text="not available"
            android:textStyle="bold"
            android:textColor="#FF0000"/>
    </RelativeLayout>

```

Switch Initialisation

Next, we'll add some code to the PeripheralControlActivity class to initialise the new switch. Make modifications as shown in the next two code fragments:

```

private Switch share_switch;
private Switch temperature_switch;
private BleAdapterService bluetooth_le_adapter;

```

and in the onCreate method....

```

share_switch = (Switch) this.findViewById(R.id.switch1);
share_switch.setEnabled(false);
share_switch.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {
    public void onCheckedChanged(CompoundButton buttonView,
                                boolean isChecked) {
        if (bluetooth_le_adapter != null) {
            share_with_server = isChecked;
            if (!isChecked && bluetooth_le_adapter.isConnected()) {
                showMsg("Switched off sharing proximity data");
                // write 0,0 to cause the peripheral to switch off all LEDs
                if (bluetooth_le_adapter.writeCharacteristic(
                    BleAdapterService.PROXIMITY_MONITORING_SERVICE_UUID,
                    BleAdapterService.CLIENT_PROXIMITY_CHARACTERISTIC,
                    new byte[]{0, 0})) {
                } else {
                    showMsg("Failed to inform peripheral sharing has been disabled");
                }
            }
        }
    }
});

```

```

    }
    }
}

});

    temperature_switch = (Switch) this.findViewById(R.id.switch2);
    temperature_switch.setEnabled(false);
    temperature_switch.setOnCheckedChangeListener(new
CompoundButton.OnCheckedChangeListener() {
        public void onCheckedChanged(CompoundButton buttonView,
                                    boolean isChecked) {
            }
    });

```

We'll complete the onCheckedChanged method soon.

Connection State Changes

When a Bluetooth connection has been established, we need to enable the new switch. If the connection drops, we need to disable it. Make the following changes to accomplish this:

```

    case BleAdapterService.GATT_CONNECTED:
        ((Button) PeripheralControlActivity.this
            .findViewById(R.id.connectButton)).setEnabled(false);
        ((Button) PeripheralControlActivity.this.findViewById(R.id.noiseButton))
            .setEnabled(true);
        share_switch.setEnabled(true);
        temperature_switch.setEnabled(true);
        // we're connected
        showMsg("CONNECTED");
        AlarmManager am = AlarmManager.getInstance();
        Log.d(Constants.TAG, "alarmIsSounding=" + am.alarmIsSounding());
        if (am.alarmIsSounding()) {
            Log.d(Constants.TAG, "Stopping alarm");
            am.stopAlarm();
        }
        bluetooth_le_adapter.discoverServices();
        break;

    case BleAdapterService.GATT_DISCONNECT:
        ((Button) PeripheralControlActivity.this
            .findViewById(R.id.connectButton)).setEnabled(true);

```

```

        // we're disconnected
        showMsg("DISCONNECTED");
        // hide the rssi distance colored rectangle
        ((LinearLayout) PeripheralControlActivity.this
            .findViewById(R.id.rectangle))
            .setVisibility(View.INVISIBLE);
        share_switch.setEnabled(false);
        temperature_switch.setEnabled(false);
        temperature_switch.setChecked(false);
        // disable the LOW/MID/HIGH alert level selection buttons
        ((Button)
PeripheralControlActivity.this.findViewById(R.id.lowButton)).setEnabled(false);
        ((Button)
PeripheralControlActivity.this.findViewById(R.id.midButton)).setEnabled(false);
        ((Button)
PeripheralControlActivity.this.findViewById(R.id.highButton)).setEnabled(false);
        // stop the rssi reading timer
        stopTimer();
        if (alert_level > 0) {
AlarmManager.getInstance().soundAlarm(getResources().openRawResourceFd(R.raw.alarm));
        }
        if (back_requested) {
            PeripheralControlActivity.this.finish();
        }
        break;

```

Temperature Measurements

Temperature measurements, received from the connected peripheral device will be displayed. They could be in either Celsius or Fahrenheit, so we'll need to display a name for the units used. Update the Constants class as shown in preparation for this:

```

package com.bluetooth.bdsk;

public class Constants {
    public static final String TAG = "bdsk";
    public static final String FIND = "Find BDSK Devices";
    public static final String STOP_SCANNING = "Stop Scanning";
    public static final String SCANNING = "Scanning";
    public static final byte [] ALERT_LEVEL_LOW = { (byte) 0x00};
    public static final byte [] ALERT_LEVEL_MID = { (byte) 0x01};
    public static final byte [] ALERT_LEVEL_HIGH = { (byte) 0x02};
    public static final String [] TEMPERATURE_UNITS = {"Celsius", "Fahrenheit"};

```



