



Developer Study Guide: An introduction to Bluetooth Low Energy Development

Creating a Bluetooth Low Energy application using the Apache Cordova SDK

Version: 5.1.0

Last updated: 28th March 2019

Contents

REVISION HISTORY	4
INTRODUCTION.....	6
Overview	6
Terminology Reminder	7
Functional Requirements	8
Apache Cordova Plugins	9
GETTING STARTED	9
Create the BDSK Project	9
Add Platform Support	10
Add Plugins	10
Install the Bluetooth LE plugin	11
Install the Device plugin.....	11
Install the Media plugin	11
Copy the starter source code into place	12
Build	13
Install.....	13
Notes.....	13
Remote Debugging	14
Remote Debugging with Google Chrome	14
Remote Debugging with Safari	17
The Bluetooth Low Energy API	18
Cordova Code Patterns	18
EXERCISE 1 - SCANNING FOR PERIPHERAL DEVICES.....	19
1. Scanning and Scan Results Screen	19
2. Device Discovery	20
3. Checkpoint: Test Scanning	21
4. Device Filtering	23
	2

5. Checkpoint: Test Device Filtering	24
6. Summary of Exercise 1	24
EXERCISE 2 – COMMUNICATING WITH THE PERIPHERAL.....	25
1. Selecting a device	25
2. Checkpoint: Select the BDSK Device	25
3. Connecting to the peripheral	26
4. connectToDevice skeleton code	27
5. onConnected	28
6. onDisconnected	29
7. Disconnect when back button pressed	32
8. Checkpoint: Connecting and Disconnecting	32
9. Acquiring the current Alert Level	33
10. Set the Alert Level	35
11. Checkpoint: Test getting and setting the Alert Level	35
12. Monitoring Signal Strength	36
13. Checkpoint: Test RSSI monitoring	39
14. Immediate Alert	39
15. Checkpoint: Test Immediate Alert	40
16. Sharing Proximity Data	40
17. Checkpoint: Test proximity sharing	42
18. Link Loss	44
19. Checkpoint: Test the Mobile Application Response to Link Loss	45
20. Temperature Monitoring	45
TEST YOUR APPLICATION	48

Revision History

Version	Date	Author	Changes
1.0	1 st May 2013	Matchbox	Initial version
2.0	3 rd 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 server 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	Added a new lab, with associated solution source code, which explains how to use the Apache Cordova SDK to create a cross platform mobile application which uses Bluetooth Low Energy.
5.0.0	19 th December 2017	Martin Woolley, Bluetooth SIG	Added support for temperature monitoring using 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”
5.1.0	28 th March 2019	Martin Woolley, Bluetooth SIG	Added Zephyr on micro:bit server lab. Deprecated Arduino server lab.

Introduction

Overview

This document is a step-by-step guide to creating a Bluetooth application for iOS or Android using the Apache Cordova SDK. The application is intended to be used with the Bluetooth peripheral device created in one of the server labs of this study guide and so ideally, you should complete one of those labs first.

Apache Cordova is an SDK which allows developers to create mobile applications using HTML, CSS and JavaScript and to then generate an installable build for a particular target platform, all from the same source code. This is a long way of saying that it lets you code once but build runtime versions for many, different platforms. We assume you will be working with an Android device, an iOS device or, if you're lucky, both.

In this lab we will not cover any basics of programming with web technologies. It's assumed that you have at least a basic understanding of HTML, CSS and JavaScript. Ideally you will also have an appreciation of how to use JavaScript with the Document Object Model (DOM). If these topics are completely new to you, it's recommended that you first review a basic HTML/CSS/JavaScript tutorial from somewhere on the web first. That said, the lab will guide you at every step of the way and you'll be starting out with a skeleton implementation, which will take care of all of the HTML and CSS that we need, as well as some of the JavaScript, so the learning curve should be pretty shallow if you have no experience of these technologies.

The lab, tasks and code that you'll create are designed to provide instruction in the principles and practice of developing Apache Cordova applications which use Bluetooth Low Energy technology.

Objectives

This lab provides instructions to achieve the following:

- 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

- Apache Cordova - see <https://cordova.apache.org/> for downloads and installation instructions
- Android Studio if you intend to build for Android or XCode if you intend to build for iOS

Equipment Requirements

- USB cable
- For Android: Android tablet or smartphone running Android 5.1 or better
- For iOS: an iPhone or iPad. See <https://cordova.apache.org/docs/en/latest/guide/platforms/ios/index.html> for version requirements.
- Peripheral device running the solution to one of the server labs which is part of this study guide and connected to the associated circuit. See the server lab documentation for details.

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 JavaScript function or if the change to be made is a relatively small change to an existing block of code, the whole block will be presented, with the code to be added or modified highlighted in **this colour**.

Troubleshooting

If you encounter problems as you proceed, check the preceeding lab steps carefully to ensure you have followed the instructions correctly.

The lab will explain how to use a web browser, specifically Chrome or Sarafi to trace execution of your code and watch for run-time errors. This is a really useful technique which will greatly assist with the solving of problems.

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 a BBC micro:bit 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 lab 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 (which is presumed to be the peripheral used during the server lab).	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 Immediate Alert Service’s Alert Level characteristic.
5	Proximity Monitoring	The connected smartphone application will track its distance from the peripheral 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 by writing to the Client Proximity characteristic of the custom Proximity Monitoring service.
7	Link Lost / Disconnected	If the connection between smartphone and peripheral 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.

Apache Cordova Plugins

We'll be using a number of Apache Cordova plugins in this lab. The most important one provides a JavaScript API with which to implement Bluetooth Low Energy (LE) GAP central mode operations. If the term "GAP central" is not familiar to you, please invest some time in reading the LE Basic Theory guide, which you'll find packaged elsewhere in the Bluetooth LE Developer Study Guide.

Getting Started

It is assumed that you have installed Apache Cordova on your PC, along with Android Studio and/or Xcode.

Apache Cordova works on Windows, Mac and Linux. Use of the SDK is almost identical across these platforms. The only place where your use of the SDK will vary, is when we come to building our application, either for Android, on any platform that the Android Studio is available for, or for iOS on a Mac.

To complete the set up of your development environment, complete the following steps, working from within a command shell. I'll illustrate the steps involved using Windows. If you're working on Mac OS or Linux, watch out for file permissions issues and prefix Cordova commands with "sudo" where necessary.

Create the BDSK Project

From within a suitable working folder, execute the following "cordova create" command to create a new Apache Cordova project to work within:

```
cordova create bdsk com.bluetooth.cordova.bdsk Bdsk
```

Figure 1 - command to create the new Bdsk Cordova project

```
C:\my_projects>
C:\my_projects>cordova create bdsk com.bluetooth.cordova.bdsk Bdsk
Creating a new cordova project.
C:\my_projects>
```

Figure 2 - Creating the new project

All Cordova SDK commands start with "cordova" and are followed by a series of parameters:

- bdsK - the name of the folder which will form the root of our project
- com.bluetooth.cordova.bdsK - a required package name or namespace
- BdsK - the name of the application

Add Platform Support

cd into the bdsK folder which is the root of your new Apache Cordova project.

```
C:\my_projects>cd bdsK
C:\my_projects\bdsK>
```

If you're curious, list the directory contents to get an idea of what a Apache Cordova project looks like after it's first created.

```
C:\my_projects\bdsK>dir
Volume in drive C has no label.
Volume Serial Number is C4AF-BE52

Directory of C:\my_projects\bdsK

07/07/2017  14:00    <DIR>          .
07/07/2017  14:00    <DIR>          ..
07/07/2017  14:00                987 config.xml
07/07/2017  14:00    <DIR>          hooks
07/07/2017  14:00    <DIR>          platforms
07/07/2017  14:00    <DIR>          plugins
07/07/2017  14:00    <DIR>          www
                   1 File(s)          987 bytes
                   6 Dir(s)  145,062,957,056 bytes free

C:\my_projects\bdsK>
```

Next, if you wish to build your application for Android, execute the following:

```
cordova platform add android
```

If you're working on a Mac and wish to build for iOS, execute the following instead:

```
sudo cordova platform add ios
```

Add Plugins

Apache Cordova projects make use of "plugins", JavaScript APIs which give access to platform features and capabilities not supported fully by standard JavaScript, as used in a web browser. We'll be using

three plugins in our project, plus one more if you're building for iOS. Execute the following commands to install the plugins we need.

Install the Bluetooth LE plugin

```
cordova plugin add cordova-plugin-ble-central --variable BLUETOOTH_USAGE_DESCRIPTION="Bluetooth is used to allow the exchange of data between phone/tablet and peripheral"
```

This plugin provides an API with which to scan for and connect to LE devices and then perform GATT operations like reading and writing to characteristics.

See <https://github.com/don/cordova-plugin-ble-central> for full details of this plugin.

Install the Device plugin

```
cordova plugin add cordova-plugin-device
```

The device plugin allows you to find out various details of the device the application is executing on. We need that so we can accommodate a difference between iOS and Android when it comes to playing an audio file to indicate that the Bluetooth connection has dropped (Feature 7 in our requirements list).

See <https://cordova.apache.org/docs/en/latest/reference/cordova-plugin-device/index.html> for full details of this plugin.

Install the Media plugin

```
cordova plugin add cordova-plugin-media
```

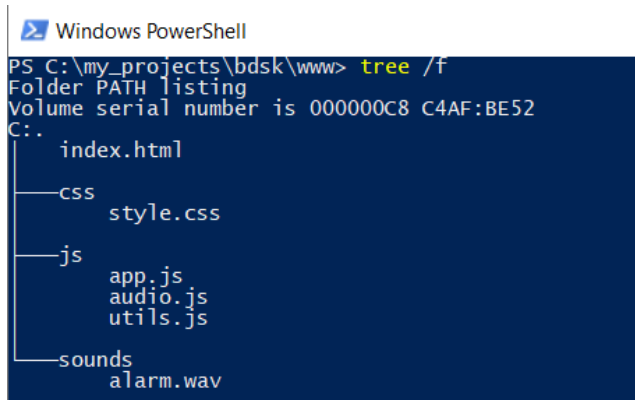
This plugin has various capabilities relating to multimedia. We'll be using it to play or stop playback of an audio file whenever the Bluetooth link drops.

Copy the starter source code into place

Rather than code every line from scratch, we'll be solely concerned with implementing the Bluetooth related features and to make life easier, will be starting out with an incomplete implementation of the lab application, with some gaps in the code which we'll need to fill as we progress.

Delete the contents of your bdsk\www folder so that the default code which was generated for you when Cordova created the project, is cleared.

Now, copy the contents of Apache Cordova\starter\bdsk\www in your BDSK package, into your project's bdsk\www folder. Your www folder should now look like this:



```
Windows PowerShell
PS C:\my_projects\bdsk\www> tree /f
Folder PATH listing
Volume serial number is 000000C8 C4AF:BE52
C:.
├── index.html
├── css
│   └── style.css
├── js
│   ├── app.js
│   ├── audio.js
│   └── utils.js
└── sounds
    └── alarm.wav
```

Figure 3 - Project www folder with starter project files copied into place

Your project consists of the following files in various states of completion:

File	Description
index.html	Defines the user interface using the popular Single-Page Application (SPA) pattern. See https://en.wikipedia.org/wiki/Single-page_application . You will not be required to change this file.
css/style.css	Contains a style sheet which defines the “look and feel” of the UI. You will not be required to change this file.
js\app.js	Contains JavaScript functions which implement the primary functionality of the BDSK application. Many of the functions are incomplete. The lab exercises will involve modifying this file and completing the JavaScript functions so that the various requirements of the application are implemented, one step at a time. Places in this file where we will make changes are prefixed by a comment line which starts with “TODO”.
js\audio.js	Contains functions which support playing back and stopping the playback of an audio file.

	You will not be required to change this file.
js\utils.js	Contains various “utility functions”. You will not be required to change this file.
sounds\alarm.wav	An audio recording of an alarm sound which we will use in the BDSK application to indicate link loss. You will not be required to change this file.

Open each of the text files in your favourite editor and have a quick look at the code, just to get orientated. Don't change anything yet. We'll systematically implement one function at a time in the exercises that follow.

Let's begin by checking exactly what state the starter app is in before we change it in Exercise 1. Build the application and install it. The following instructions indicate the commands you will use for these steps. We'll be doing this repeatedly throughout the exercises and you should be able to recall them using your shell's command history buffer. It will be assumed from now on, that if the lab requires you to build and install the application, you'll know what to do. The following instructions will not be repeated in this lab document again.

Build

Android	iOS
cordova build android	sudo cordova build ios

Install

Android	iOS
adb install -r platforms\android\build\outputs\apk\android- debug.apk	Open the Xcode project which the build process has created for you from the platforms/ios folder. Build / run in Xcode with a USB-connected iOS device as the target.

Notes

If you run into permissions problems when building on Mac OS, set all file permissions in your bdsk project using a command similar to the following. Substitute your own user and group names for your machine.

```
sudo chown -R martin:staff *
```

Launch the application on your smartphone. It should look like the screen shot in Figure 4.

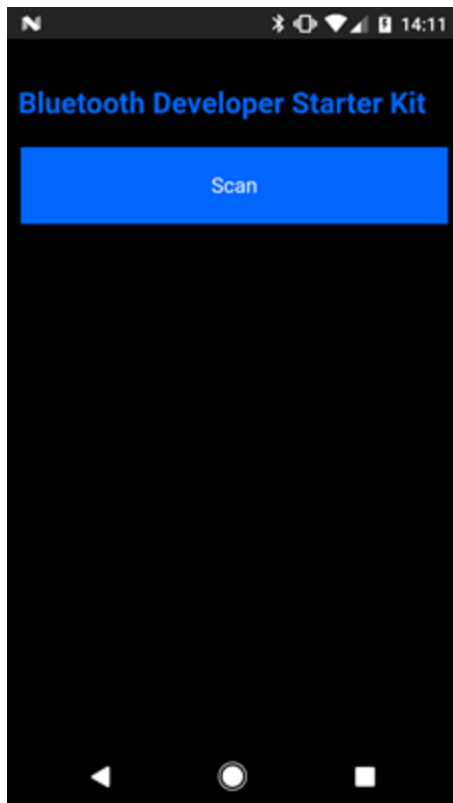


Figure 4 - The device discovery screen

Select the Scan button. Nothing should happen because we haven't yet implemented the associated JavaScript function.

Remote Debugging

In fact *something* should happen. It just has no effect on the UI, so you can't see it at this stage. Each of the functions in the app.js file announces itself by writing a message to the JavaScript console. We can see these messages by using a remote debugging technique, supported by the Google Chrome browser, which is available for Windows, Mac OS and Linux or by Safari on Mac OS. This is a really useful facility and it's recommended that you use it throughout these exercises, to monitor execution and to help solve problems.

With your test device (smartphone or tablet) connected to your computer over USB, proceed as described in one of the next two sections, according to the browser you are using. Note that Android devices must be in "developer mode" with "USB debugging" enabled in the Settings\Developer options screen. You'll find information on this topic elsewhere on the internet.

Remote Debugging with Google Chrome

With your test device plugged into your computer over USB and the BDSK application running:

Enter "chrome://inspect" in the Chrome browser address bar and press Enter. You should see a web page similar to Figure 5.

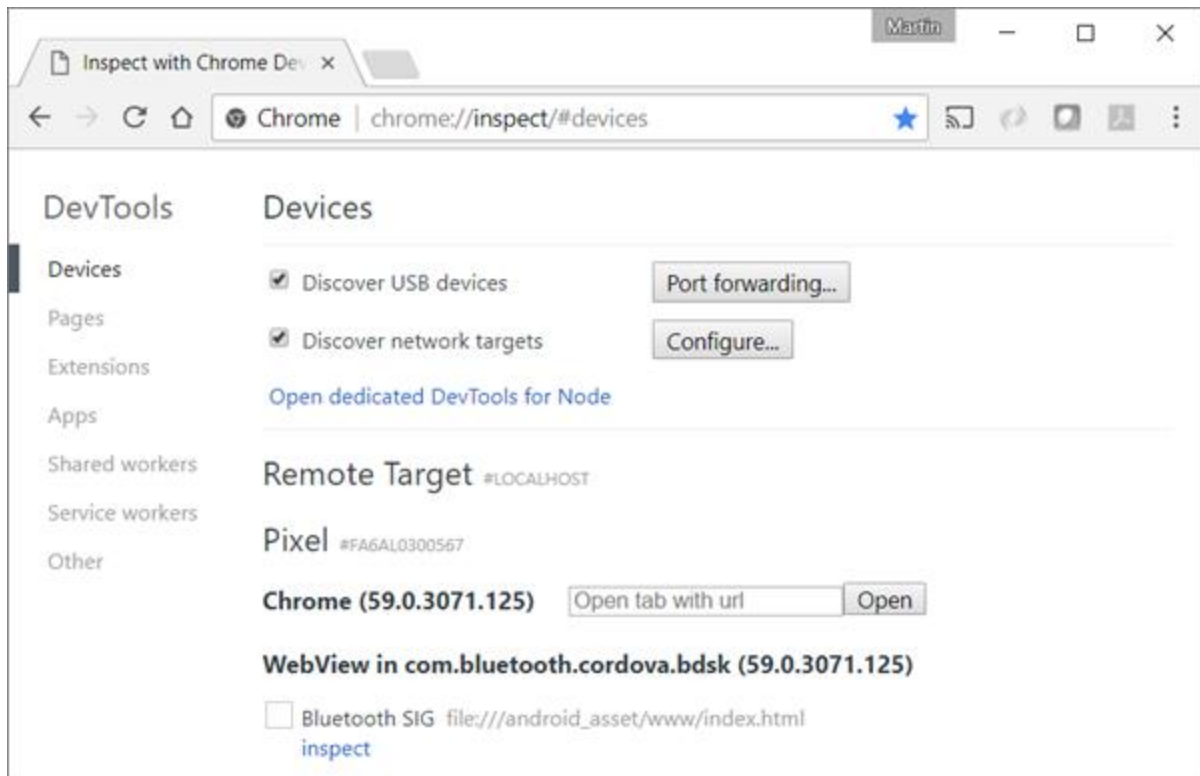


Figure 5 - Chrome and the Inspect Devices Page

Click the “inspect” link under “WebView in com.bluetooth.cordova.bdsk” and you’ll be presented with the remote debugging page, with the screen of your connected device echoed on the left hand side of the page and the JavaScript console on the right.

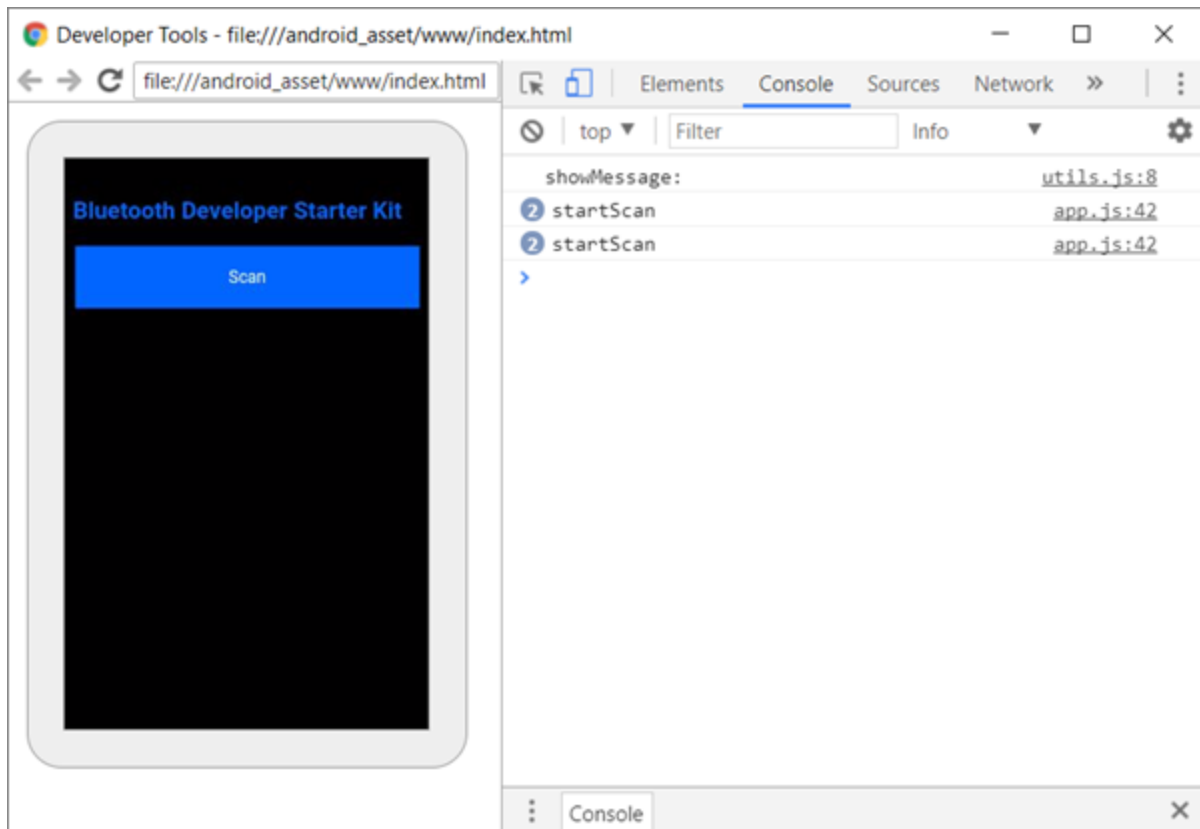


Figure 6 - The Google Chrome remote debugging screen

Click the Scan button a few times and you'll see the output from the *app.startScan()* function.

```
35  app.findDevices = function() {  
36      app.clearDeviceList();  
37      app.startScan();  
38  }  
39  
40  app.startScan = function()  
41  {  
42      console.log("startScan");  
43  }
```

Figure 7 - console.log statement in app.startScan ()

Remote Debugging with Safari

To perform remote debugging using Safari, follow these steps:

1. Plug your iOS device into your laptop over USB
2. On the iOS device, go into Settings / Safari / Advanced and switch Web Inspector on
3. Launch Safari on the Mac
4. In the Safari / Preferences menu, enabled "Show develop menu in menu bar"
5. Launch your mobile application
6. In Safari's Develop menu, your device should be listed as a menu with your running mobile application available as a menu item within it
7. Select the application menu item in Safari's Develop menu on your Mac.
8. Test your mobile application and watch console messages appear in Safari

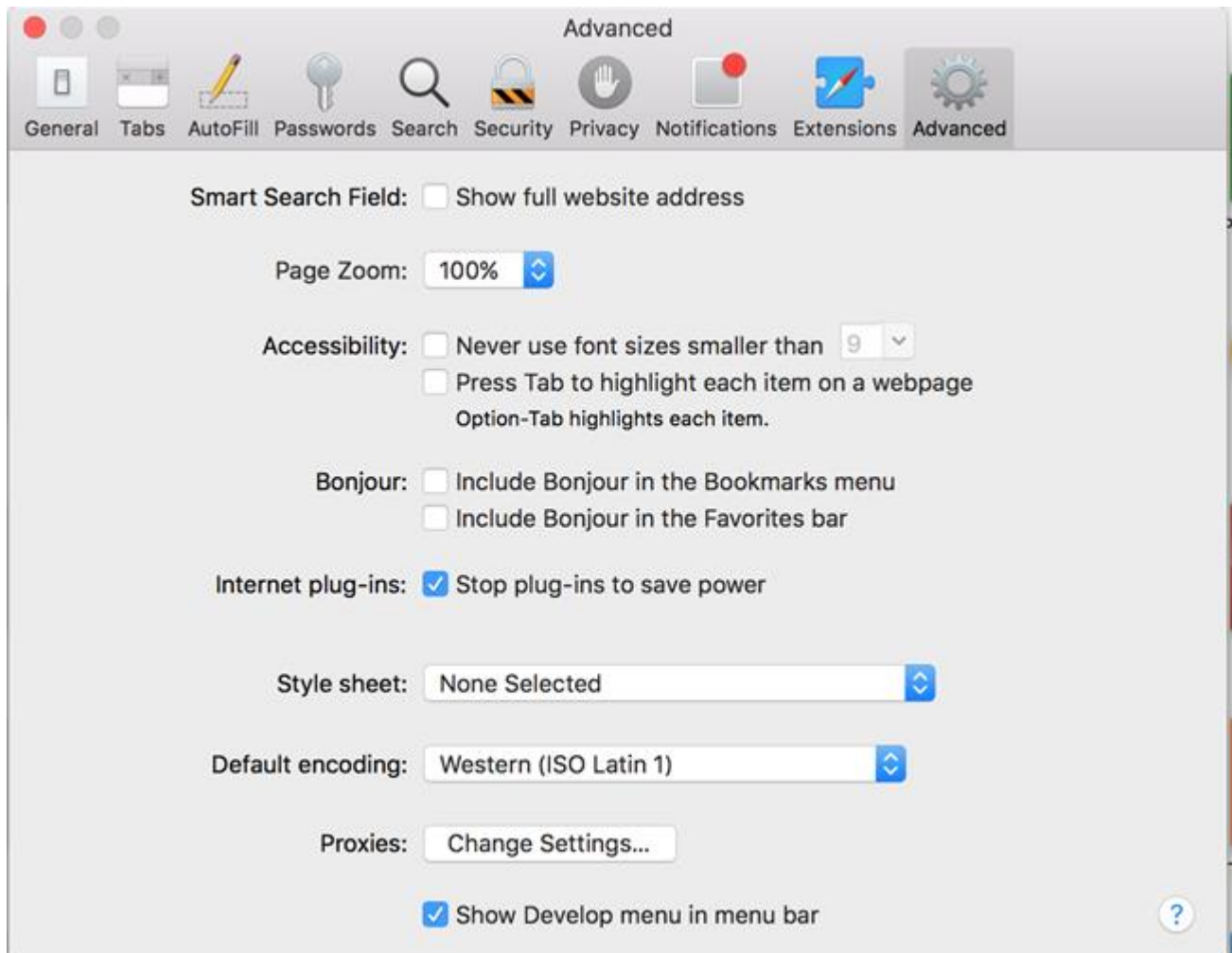


Figure 8 - Show Develop Menu in Safari preferences

The Bluetooth Low Energy API

We're using a 3rd party Cordova plugin to provide us with an API for Bluetooth Low Energy operations. You'll find both its code and API documentation here:

<https://github.com/don/cordova-plugin-ble-central>

Cordova Code Patterns

It's common for Cordova programming to involve making function calls where the function called requires two special, user-written functions amongst its arguments. The first function argument is to be called when execution of the main function is successful and the second, in the event that there is an error. So the general pattern looks something like this:

```
// define the function to be called if everything works as expected
function onEverythingOk() {
    alert("Success!");
}
// define the function to be called if an error occurs
function onError(err_code) {
    alert("Error! "+err_code);
}

api_object.doSomething(arg1, arg2, onEverythingOk, onError);
```

Figure 9 - Example of the closure pattern used with Cordova plugins

As you can see, API functions belong to JavaScript objects, so we prefix calls to particular functions with the name of that object. We then pass parameters, as indicated in the API documentation, and typically this will include references to two functions, as described.

The Bluetooth Low Energy API we are using in this lab is available via a JavaScript object with the name "ble". It has been automatically created and is available for use.

Exercise 1 - Scanning for peripheral devices

For this exercise you need to have your peripheral, running the code created in the server lab (or the complete solution code which we have included). Our goal in this exercise is to give our Cordova 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.

1. Scanning and Scan Results Screen

When the BDSK application first starts, an HTML div with id="device_list" is visible. This div element acts as the screen from which Bluetooth device scanning is performed. It has a button marked "Scan" at the top.

```
56 <!-- Scanning and Device List -->
57 <div id="device_list" class="screen">
58   <h2>Bluetooth LE Developer Study Guide</h2>
59   <table>
60     <tr>
61       <td><button id="find" class="wide_button" onclick="app.findDevices()">Scan</button></td>
62     </tr>
63   </table>
64
65   <div class="status" id="message"></div>
66   <table id="tbl_devices">
67     </table>
68 </div>
```

Figure 10 - HTML implementing the device scanning screen

As you can see from Figure 4, clicking the Scan button triggers a call to a JavaScript function called *app.findDevices()*. Figure 7, repeated here for convenience as Figure 9,

```
35 app.findDevices = function() {
36   app.clearDeviceList();
37   app.startScan();
38 }
39
40 app.startScan = function()
41 {
42   console.log("startScan");
```

Figure 11 - the findDevices() and startScan() functions

Exercise one consists of two stages. In the first stage, we'll make the Scan button discover any and all Bluetooth Low Energy devices which are in range and advertising and we'll list them all on the Scan

screen. In the second stage, we'll improve the user experience by filtering the list of discovered devices so that only the peripheral BDSK device is selected for inclusion in the device list within our UI.

2. Device Discovery

We have four tasks to complete, here.

- Define a success function
- Define an error function
- Initiate scanning for 5 seconds
- Check whether or not any devices were discovered during scanning

Find the *startScan()* function in *app.js* and add the following code under the first TODO comment.

```
//TODO create onDeviceFound function
function onDeviceFound(peripheral) {
    console.log("Found " + JSON.stringify(peripheral));
    if (app.isMyDevice(peripheral.name)) {
        found_my_device = true;
        app.showDiscoveredDevice(peripheral.id, peripheral.name);
    }
}
```

Figure 12 - scanning onDeviceFound function

This function will be called each time a new, advertising device is discovered. It receives a JSON object containing details of the discovered device, as a parameter. We log the peripheral object and examples look like this:

```
{"name":"Kontakt","id":"C5:EA:CF:9A:8F:73","advertising":{},"rssi":-78}
{"name":"BDSK","id":"98:4F:EE:0F:40:4F","advertising":{},"rssi":-41}
{"id":"E6:D4:CD:29:E9:58","advertising":{},"rssi":-69}
```

Note that not all peripheral objects have a name property. We'll need to accommodate this fact later on.

Our application lists any device of interest, as determined by the *isMyDevice* function, in a table in the *device_list* screen. Note currently, the *isMyDevice* function always returns true so that all devices that are discovered are listed. Take a look at that function to be sure you're clear on this point.

The *showDiscoveredDevice* function takes care of updating the UI with details of the discovered device. Take a look at it if you're interested in how this dynamic HTML update is achieved.

Now add the following code under the second TODO comment.

```
//TODO create scanFailure function
```

```
function scanFailure(reason) {  
    alert("Scan Failed: "+JSON.stringify(reason));  
}
```

This function will be called if an error occurs whilst scanning. A parameter containing information about the nature of the failure will be included in the call to this function. To keep things simple, all we'll do in this situation is display an error dialogue.

Now initiate scanning for 5 seconds, using the Cordova ble API. Provide our success and failure functions as arguments as shown below.

```
//TODO start scanning for devices for 5 seconds  
ble.scan([], 5, onDeviceFound, scanFailure);  
showMessage("Scanning...");
```

ble.scan is of course, one of the Cordova plugin's API functions. Check the documentation for the function to ensure you understand each of the 4 arguments. The only one which might not be immediately understandable is the first one, which we're not using.

showMessage is a function, provided in the starter code, which displays a message in the scanning screen UI.

The fourth and final step of this part of the exercise requires us to add some code which will check whether or not any devices were found and displayed in the device list. Add the following code to the *startScan()* function:

```
//TODO check outcome of scanning after 5 seconds have elapsed using a timer  
setTimeout(function() {  
    showMessage("");  
    if (!found_my_device) {  
        alert("Could not find your device");  
    }  
}, 5000);
```

To understand this code, requires us to look at how the *found_my_device* variable is used. The starter code initialises it to false when *startScan* is first entered. In the *onDeviceFound* function, we set it to true if it's selected by the *isMyDevice* function, which currently, all discovered devices are. *setTimeout* is a standard JavaScript function and it sets up a delayed call to a specified function. In our case, we call that function 5000ms later and as you can see, we check the *found_my_device* variable to see whether or not it has been set to true, indicating that at least one device was found.

3. 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. Plug in your peripheral, with the lab code installed on it. Build and run your application on a physical device now. If you're testing in an Android device, depending on the Android version, you may be prompted to grant location permissions when you click the Scan button, as shown in Figure 12. This is as expected. Android's permissions system categorises Bluetooth scanning as being location related. Click the Allow option if the permissions dialogue appears.

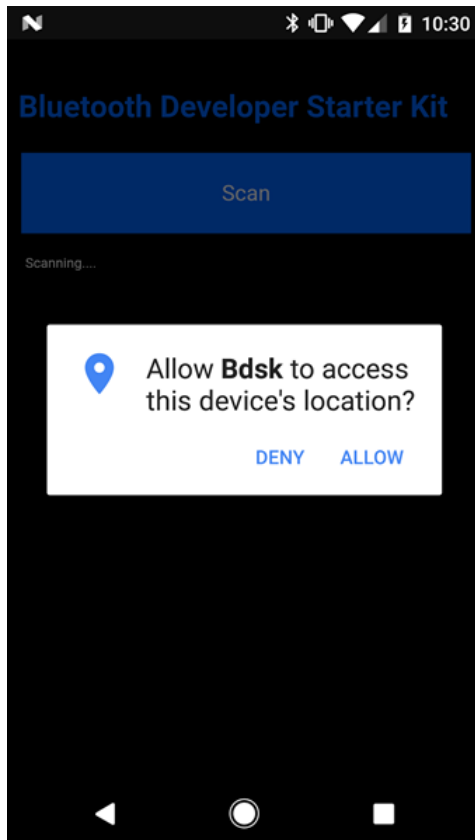


Figure 13 - Android may request Location Permissions

If the scanning process times out while you are considering granting permissions, you'll see an alert box with the message "Could not find your device". If this happens, just click Scan again.

When you've successfully initiated scanning, you should see a list of all of the advertising Bluetooth LE devices in range appear, including your peripheral BDSK device. It could look something like this:

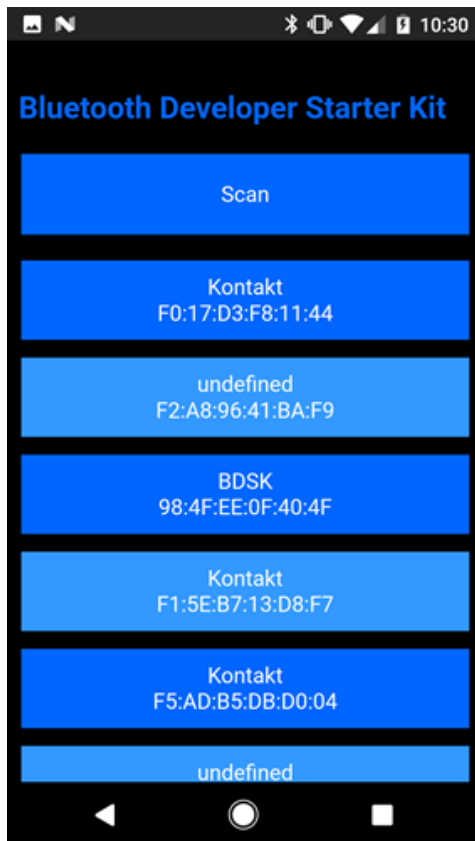


Figure 14 - List of all devices found during scanning

4. Device Filtering

At this stage, our application scans for and lists **all** Bluetooth peripherals it finds. It would be much better if it only displayed BDSK peripheral devices since these are the only devices our application will work with. The *isMyDevice* JavaScript function will take care of this with a simple modification.

Towards the top of *app.js* we have the following constant defined:

```
app.device.ADV_NAME = 'BDSK'
```

This is the value which the peripheral includes in its Bluetooth advertising packets and next we'll modify the *isMyDevice* function to check for it.

Update *isMyDevice* so that it looks like this:

```
app.isMyDevice = function(device_name)
{
    console.log("isMyDevice("+device_name+")");
    //TODO implement device name matching
    if (device_name == null || device_name == undefined) {
        return false;
    }
}
```

```
}  
    console.log('device name: ' + device_name);  
    return (device_name == app.device.ADV_NAME);  
};
```

Notice that we allow for the possibility that the device to be checked is not advertising a name. If, on the other hand, the advertised name is “BDSK” then we’ve found our peripheral device and we return true so that only this device is selected and displayed in the scan results list.

5. Checkpoint: Test Device Filtering

Build, install and test your application. After clicking the Scan button, you should only see the BDSK device listed. If you don’t check that your peripheral is plugged in and if necessary reset it to ensure it is advertising. Check your code matches the code in the provided solution.

6. Summary of Exercise 1

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

1. Implemented Bluetooth scanning and listed all advertising Bluetooth peripherals found.
2. Improved scanning by filtering out all devices except the peripheral running the BDSK code from the server lab of the Bluetooth LE Developer Study Guide.

Exercise 2 – Communicating with the peripheral

In the next part of this lab we will implement functions which allow the user to interact with the peripheral over Bluetooth in various ways, as described in the functional requirements table above. The user interface is already present in the starter project so that all we need to focus on is the Bluetooth-related functionality.

1. Selecting a device

The device(s) listed on the scan results screen are in fact HTML buttons which have been dynamically inserted into the UI. Each one has an onclick property and was created from a JavaScript string that looks like this:

```
var btn_connect = "<button class=\""+button_class+"\" onclick=\"app.showMain('"+address+"')\"<br>"+name+"<br>"+address+"</button>";
```

In all cases, selecting one of the listed devices triggers a call to the *showMain* function. The Bluetooth device address is passed to *showMain* as an argument. Review this code and trace the path that device addresses take, from having been discovered during scanning, to being passed through to *showMain* when selected by the user.

When *showMain* executes, we store the selected device's address in a variable:

```
selected_device_address = address;
```

We'll use this variable in many of our Bluetooth operations.

2. Checkpoint: Select the BDSK Device

On your test device, scan and this time select the BDSK device listed in the scan results. This selects the device for use in subsequent interactions, stores the Bluetooth address of the device and causes the UI to navigate to the second screen. Your UI should look something like this:

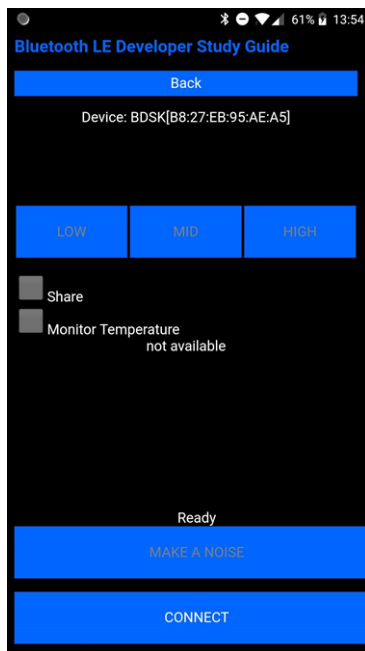


Figure 15 - The main screen

Note that the selected device address has been displayed near the top of the UI.

The HTML which defines this screen is shown in Figure 16.

```

70 <!-- main Page -->
71 <div id="main_page" class="full_screen" hidden="true">
72   <h2>Bluetooth LE Developer Study Guide</h2>
73   <button class="wide_button_slim" onclick="app.exitMain()">Back</button>
74   <div class="centred" id="device_details"></div>
75   <div class="rssi" id="rssi">&nbsp;</div>
76   <div id="proximity_rectangle" class="proximity_rectangle"></div>
77   <table>
78     <tr>
79       <td class="third"><button id="btn_low" class="wide_button" onclick="app.setAlertLevel(0)">LOW</button></td>
80       <td class="third"><button id="btn_mid" class="wide_button" onclick="app.setAlertLevel(1)">MID</button></td>
81       <td class="third"><button id="btn_high" class="wide_button" onclick="app.setAlertLevel(2)">HIGH</button></td>
82     </tr>
83   </table>
84   <div class="switch"><label class="switch_label"><input class="large_cb" type="checkbox" id="sharing" value="0"
85     onclick="app.setSharing()">Share</label></div>
86   <div class="switch"><label class="switch_label"><input class="large_cb" type="checkbox" id="temp_monitoring"
87     value="0" onclick="app.setTemperatureMonitoring()">Monitor Temperature</label></div>
88   <div class="full_centred" id="temperature">not available</div>
89   <div class="footer">
90     <div id="info" class="full_centred"></div>
91     <div class="full_centred"><button id="btn_noise" class="wide_button" onclick="app.makeNoise()">MAKE A NOISE</button></div>
92     <div class="full_centred"><button id="btn_connect" class="wide_button" onclick="app.toggleConnectionState()">CONNECT</button></div>
93   </div>
94 </div>

```

Figure 16 - The main screen HTML

Note the use of onclick to connect the UI to various JavaScript functions.

3. Connecting to the peripheral

Next, we'll add code to handle the CONNECT button being clicked. At the top of app.js you'll find a variable called *connected* which has an initial value of false. As you might expect, we use this variable to keep track of whether we're connected to the peripheral or not. Its value determines much regarding

the state of the main UI screen, including the label on the Connect button and what happens when the user clicks it.

As you can see from Figure 15, clicking the CONNECT button causes the *toggleConnectionState()* function to be called. Update *toggleConnectionState* so that it matches the following code:

```
app.toggleConnectionState = function() {
  console.log("toggleConnectionState("+selected_device_address+") : connected="+connected);
  if (!connected) {
    app.connectToDevice(selected_device_address);
  } else {
    // We'll add this code later!
  }
};
```

To start with, we'll focus on the act of connecting to the peripheral. We'll deal with disconnecting (when the *connected* variable indicates we're already connected), later.

Find the *connectToDevice* function in *app.js*. As the name suggests, we'll implement code which connects the smartphone to the peripheral here. But due to the nature of the API we're using, we also need to handle failed attempts to connect and both unexpected and explicitly requested disconnects.

The API function which initiates connecting is the *ble.connect* function. It uses the familiar code pattern, requiring arguments that include a function which handles the success case and one which handles failures. The failure function is called if the attempt to connect to the BDSK device fails or if the connection is dropped later on, either unexpectedly or deliberately.

4. connectToDevice skeleton code

Let's start with some skeleton code. Update *connectToDevice* so it looks like this:

```
app.connectToDevice = function(device_address)
{
  console.log('connectToDevice: Connecting to '+device_address);
  //TODO connect to the BDSK device and handle failures to connect and unexpected
  disconnections

  function onConnected(peripheral)
  {
  }

  function onDisconnected(peripheral)
  {
  }

  ble.connect(device_address, onConnected, onDisconnected);
};
```

That's the basic, skeleton code in place. The *peripheral* parameter passed to the two functions is a JSON object which contains information about the device we attempted to connect to.

5. onConnected

Now let's complete the *onConnected* function, which is the function which gets called when we successfully connect. Here's what we need to do:

- set the connection state tracker variable *connected* to true
- check whether or not the device we've connected to has the required Bluetooth GATT services
 - If it does, we'll modify the UI so that buttons which are only valid when in a connected state are enabled
 - If it does not, we'll tell the user and deliberately disconnect from this device

Update *connectToDevice* with the changes shown here:

```
app.connectToDevice = function(device_address)
{
    console.log('connectToDevice: Connecting to '+device_address);
    //TODO connect to the BDSK device and handle failures to connect and unexpected
    disconnections
    function onConnected(peripheral)
    {
        console.log('connectToDevice: onConnected');
        connected = true;
        // check we have the required services
        if (app.hasService(peripheral,app.device.IMMEDIATE_ALERT_SERVICE)
            && app.hasService(peripheral,app.device.LINK_LOSS_SERVICE)
            && app.hasService(peripheral,app.device.PROXIMITY_MONITORING_SERVICE)
            && app.hasService(peripheral,app.device.HEALTH_THERMOMETER_SERVICE)
        ) {
            app.setControlsConnectedState();
            showInfo("connected",0);
        } else {
            showInfo("ERROR: missing required GATT services",2);
            ble.disconnect(
                selected_device_address,
                function() {
                    console.log("connectToDevice: disconnected OK");
                    showInfo("Disconnected");
                },
                function(error) {
                    console.log("connectToDevice: disconnect failed: "+error);
                    alert("Error: Failed to disconnect");
                }
            );
        }
    }
}
```

```

        showInfo("Error: Failed to disconnect",2);
    });
}

function onDisconnected(peripheral)
{
}

ble.connect(device_address, onConnected, onDisconnected);

};

```

Review the code we just added and make sure you understand it. The *hasService* function just searches through an array of service UUIDs which is contained within the JSON peripheral object provided by the ble API and returns true if a match is found. Our application requires the connected device to host all three of the Link Loss, Immediate Alert and Proximity Monitoring services. Their UUIDs are defined towards the top of the app.js file. The *setControlsConnectedState* sets UI controls like buttons into the appropriate state now that we have a connection.

If the device does not have the required services, as you can see, we use the API *ble.disconnect* to deliberately disconnect from it after displaying an error message.

6. onDisconnected

Now add the following code to the *onDisconnected* function.

```

app.connectToDevice = function(device_address)
{
    console.log('connectToDevice: Connecting to '+device_address);
    //TODO connect to the BDSK device and handle failures to connect and unexpected
    disconnections
    function onConnected(peripheral)
    {
        console.log('connectToDevice: onConnected');
        connected = true;
        // check we have the required services
        if (app.hasService(peripheral,app.device.IMMEDIATE_ALERT_SERVICE)
            && app.hasService(peripheral,app.device.LINK_LOSS_SERVICE)
            && app.hasService(peripheral,app.device.PROXIMITY_MONITORING_SERVICE)
            && app.hasService(peripheral,app.device.HEALTH_THERMOMETER_SERVICE)
        ) {
            app.setControlsConnectedState();
            showInfo("connected",0);
        } else {
            showInfo("ERROR: missing required GATT services",2);
        }
    }
}

```

```

        ble.disconnect(
            selected_device_address,
            function() {
                console.log("connectToDevice: disconnected OK");
                showInfo("Disconnected");
            },
            function(error) {
                console.log("connectToDevice: disconnect failed: "+error);
                alert("Error: Failed to disconnect");
                showInfo("Error: Failed to disconnect",2);
            });
    }
}

function onDisconnected(peripheral)
{
    console.log('connectToDevice: onDisconnected');
    // called if ble.connect fails OR if an established connection drops
    if (!connected) {
        // we tried to connect and failed
        console.log('connectToDevice: Error: Connection failed');
        console.log(JSON.stringify(peripheral));
        showInfo("Error: not connected",2);
        alert("Error: could not connect to selected device");
    } else {
        // we were already connected and disconnection was unexpected
        showInfo("Error: unexpectedly disconnected",2);
    }
    connected = false;
    app.setControlsDisconnectedState();
}

ble.connect(device_address, onConnected, onDisconnected);
};

```

The *onDisconnected* function merely determines whether the event was expected or unexpected and informs the user accordingly. It also calls a canned function to set the UI into an appropriate state for when we're not connected to the BDSK device.

We also need to update the *toggleConnectionState* code so that if we're currently connected, executing this function will cause the smartphone to disconnect from the peripheral. Update the function as shown here:

```
app.toggleConnectionState = function() {
```

```
console.log("toggleConnectionState("+selected_device_address+") : connected="+connected);
if (!connected) {
    app.connectToDevice(selected_device_address);
} else {
    console.log("disconnecting");
    ble.disconnect(
        selected_device_address,
        function() {
            console.log("toggleConnectionState: disconnected OK");
            showInfo("Disconnected");
            connected = false;
            app.setControlsDisconnectedState();
        },
        function(error) {
            console.log("toggleConnectionState: disconnect failed: "+error);
            alert("Failed to disconnect");
            showInfo("Error: Failed to disconnect",2);
        });
    }
};
```

7. Disconnect when back button pressed

When the user exits the main screen, either using their smartphone's navigation controls or by selecting the Back button on our UI, we should disconnect from the peripheral, assuming we're currently connected.

Find the *exitMain()* function, which currently looks like this:

```
app.exitMain = function() {  
    app.showDeviceList();  
};
```

Replace the code in the *exitMain* function so that it now looks like this:

```
app.exitMain = function() {  
    if (connected) {  
        showInfo("Disconnecting");  
        ble.disconnect(  
            selected_device_address,  
            function() {  
                console.log("disconnected OK");  
                app.showDeviceList();  
            },  
            function(error) {  
                console.log("disconnect failed: "+error);  
                alert("Failed to disconnect");  
                app.showDeviceList();  
            }  
        ));  
    } else {  
        app.showDeviceList();  
    }  
}
```

Review the new *exitMain* code. In all cases, we leave the main screen and show the device discovery screen. If we're currently connected, we disconnect.

8. Checkpoint: Connecting and Disconnecting

Build, install and test your application:

1. Scan, select the BDSK device and then press the CONNECT button to connect to it.

- The peripheral LCD display or peripheral console (depending on the peripheral you're using) should indicate a connection has been established and the application UI should enable various buttons.
- 2. Press DISCONNECT.
 - The peripheral LCD / console should say "Disconnected" and most buttons on the application UI should be disabled.
- 3. Press CONNECT again
 - Per test result (1) above
- 4. Unplug your peripheral and wait approximately 20-30 seconds.
 - The smartphone application should notice that the connection has dropped unexpectedly and inform the user.
- 5. Plug the peripheral back in and press the Connect button on the UI
 - Per test result (1) above

Note: if you have an LCD display connected to your peripheral or you're using your device's console for output (see the server lab documentation), you'll see it change in response to the peripheral being connected to or disconnected from.

Note: if you're wondering why it takes 20-30 seconds for your application to notice that the peripheral has disconnected, this is due to a Bluetooth timeout parameter called the Supervisor Timeout which is associated with the connection and determines how long the devices should wait to hear from the other one at the Bluetooth link layer, before concluding the connection has dropped. Your smartphone is setting the supervisor timeout when it connects to the peripheral and the APIs available to smartphone developers do not allow them to modify the value used.

9. Acquiring the current Alert Level

Soon we'll make it possible to set the Link Loss alert level to a value of 0, 1 or 2 using the three LOW, MID, and HIGH buttons. We also want the UI to reflect the currently selected alert level though, so after connecting to the peripheral, we want to read the Link Loss service's Alert Level characteristic and use the value obtained to highlight the current selection by changing the colour of the label on one of the three, related buttons.

Find the *establishCurrentAlertLevel()* function in the app.js file and update it so that it looks like this:

```
app.establishCurrentAlertLevel = function() {
    console.log("establishCurrentAlertLevel");
    ble.read(selected_device_address, app.device.LINK_LOSS_SERVICE,
app.device.ALERT_LEVEL_CHARACTERISTIC,
        function(data) {
            console.log("read current link loss alert level OK");
            var alert_level_data = new Uint8Array(data);
            if (alert_level_data.length > 0) {
                console.log("alert_level="+alert_level_data[0]);
            }
        }
    );
}
```

```

        alert_level = alert_level_data[0];
        app.setAlertLevelSelected();
    }
},
function(err) {
    console.log("ERROR: reading current alert level: "+err);
});
}

```

The *ble.read* function is part of the Bluetooth LE API we're using and it initiates the reading of a specified characteristic. The arguments it requires are the address of the device we're communicating with, the UUID of the GATT service that contains the characteristic we want to read, the UUID of the characteristic, a function which receives the read data if the operation succeeds, and as usual a function to call in the event of an error.

function(data) is the success function and *data* contains the value of the characteristic which was obtained. We treat it as a byte array and take the first byte only, converting it into a standard JavaScript variable. We store the value for future use and update the UI by calling a pre-written function called *setAlertLevelSelected* which was provided in the starter project. This causes the LOW, MID or HIGH button text to change colour to red, depending on whether the characteristic value was 0, 1 or 2 respectively.

We need to call *establishCurrentAlertLevel* after connecting and confirming that the remote device contains the right GATT services. Find the *onConnected* function inside the *connectToDevice* function and add the call to the if block which checks the GATT services which have been discovered:

```

// check we have the required services
if (app.hasService(peripheral,app.device.IMMEDIATE_ALERT_SERVICE)
    && app.hasService(peripheral,app.device.LINK_LOSS_SERVICE)
    && app.hasService(peripheral,app.device.PROXIMITY_MONITORING_SERVICE)
    && app.hasService(peripheral,app.device.HEALTH_THERMOMETER_SERVICE)
) {
    app.setControlsConnectedState();
    app.establishCurrentAlertLevel();
    showInfo("connected",0);
} else {

```

10. Set the Alert Level

Next, we'll add code which responds to the LOW, MID and HIGH buttons being pressed by writing a corresponding value of 0, 1 or 2 to the Link Loss service's Alert Level characteristic. Find the *setAlertLevel* function in *app.js* and update it so that it looks like this:

```
app.setAlertLevel = function(level) {
    console.log("setAlertLevel("+level+")");
    var alert_level_bytes = [0x00];
    alert_level_bytes[0] = level;
    var alert_level_data = new Uint8Array(alert_level_bytes)
    ble.write(
        selected_device_address,
        app.device.LINK_LOSS_SERVICE,
        app.device.ALERT_LEVEL_CHARACTERISTIC,
        alert_level_data.buffer,
        function() {
            console.log("written LL alert level OK");
            alert_level = level;
            app.setAlertLevelSelected();
        },
        function(e) {
            console.log("ERROR: writing LL alert level: "+e);
        });
};
```

The *ble.write* API is similar in form to the others we've seen so far. The device is identified in the first argument and then a service UUID and characteristic UUID identify the characteristic to write to. The fourth argument contains the data value to write, in the form of a byte array and the final two arguments are a function for the positive case and a function for handling errors. Code prior to calling *ble.write* is simply preparing the specified level value in the required format. Note that in the success case, we call *setAlertLevelSelected* to update the LOW, MID or HIGH button so that the user can see that it has been selected.

11. Checkpoint: Test getting and setting the Alert Level

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

1. Scan, select and connect.
 - One of the three LOW|MID|HIGH buttons will have a red label indicating that the peripheral was currently set to the corresponding alert level. Which button changes to red depends on the state your peripheral was in when you connected to it.
2. Select LOW

- 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 you should see the green LED flash by way of acknowledgement.
- 3. Go back and disconnect. Select and connect again.
 - The LOW button should be highlighted red to indicate the current alert level read back from the peripheral is “LOW” (0).
- 4. Select MID.
 - 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 you should see the yellow LED flash by way of acknowledgement.
- 5. Go back and disconnect. Select and connect again.
 - The peripheral LEDs should all flash when you disconnect. This is the peripheral’s responding to the link having been lost.
 - The MID button should be highlighted red to indicate the current alert level read back from the peripheral is “MID” (1).
- 6. Select HIGH.
 - The HIGH button should be highlighted in red to indicate the current alert level read back from the peripheral is “HIGH” (2). If you have the appropriate circuit connected to your peripheral you should see the red LED flash by way of acknowledgement.
- 7. Go back and disconnect. Select and connect again.
 - The peripheral LEDs should all flash and the buzzer should sound at the same time when you disconnect. This is the peripheral’s responding to the link having been lost.
 - The HIGH button should be highlighted red to indicate the current alert level read back from the peripheral is “HIGH” (2).

12. 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.

To implement RSSI polling, we need to start a JavaScript timer which will call an API function *ble.readRSSI* every second. We create the timer when a connection has been established and cancel the timer task when the connection is dropped. On reading an RSSI value, we’ll set the colour of the UI “proximity_rectangle” div in the index.html and display the value.

Find the *startRssiPolling* function and update it so that it looks like this:

```
app.startRssiPolling = function() {
  console.log("startRssiPolling");
  if (connected) {
    rssi_timer = setInterval(
      function() {
        if (connected) {
```

```

        ble.readRSSI(selected_device_address,
            function(rssi) {
                document.getElementById("rssi").innerHTML = "RSSI = "+rssi;
                var rssi_value = parseInt(rssi);
                if (rssi_value < -80) {
                    rectangle.style.backgroundColor = "#FF0000";
                } else if (rssi_value < -50) {
                    rectangle.style.backgroundColor = "#FF8A01";
                } else {
                    rectangle.style.backgroundColor = "#00FF00";
                }
            },
            function(err) {
                console.log('Error: unable to read RSSI',err);
                showInfo("Error: unable to read RSSI",2);
                clearInterval(rssi_timer);
            })
        )
    },
    1000);
}
};

```

As described, if we're in a connection with the peripheral, this code establishes a timer which calls *ble.readRSSI* every second. This is another of our Bluetooth API functions. Its three arguments are the address of the selected device, a success function which gets passed the RSSI value and an error function which gets called if something goes wrong.

In the success value, we display the measured RSSI value and then we simply check it against three thresholds (which were established through experimentation) and use this to designate the peripheral as being nearby, at middle distance from the smartphone or at a long distance from it (but still in range). We then set the UI proximity indicator rectangle to green, yellow or red accordingly.

Add a call to *startRssiPolling* in the *onConnected* function within the *connectToDevice* function.

```

function onConnected(peripheral)
{
    console.log('connectToDevice: onConnected');
    connected = true;
    app.startRssiPolling();
    // check we have the required services
    if (app.hasService(peripheral,app.device.IMMEDIATE_ALERT_SERVICE)
        && app.hasService(peripheral,app.device.LINK_LOSS_SERVICE)
    )
    {
        // ...
    }
}

```

```

    && app.hasService(peripheral,app.device.PROXIMITY_MONITORING_SERVICE)
    && app.hasService(peripheral,app.device.HEALTH_THERMOMETER_SERVICE)
  ) {

```

When we disconnect, we need to stop the timer which polls the RSSI value. Update the *toggleConnectionState* function as shown below:

```

app.toggleConnectionState = function() {
  console.log("toggleConnectionState("+selected_device_address+") : connected="+connected);
  if (!connected) {
    app.connectToDevice(selected_device_address);
  } else {
    console.log("disconnecting");
    clearInterval(rssi_timer);
    document.getElementById("rssi").innerHTML = "&nbsp;";
    ble.disconnect(
      selected_device_address,
      function() {
        console.log("toggleConnectionState: disconnected OK");
        showInfo("Disconnected");
        connected = false;
        app.setControlsDisconnectedState();
      },
      function(error) {
        console.log("toggleConnectionState: disconnect failed: "+error);
        alert("Failed to disconnect");
        showInfo("Error: Failed to disconnect",2);
      });
  }
};

```

13. Checkpoint: Test RSSI monitoring

Build and test your application. After connecting to the peripheral 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 and watch the RSSI value change and occasionally, the rectangle change colour. The UI should look something like Figure 16.

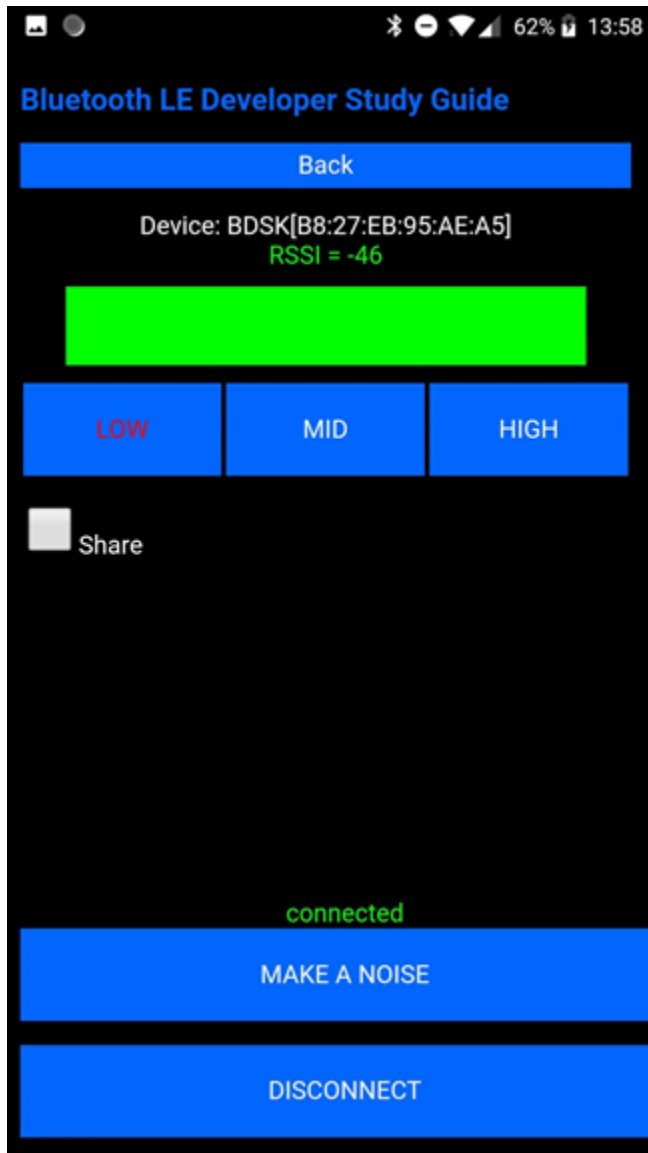


Figure 17 - UI showing RSSI monitoring

14. 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.

Find the *makeNoise* function in *app.js* and update it so that it looks like this:

```
app.makeNoise = function() {
  console.log("makeNoise");
  var alert_level_bytes = [0x00];
  alert_level_bytes[0] = alert_level;
  var alert_level_data = new Uint8Array(alert_level_bytes)
  ble.writeWithoutResponse(
    selected_device_address,
    app.device.IMMEDIATE_ALERT_SERVICE,
    app.device.ALERT_LEVEL_CHARACTERISTIC,
    alert_level_data.buffer,
    function() {
      console.log("written IA alert level OK");
    },
    function(e) {
      console.log("ERROR: writing IA alert level: "+e);
    }
  );
};
```

Note that here we used the *ble.writeWithoutResponse* API function rather than the *ble.write* function. This is because the alert level characteristic when used within the Immediate Alert service supports the GATT write without response procedure but not the write (with response) procedure.

15. Checkpoint: Test Immediate Alert

Build, install and run your application. After connecting to the peripheral, 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-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.

16. 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 (1=near, 2=middle distance, 3=far away) to the peripheral by writing to the Client Proximity characteristic of the Proximity Monitoring service. Let's implement that now.

Find the *shareProximityData* function in *app.js*. Replace it with the following code:

```
app.shareProximityData = function(proximity_band, rssi) {
  console.log("shareProximityData("+proximity_band+", "+rssi+"");
```



```

var proximity_bytes = [0x00, 0x00];
proximity_bytes[0] = proximity_band;
proximity_bytes[1] = rssi;
var proximity_data = new Uint8Array(proximity_bytes)
ble.writeWithoutResponse(
    selected_device_address,
    app.device.PROXIMITY_MONITORING_SERVICE,
    app.device.CLIENT_PROXIMITY_CHARACTERISTIC,
    proximity_data.buffer,
    function() {
        console.log("written client proximity OK");
    },
    function(e) {
        console.log("ERROR: writing client proximity: "+e);
    });
};

```

We need to call this function every time we have a new RSSI reading, but only if the Share checkbox in the UI has been selected.

Find the *startRssiPolling* function in app.js and make the changes highlighted below:

```

app.startRssiPolling = function() {
    console.log("startRssiPolling");
    if (connected) {
        rssi_timer = setInterval(
            function() {
                if (connected) {
                    ble.readRSSI(selected_device_address,
                        function(rssi) {
                            document.getElementById("rssi").innerHTML = "RSSI = "+rssi;
                            proximity_band = 3;
                            var rssi_value = parseInt(rssi);
                            if (rssi_value < -80) {
                                rectangle.style.backgroundColor = "#FF0000";
                            } else if (rssi_value < -50) {
                                rectangle.style.backgroundColor = "#FF8A01";
                                proximity_band = 2;
                            } else {
                                rectangle.style.backgroundColor = "#00FF00";
                                proximity_band = 1;
                            }
                        }
                    )
                }
            },
            1000
        );
    }
}

```

```

        if (sharing) {
            app.shareProximityData(proximity_band, rssi_value);
        }
    },
    function(err) {
        console.log('Error: unable to read RSSI',err);
        showInfo("Error: unable to read RSSI",2);
        clearInterval(rssi_timer);
    })
}
},
1000);
}
};

```

All we do here is set a variable called *proximity_band* to value of 1, 2 or 3, representing near, middle distance or far away and then if the Share switch has been selected, we call the *shareProximityData* function which will write the RSSI value and the *proximity_band* variable to the peripheral's Client Proximity characteristic.

One last thing we need to do is tell the peripheral to clear any proximity sharing information on its LCD / console display and switch off LEDs when we first connect to it so that we can start again.

Update *onConnected* inside *connectToDevice* as shown:

```

app.connectToDevice = function(device_address)
{
    console.log('connectToDevice: Connecting to '+device_address);
    //TODO connect to the BDSK device and handle failures to connect and unexpected
    disconnections
    function onConnected(peripheral)
    {
        console.log('connectToDevice: onConnected');
        connected = true;
        // this will clear the peripheral LCD / console of proximity info and switch off all LEDs
        app.shareProximityData(0,0);
        app.startRssiPolling();
    }
}

```

The peripheral has been programmed to reset the LCD / console and LED lights in response to the Client Proximity characteristic having a value of 0x0000 written to it.

17. Checkpoint: Test proximity sharing

Test this feature by toggling the Share switch after connecting to the peripheral. With sharing enabled you should see RSSI values displayed on the LCD / console connected to the peripheral. The LED of the same colour as our UI's RSSI rectangle should also be lit.



18. 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.

The *audio.js* file contains functions for controlling the playback of a wave file which contains a recording of an alarm sound. Our next task is to update the *onDisconnected* function inside *connectToDevice* so that if the Bluetooth connection was unexpectedly lost, we play the alarm sound for 10 seconds.

Update your code as shown below:

```
function onDisconnected(peripheral)
{
    console.log('connectToDevice: onDisconnected');
    // called if ble.connect fails OR if an established connection drops
    if (!connected) {
        // we tried to connect and failed
        console.log('connectToDevice: Error: Connection failed');
        console.log(JSON.stringify(peripheral));
        showInfo("Error: not connected",2);
        alert("Error: could not connect to selected device");
    } else {
        // we were already connected and disconnection was unexpected
        showInfo("Error: unexpectedly disconnected",2);
        audio.play(alarm_sound_path);
        setTimeout(function() {
            audio.stop();
        }, 10000);
    }
    connected = false;
    app.setControlsDisconnectedState();
}
```

In the event that we experience an unexpected disconnection, we call *audio.play* to start the alarm sound and set up a JavaScript timer to call *audio.stop* 10 seconds later to stop it.

We also need to stop the alarm if it is sounding when we (re)connect to the peripheral. Update the *onConnected* function as shown:

```
function onConnected(peripheral)
{
    console.log('connectToDevice: onConnected');
    connected = true;
```

```
audio.stop();

// this will clear the peripheral LCD / console of proximity info and switch off all LEDs
app.shareProximityData(0,0);
app.startRssiPolling();
```

19. Checkpoint: Test the Mobile Application Response to Link Loss

Test this feature by connecting to the peripheral and then unplugging it. Wait approximately 20-30 seconds. The application should detect that the link has dropped and play the alarm sound for 10 seconds.

20. 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.

Find and update the `setTemperatureMonitoring` function so that it looks like this:

```
app.setTemperatureMonitoring = function() {
    console.log("setTemperatureMonitoring");
    monitoring_temperature = cb_temperature.checked;
    console.log("monitoring_temperature="+monitoring_temperature);
    if (monitoring_temperature) {
        app.startIndications();
    } else {
        app.stopIndications();
    }
};
```

As you can see, we're calling one of two functions to either start or stop temperature measurement indications, according to the state of the UI switch. Let's add those functions now:

```
app.stopIndications = function()
{
    console.log("stopping indications");
    ble.stopNotification(
        selected_device_address,
        app.device.HEALTH_THERMOMETER_SERVICE,
        app.device.TEMPERATURE_MEASUREMENT_CHARACTERISTIC);
    document.getElementById('temperature').innerHTML = 'not available';
};

app.startIndications = function () {
```

```

console.log('Starting indications');
ble.startNotification(selected_device_address,
    app.device.HEALTH_THERMOMETER_SERVICE,
    app.device.TEMPERATURE_MEASUREMENT_CHARACTERISTIC,
    function (buffer) {
        monitoring_temperature = 1;
        var data = new Uint8Array(buffer);
        console.log("data="+data.toString());
        var signed_data = new Int8Array(buffer);
        var units = "C";
        if (data[0] == 1) {
            units = "F";
        }

        // 4 bytes required for Int32 conversion. Also, make big endian while we're at it.
        var mantissa_bytes = new Uint8Array([0x00,0x00,0x00,0x00]);
        mantissa_bytes[1] = data[3];
        mantissa_bytes[2] = data[2];
        mantissa_bytes[3] = data[1];
        // propagate sign bit of most significant byte of the 3 mantissa bytes
        if (mantissa_bytes[1] & 0x80 == 0x80) {
            // sign bit is set so....
            mantissa_bytes[0] = 0xff
        }

        // now convert to a standard signed int
        var mantissa = 0;
        for (var i = 0; i < 4; ++i) {
            mantissa += mantissa_bytes[i];
            if (i < 3) {
                mantissa = mantissa << 8;
            }
        }
        console.log("mantissa=" + mantissa);
        var exponent = signed_data[4];
        console.log("exponent=" + exponent);
        var temperature = (mantissa * Math.pow(10, exponent) * 10);
        temperature = Math.round(temperature) / 10;
        document.getElementById('temperature').innerHTML = temperature + units;
    },
    function (e) {
        console.log("Error handling indication:" + e);
    }
);

```

```
        monitoring_temperature = 0;
    });
};
```

The API we're using makes no apparent distinction between indications (which require a response from the client) and notifications (which do not), so don't be confused by the fact that the API calls are to functions **startNotifications** and **stopNotifications**.

When starting indications, we supply the selected device's address and the service and characteristic UUIDs for the characteristic whose values we want transmitted as indications. We also supply two functions, one for the success case and one for the error case. The success function is called every time we receive an indication from the peripheral device and we receive the value as an `ArrayBuffer`. Using JavaScripts typed array functions, we convert the `ArrayBuffer` into both a signed octet version and an unsigned version. This is because we're receiving 5 octets in total, all of which are unsigned except for one and this is just an easy way to ensure we have access to values with or without their sign, as we require.

```
var data = new Uint8Array(buffer);
var signed_data = new Int8Array(buffer);
```

The first octet tells us whether the data is in degrees Celcius (0) or Fahrenheit (1). The following 3 octets comprise the mantissa in a floating point number and the final octet is the exponent. Using this data we calculate a temperature in the appropriate units and insert it into our main web page for display.

Note that we also maintain a flag called `monitoring_temperature`. A value of 1 means that indications are currently enabled. We'll use that to unsubscribe from them when the user decides to go back to the device discovery screen or disconnect from the peripheral.

Update the `exitMain` function as shown:

```
app.exitMain = function() {
    if (connected) {
        if (monitoring_temperature == 1) {
            app.stopIndications();
        }
        showInfo("Disconnecting");
        ble.disconnect(
            selected_device_address,
            function() {
                console.log("disconnected OK");
                app.showDeviceList();
            },
            function(error) {
```

```
        console.log("disconnect failed: "+error);
        alert("Failed to disconnect");
        app.showDeviceList();
    });
} else {
    app.showDeviceList();
}
}
```

And finally, update toggleConnectionState so that if the user opts to disconnect, we first unsubscribe from temperature indications if they're active.

```
app.toggleConnectionState = function () {

    console.log("toggleConnectionState(" + selected_device_address + ") : connected=" + connected);
    if (!connected) {
        app.connectToDevice(selected_device_address);
    } else {
        console.log("disconnecting");
        clearInterval(rssi_timer);
        document.getElementById("rssi").innerHTML = "&nbsp;";
        if (monitoring_temperature == 1) {
            app.stopIndications();
        }
    }
}
```

Test your Application

You're now ready to take your cross platform, hybrid mobile 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 peripheral BDSK device which should have the name "BDSK".
Connect	connects to the peripheral using Bluetooth peripheral
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. 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, 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	When enabled, the client should receive regular temperature measurement indications from the peripheral and display them. When unsubscribed, indications should be switched off and the temperature field in the UI read "not available".

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, monitors the RSSI value and can enable, disable and process temperature measurement indications. What's more, you can build a version which runs on Android and a version which runs on iOS all from the same source code!