```
}
```

If the user switches temperature monitoring on, we need to subscribe to indications from the Temperature Measurement characteristic and handle them as they are received. If the user switches temperature monitoring off, we need to unsubscribe.

Add the following method to the `BleAdapterService` class:

```
public boolean setIndicationsState(String serviceUuid, String characteristicUuid, boolean
enabled) {

    if (bluetooth_adapter == null || bluetooth_gatt == null) {
        sendConsoleMessage("setIndicationsState: bluetooth_adapter|bluetooth_gatt null");
        return false;
    }

    BluetoothGattService gattService =
bluetooth_gatt.getService(java.util.UUID.fromString(serviceUuid));
    if (gattService == null) {
        sendConsoleMessage("setIndicationsState: gattService null");
        return false;
    }

    BluetoothGattCharacteristic gattChar =
gattService.getCharacteristic(java.util.UUID.fromString(characteristicUuid));
    if (gattChar == null) {
        sendConsoleMessage("setIndicationsState: gattChar null");
        return false;
    }

    bluetooth_gatt.setCharacteristicNotification(gattChar, enabled);
    // Enable remote notifications
    descriptor = gattChar.getDescriptor(UUID.fromString(CLIENT_CHARACTERISTIC_CONFIG));

    if (enabled) {
        descriptor.setValue(BluetoothGattDescriptor.ENABLE_INDICATION_VALUE);
    } else {
        descriptor.setValue(BluetoothGattDescriptor.DISABLE_NOTIFICATION_VALUE);
    }

    boolean ok = bluetooth_gatt.writeDescriptor(descriptor);

    return ok;
}
```

This method allows indications to be enabled or disabled for a specified characteristic belonging to a specified service. This involves two primary Android API calls, the first on the characteristic object (`setCharacteristicNotification`) and the second to write to the characteristic's associated client characteristic configuration descriptor. It is the latter step which impacts the remote device.

We also need a callback method which will be invoked by the system when a notification or indication message is received over the Bluetooth connection from the remote device. In that method, we'll pass whatever we receive to our `PeripheralControlActivity` class via the `Handler`. Let's add that method in `BleAdapterService` now:

```
@Override
public void onCharacteristicChanged(BluetoothGatt gatt,
                                   BluetoothGattCharacteristic characteristic) {
    Bundle bundle = new Bundle();
    bundle.putString(PARCEL_CHARACTERISTIC_UUID, characteristic.getUuid().toString());
    bundle.putString(PARCEL_SERVICE_UUID,
characteristic.getService().getUuid().toString());
    bundle.putByteArray(PARCEL_VALUE, characteristic.getValue());
    // notifications and indications are both communicated from here in this way
    Message msg = Message.obtain(activity_handler, NOTIFICATION_OR_INDICATION_RECEIVED);
    msg.setData(bundle);
    msg.sendToTarget();
}
```

And of course we need to update the `Handler` to process this case:

```
case BleAdapterService.NOTIFICATION_OR_INDICATION_RECEIVED:
    bundle = msg.getData();
    service_uuid = bundle.getString(BleAdapterService.PARCEL_SERVICE_UUID);
    characteristic_uuid = bundle.getString(BleAdapterService.PARCEL_CHARACTERISTIC_UUID);
    b = bundle.getByteArray(BleAdapterService.PARCEL_VALUE);
    Log.d(Constants.TAG, byteArrayAsHexString(b));
    if (characteristic_uuid
.equalsIgnoreCase((BleAdapterService.TEMPERATURE_MEASUREMENT_CHARACTERISTIC.toString()))) {
        if (b.length == 5) {
            int units = b[0]; // 0=Celsius, 1=Fahrenheit
            // temperatures are in a 4 byte array in little endian format (IEEE 11073 32-bit
FLOAT)
            int mantissa = 0;
            byte[] mantissa_bytes = { 0x00, 0x00, 0x00, 0x00 };
            mantissa_bytes[1] = b[3];
            mantissa_bytes[2] = b[2];
```

```

        mantissa_bytes[3] = b[1];

        // propagate sign bit of most significant byte of the 3 mantissa bytes
        if ((mantissa_bytes[1] & (byte) 0x80) == (byte) 0x80) {
            // sign bit is set so...
            mantissa_bytes[0] = (byte) 0xff;
        }

        // now convert to a standard signed int
        mantissa = java.nio.ByteBuffer.wrap(mantissa_bytes).getInt();

        int exponent = b[4];
        Log.d(Constants.TAG, "mantissa=" + mantissa);
        Log.d(Constants.TAG, "exponent=" + exponent);
        int temperature = (int) (mantissa * Math.pow(10, exponent) * 10);
        double temp = temperature / 10.0;
        Log.d(Constants.TAG, "temperature=" + temp);
        showTemperature(temp, units);
    } else {
        showMsg("Invalid temperature measurement indication received. Length wrong");
        return;
    }
}

```

Add the showTemperature function which is called by the code we just added to the handler:

```

private void showTemperature(final double temperature, final int units) {
    Log.d(Constants.TAG, temperature+" "+Constants.TEMPERATURE_UNITS[units]);
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            ((TextView) findViewById(R.id.temperatureValue)).setText(temperature+"
"+Constants.TEMPERATURE_UNITS[units]);
        }
    });
}

```

Now complete the code associated with the temperature monitoring switch, to either enable or disable indications:

```

        temperature_switch.setOnCheckedChangeListener(new
        CompoundButton.OnCheckedChangeListener() {
            public void onCheckedChanged(CompoundButton buttonView,

```

```

        boolean isChecked) {

        if (bluetooth_le_adapter != null && bluetooth_le_adapter.isConnected()) {
            if (!isChecked) {
                showMsg("Switching off temperature monitoring");
                if (bluetooth_le_adapter.setIndicationsState(
                    BleAdapterService.HEALTH_THERMOMETER_SERVICE_UUID,
                    BleAdapterService.TEMPERATURE_MEASUREMENT_CHARACTERISTIC,
                    false)) {
                    clearTemperature();
                } else {
                    showMsg("Failed to inform temperature monitoring has been disabled");
                }
            } else {
                showMsg("Switching on temperature monitoring");
                if (bluetooth_le_adapter.setIndicationsState(
                    BleAdapterService.HEALTH_THERMOMETER_SERVICE_UUID,
                    BleAdapterService.TEMPERATURE_MEASUREMENT_CHARACTERISTIC,
                    true)) {
                } else {
                    showMsg("Failed to inform temperature monitoring has been enabled");
                }
            }
        }
    }
}

});

```

And add the function we call in the previous fragment to reset the contents of the temperature field in the UI:

```

private void clearTemperature() {
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            ((TextView) findViewById(R.id.temperatureValue)).setText("not available");
        }
    });
}

```

And that should be all we need to do to add temperature monitoring to our Android application! So let's turn our attention to testing next.

Test your Application

You're now ready to take your Android app for a test drive! Here's a summary of the expected behaviours for you to check when testing:

Feature	Expected Behaviour
Scan Button	finds your BDSK device which should have the name "BDSK".
Connect	connects to the peripheral device
Select any of the Low, Mid or High buttons	Write 0, 1 or 2 to the Link Loss service Alert Level characteristic in the peripheral device. The corresponding LED (0=green, 1=yellow, 2=red) on your board should flash 4 times as an acknowledgement.
Make a Noise	The Alert Level characteristic of the Immediate Alert service should be written to using the value last selected by pressing one of Low, Mid or High. If the alert level is 1 or 2, the board should respond by beeping 5 times and flashing all three LEDs in unison. If 0 then the LEDs should flash but the buzzer should be silent.
Sharing ON	The board's LED of the same colour as the proximity rectangle on the UI should be illuminated. The buzzer will be silent. As you move the phone away or towards the peripheral device, as the rectangular proximity indicator changes value, the illuminated LED should change value after a short delay.
Sharing OFF	All LEDs should be switched off unless enabled by some other action.
Link Lost / Disconnected	The phone and board should make a noise for an extended period of time and all LEDs should flash.
Temperature monitoring on	Application should subscribe to temperature measurement indications (the act of subscribing may be visible in the peripheral device's logs). Having subscribed, temperature measurement values should appear on the screen at a rate of one per second.
Temperature monitoring off	Application should unsubscribe from temperature measurement indications (the act of unsubscribing may be visible in the peripheral device's logs) and temperature values should stop being received.

Well done! By following this hands-on lab you have created an application that scans for Bluetooth Peripheral devices, can establish a connection to a selected device, examines the GATT services it supports, reads and writes characteristic values, works with indications and monitors the RSSI value.

Project Structure

If you've had problems along the way and suspect you may have created project classes, interfaces or other resource types in the wrong place in your project, here's what it should look like by the time you get to this point:

