Developer Study Guide: An introduction to Bluetooth Low Energy Development

Creating a Bluetooth Low Energy application for iOS using Swift

Version: 5.0.3

Last updated: 17th December 2018

Contents

Revision History

Version	Date	Author	Changes
1.0	1st 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	7th 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	3rd October	Martin Woolley, Bluetooth SIG	Android code retired to the legacy folder and designated 'Android 4' as it uses the Android 4.4 APIs which have been deprecated. Android lab modified to be based on Android Studio instead of Android Studio Code substantially refactored. Android lab and solution now uses the Android 5+ APIs and is compatible with Android 6+ with respect to the new permissions model. Key Android Bluetooth classes are now introduced and explained. New much more alarm-like alarm sound used.
3.2.0	16th December 2016	Martin Woolley, Bluetooth SIG	Old iOS Objective-C based resources retired and replaced with new lab and solution written in Swift.

4.0.0	7th July 2017	Martin Woolley, Bluetooth SIG	Added a new lab, with associated solution source code, which explains how to use the Apache Cordova SDK to create a cross platform mobile application which uses Bluetooth Low Energy. iOS lab text updated to improve description of recommended test steps in the second Checkpoint in 16. Link Loss Alert Level .
5.0.0	20th December 2017	Martin Woolley, Bluetooth SIG	Added new use case involving temperature measurements and Bluetooth Indications.
5.0.2	15th June 2018	Martin Woolley, Bluetooth SIG	Removed use of Bluetooth Developer Studio
5.0.3	17th December 2018	Martin Woolley, Bluetooth SIG	Changed name to "Developer Study Guide: An introduction to Bluetooth Low Energy Development"

Introduction

Overview

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

In this lab we will not cover any basics of Swift programming or how to set up an iOS development environment or use Xcode, 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 iOS applications using Swift and which use Bluetooth Low Energy.

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

- Xcode with support for Swift 3

Equipment Requirements

- USB cable
- iPhone or iPad. Check Apple developer web site for OS version requirements with respect to code built with Xcode using Swift 3.2.
- 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 may 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 are left to you, since you are expected to have at least a little experience of Swift and iOS programming. Other types of issue, such as calling a function 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 Swift and Xcode 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. We've also included a short list of some of the issues we bumped into whilst developing this lab and how we resolved them, just in case you have the same 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.

Functional Requirements

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

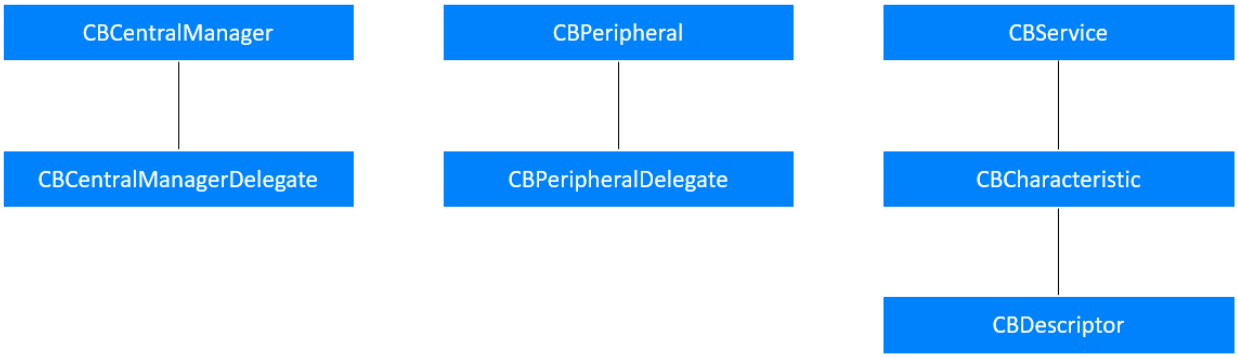
	Feature	Expected Behaviour
1	Be able to scan for and discover devices running and advertising the custom profile we designed and implemented in the server labs of this study guide.	Clicking a button will initiate Bluetooth scanning. Filtering will ensure that only devices which are advertising with a 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.

Swift Bluetooth Classes

We'll be using classes from the iOS / Swift CoreBluetooth framework. You should use the Swift API documentation to familiarise yourself with classes we'll be using, as you encounter them in this lab. See https://developer.apple.com/reference/corebluetooth for full details.

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



The delegate design pattern is very commonly used in iOS programming and the CoreBluetooth framework is no exception. Classes often require delegate counterparts associated with them and in the case of CoreBluetooth, those delegate objects are typically used to deliver the results of Bluetooth operations when they have completed.

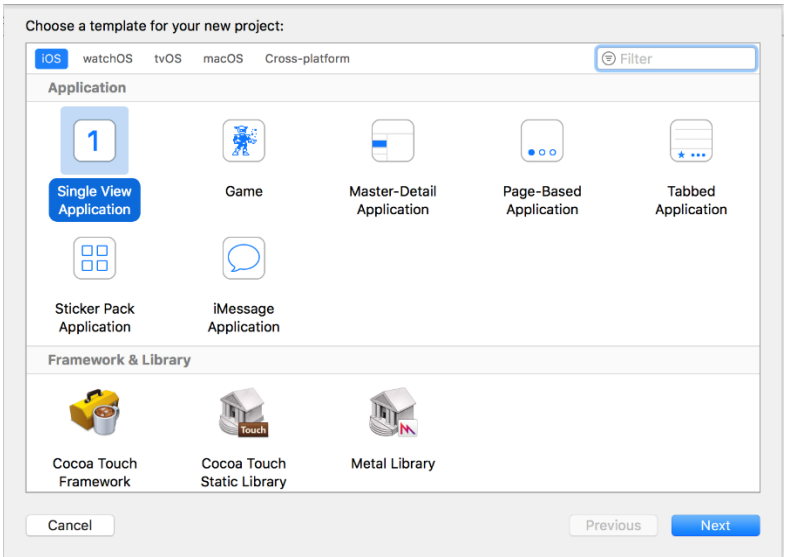
CBCentralManager provides APIs for initiating the usual GAP Central procedures such as scanning and, therefore device discovery. CBCentralManagerDelegate is its delegate.

CBPeripheral represents a GAP Peripheral device, perhaps discovered by CBCentralManager. It's to a CBPeripheral that a connection will be established. CPPeripheralDelegate is its delegate.

The usual GATT types (see the LE Basic Theory document if you're not familiar with the term GATT) are represented by CBService, CBCharacteristic and CBDescriptor.

Getting Started

Start Xcode and create a “single view” project to work within.



In the Deployment Info section of the project's General tab, ensure only the Portrait option is selected in the Device Orientation options section.

1. Utils Class

Create a Swift class called Utils. We use this as a container for some useful generic functions

```
import UIKit
class Utils {
    static let sharedInstance = Utils()
    func info(message: String, ui: UIViewController, cbOK: @escaping () -> Void) {
        let dialog = UIAlertController(title: "Information", message: message, preferredStyle:
UIAlertControllerStyle.alert)
        let OKAction = UIAlertAction(title: "OK", style: .default) { (action) in
            cbOK()
        }
        dialog.addAction(OKAction)
        // Present the dialog
        ui.present(dialog,animated: false, completion: nil)
    }
    func error(message: String, ui: UIViewController, cbOK: @escaping () -> Void) {
        let dialog = UIAlertController(title: "ERROR", message: message, preferredStyle:
UIAlertControllerStyle.alert)
        let OKAction = UIAlertAction(title: "OK", style: .default) { (action) in
            cbOK()
        }
```

```
        dialog.addAction(OKAction)
        // Present the dialog
        ui.present(dialog,animated: false, completion: nil)
    }
}
```

2. Bluetooth Adapter

We'll use a central class as a container for all the primary Bluetooth operations our application will need.

Create a Swift class called BleAdapter.swift and populate it with the following code:

```
import UIKit
import CoreBluetooth

class BLEAdapter: NSObject,CBCentralManagerDelegate, CBPeripheralDelegate {
    var central_manager: CBCentralManager?
    var selected_peripheral: CBPeripheral?
    var peripherals: NSMutableArray = []
    var powered_on : Bool?
    var scanning: Bool?
    var connected: Bool?
    static let sharedInstance = BLEAdapter()

    func centralManagerStateToString(_ state: CBManagerState)->[CChar]?
    {
        var returnVal = "Unknown state"
        if(state == CBManagerState.unknown)
        {
            returnVal = "State unknown (CBManagerStateUnknown)"
        }
        else if(state == CBManagerState.resetting)
        {
            returnVal = "State resetting (CBManagerStateUnknown)"
        }
        else if(state == CBManagerState.unsupported)
        {
            returnVal = "State BLE unsupported (CBCentralManagerStateResetting)"
        }
        else if(state == CBManagerState.unauthorized)
        {
            returnVal = "State unauthorized (CBCentralManagerStateUnauthorized)"
        }
```

```
        else if(state == CBManagerState.poweredOff)
        {
            returnVal = "State BLE powered off (CBCentralManagerStatePoweredOff)"
        }
        else if(state == CBManagerState.poweredOn)
        {
            returnVal = "State powered up and ready (CBCentralManagerStatePoweredOn)"
        }
        else
        {
            returnVal = "State unknown"
        }
        return (returnVal.cString(using: String.Encoding.utf8))
    }
    func printKnownPeripherals(){
        print("List of currently known peripherals : ");
        let count = self.peripherals.count
        if(count > 0)
        {
            for i in 0...count - 1
            {
                let p = self.peripherals.object(at: i) as! CBPeripheral
                self.printDeviceDetails(p)
            }
        }
    }
    func printDeviceDetails(_ peripheral: CBPeripheral)
    {
        print("Peripheral Info :");
        print("Name : \(peripheral.name)")
        print("ID   : \(peripheral.identifier)")
    }
    // CBCentralManagerDelegate
    func centralManager(_ central: CBCentralManager, willRestoreState dict: [String : Any]) {
        self.peripherals = dict[CBCentralManagerRestoredStatePeripheralsKey] as! NSMutableArray;
    }
    func centralManagerDidUpdateState(_ central: CBCentralManager) {
    }
}
```

Exercise 1 - Scanning for peripheral devices

1. Scanning and Scan Results Screen

Delete whatever the Main.storyboard currently contains. Drag a TableViewController into the main storyboard. Select Editor / Embed in / NavigationController. This will add a NavigationController to the story board and link it to the TableViewController.

Select the Navigation Controller in the storyboard. In the Attributes Inspector select the Is Initial View Controller checkbox.

2. Checkpoint

Check your application builds with no compilation errors. Plug an iPhone or iPad into your computer via USB and run the application, with the plugged in device selected as the destination. Note that using a simulator will work at this stage but not when we have Bluetooth functions to test so we'll start as we mean to go on and use a physical device for all testing. Your application should run and present a screen with horizontal lines, indicating rows of an empty table.

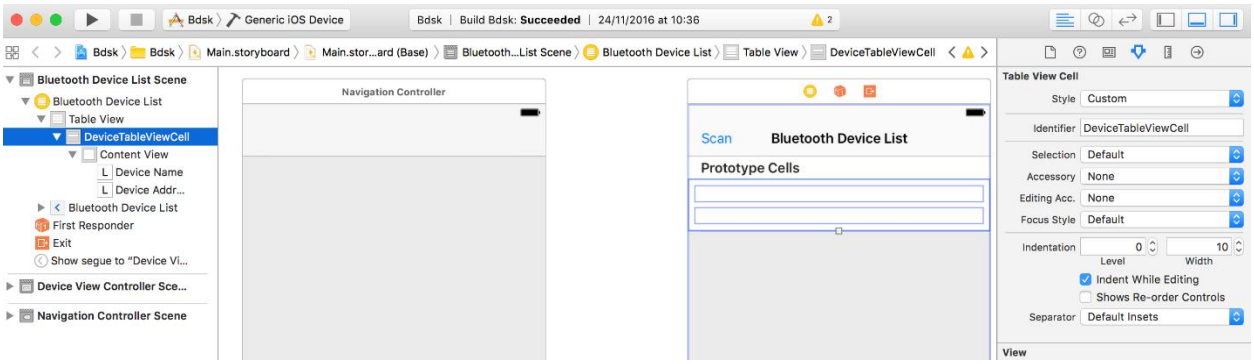
3. TableViewController storyboard completion

Next, drag a Bar Button Item to the Navigation Item area above the Prototype Cells in the TableViewController. Change its title to 'Scan'.

Change the title of the Navigation Item component to 'Bluetooth Device List'

Drag two Label views to the Prototype Cells area, adjusting its height as necessary. Delete the default text value from each of these labels. We'll use one for a Bluetooth device name and the other for the device's Bluetooth device address.

In the Document Outline, change the name of the Table View Controller to Bluetooth Device List, Table View Cell to DeviceTableViewCell and the label names to Device Name and Device Address.



4. Device and Device UI Classes

Device

Create a Swift file called Device.swift with the following code in it:

```
import UIKit
import CoreBluetooth
class Device {
    // MARK: Properties
    var peripheral: CBPeripheral
    var deviceAddress: UUID
    var deviceName: String
    // MARK: Initialization

    init(peripheral: CBPeripheral, deviceAddress: UUID, deviceName: String?) {
        self.peripheral = peripheral
        self.deviceAddress = deviceAddress
        if let dn = deviceName {
            self.deviceName = dn
        } else {
            self.deviceName = "no name"
        }
    }
}
```

DeviceViewTableCell

Create a Cocoa Touch class called DeviceTableViewCell which must be a subclass of UITableViewCell. Initially it should look like this:

```
import UIKit
class DeviceTableViewCell: UITableViewCell {
    override func awakeFromNib() {
        super.awakeFromNib()
        // Initialization code
    }
    override func setSelected(_ selected: Bool, animated: Bool) {
        super.setSelected(selected, animated: animated)
        // Configure the view for the selected state
    }
}
```

Device table cells

Select the Device Table View Cell element in the document outline and in the Attributes Inspector, set its identifier to 'DeviceTableViewCell'. In the Identity Inspector set the class name to 'DeviceTableViewCell'.

Open the Assistant Editor and arrange Xcode so that both the main storyboard and the DeviceTableViewCell swift file are open.

Create outlets for the labels in the table view cell. ctrl+drag from each of the labels in the device table view cell in the storyboard into the swift code so that afterwards it looks like this:

```
import UIKit

class DeviceTableViewCell: UITableViewCell {

    // MARK: Properties

    @IBOutlet weak var deviceName: UILabel!
    @IBOutlet weak var deviceAddress: UILabel!

    override func awakeFromNib() {
        super.awakeFromNib()
        // Initialization code
    }

    override func setSelected(_ selected: Bool, animated: Bool) {
        super.setSelected(selected, animated: animated)
        // Configure the view for the selected state
    }
}
```

DeviceListViewController

You should have a class called ViewController.swift which was generated for you. Delete it.

Create a new Cocoa Touch class called DeviceListViewController and make it a subclass of UITableViewController.

Select the TableViewController in the storyboard. In the Identity Inspector in the Custom Class section, change the class name to DeviceListViewController.

Add the following member variables to the DeviceListViewController class:

```
var adapter: BLEAdapter!
var utils: Utils!
var devices :NSMutableArray = []
var scan_timer = Timer()
```

Change the function numberOfSections so that it returns 1.

Change the function with signature "tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int" so that it returns devices.count.

Uncomment the function with signature "func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell" and add the code shown below so that individual rows of the table in our view are populated with data from a corresponding Device object in the devices array:

```
    override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier: "DeviceTableViewCell", for:
indexPath) as! DeviceTableViewCell
        let device = devices.object(at: indexPath.row) as! Device
        cell.deviceName.text = device.deviceName
        cell.deviceAddress.text = device.deviceAddress.uuidString
        return cell
    }
```

5. Checkpoint

Build your project to check for compilation errors. There should be none at this stage.

6. Bluetooth initialisation

Edit info.plist and add the following inside the <dict> element:

```
<key>UIBackgroundModes</key>
<array>
       <string>bluetooth-central</string>
</array>
```

This allows Bluetooth operations such as scanning to continue even if you switch the app into the background.

Add the following function to BleAdapter:

```
    func initBluetooth(_ device_list : DeviceListViewController)->Int
    {
        powered_on = false
        scanning = false
        connected = false
        self.central_manager = CBCentralManager(delegate: self, queue: nil, options:
[CBCentralManagerOptionRestoreIdentifierKey:"BDSK"])
        return 0
    }
```

Note that the CBCentralManager initialisation takes three parameters; a reference to a delegate (instance of CBCentralManagerDelegate), a dispatch queue which here we're setting to nil and some options. The option specified here ("BDSK") is a string containing a UID for the central manager that is being instantiated.

Replace the didUpdateState function in BleAdapter with the following:

```
    func centralManagerDidUpdateState(_ central: CBCentralManager) {
        if(central.state == CBManagerState.poweredOn)
        {
            powered_on = true
            return
        }
        if(central.state == CBManagerState.poweredOff)
        {
            powered_on = false
            return
        }
    }
```

Add the following code to the viewDidLoad function in DeviceListViewController:

```
        adapter = BLEAdapter.sharedInstance
        adapter.initBluetooth(self)
        utils = Utils.sharedInstance
```

This obtains an instance of BleAdapter and calls its initBluetooth function. It also obtains a Utils object.

7. Checkpoint

Build and run on your device to check there are no 'surprises'.

8. Scanning

ScanResultsConsumer

Add the following protocol declaration to BleAdapter above the class declaration:

```
protocol ScanResultsConsumer {
    func onDeviceDiscovered(_ device: CBPeripheral)
}
```

Add the following variable definition:

```
var scan_results_consumer: ScanResultsConsumer?
```

Change the class declaration for DeviceListViewController so that it adopts the ScanResultsConsumer protocol:

```
class DeviceListViewController: UITableViewController, ScanResultsConsumer {
```

and add the following function to satisfy the ScanResultsConsumer contract:

```
    func onDeviceDiscovered(_ peripheral: CBPeripheral) {
        if let discovered_device_name = peripheral.name {
            let device = Device(peripheral: peripheral, deviceAddress: peripheral.identifier,
deviceName: discovered_device_name)
            devices.insert(device, at: 0)
        } else {
            let device = Device(peripheral: peripheral, deviceAddress: peripheral.identifier,
deviceName: "<no name>")
            devices.insert(device, at: 0)
        }
        let indexPath = IndexPath(row: 0, section: 0)
        var indexesPath:[IndexPath] = [IndexPath]()
        indexesPath.append(indexPath)
        self.tableView.insertRows(at: indexesPath, with: UITableViewRowAnimation.automatic)
    }
```

Note the use of 'if let'. Not all Bluetooth devices have a name and this construct therefore treats peripheral.name as a Swift Optional and ensures we place a suitable value into the table cell for this peripheral's name, regardless of whether it actually has one or not.

Now add the following import statement to DeviceListViewController:

```
import CoreBluetooth
```

CoreBluetooth is the library which contains all the Bluetooth classes we'll be using
Update the BleAdapter initBluetooth function by adding the following:

```
self.scan_results_consumer = device_list
```

9. Checkpoint

Build and check for errors.

10. Scanning Functions

Add the following functions to BleAdapter:

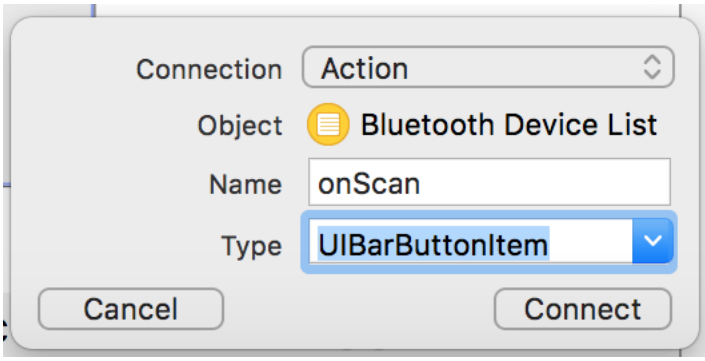
```
    func findDevices(_ timeout: Int, _ name: String, _ consumer: ScanResultsConsumer)->Int
    {
        if(self.central_manager?.state != CBManagerState.poweredOn)
        {
            print("Bluetooth is not powered on");
            return -1
        }

        peripherals.removeAllObjects()

        Timer.scheduledTimer(timeInterval: Double(timeout), target: self, selector:
#selector(BLEAdapter.stopScanning(_:)), userInfo: nil, repeats: false)

        scanning = true
        self.central_manager?.scanForPeripherals(withServices: nil, options: nil)
        return 0

    }

    @objc func stopScanning(_ timer: Timer)
    {
        if (scanning == true) {
            self.central_manager?.stopScan()
            scanning = false
        }
    }

    func stopScanning()
    {
        if (scanning == true) {
            self.central_manager?.stopScan()
            scanning = false
        }
    }
```

The first function initiates scanning for Bluetooth peripheral devices and will be called from the DeviceListViewController. It uses a Timer to call the second function after the time specified in the timeout parameter and this tells the CBCentralManager object to stop scanning.

11. The Scan Button

Select the main storyboard, open the Assistant Editor and select the DeviceListViewController swift file

Ctrl+Drag from the Scan button in the storyboard to the swift file to create an action called onScan with a UIBarButtonItem as argument.



The resultant function should have a signature which looks like this:

```
@IBAction func onScan(_ sender: UIBarButtonItem) {
```

Update DeviceListViewController by replacing the onScan function with the code shown here and add the scanningFinished function which follows:

```
    @IBAction func onScan(_ sender: UIBarButtonItem) {
        if (adapter.scanning == true) {
            print("Already scanning - ignoring")
            return
        }
        if (adapter.powered_on == false) {
            utils.info(message : "Bluetooth is not available yet - is it switched on?",
                       ui : self,
                       cbOK: {
                        print("OK callback")
            })
            return
        }
        if(adapter.scanning == false) {
            adapter.scanning = true
            print("Will start scanning shortly")
            devices.removeAllObjects()
            self.tableView.reloadData()
```

```
            // scan for 10 seconds
            let rc = adapter.findDevices(10,"BDSK", self)

            if (rc == -1) {
                utils.info(message : "Bluetooth is not available - is it switched on?",
                           ui : self,
                           cbOK: {
                            print("OK callback")
                })
            } else {
                print("Setting up timer for when scanning is finished")
                scan_timer = Timer.scheduledTimer(
                    timeInterval: 10.0,
                    target: self,
                    selector: #selector(DeviceListViewController.scanningFinished(_:)),
                    userInfo: nil,
                    repeats: false)
                }
        }

    }

    @objc func scanningFinished(_ timer: Timer)
    {
        print("Finished scanning")
        adapter.scanning = false
        if(adapter.peripherals.count > 0)
        {
            let msg = "Finished scanning - found " + String(adapter.peripherals.count) + "
devices"
            utils.info(message : msg, ui : self,
                       cbOK: {
                        print("OK callback")
            })
        } else {
            let msg = "No devices were found"
            utils.info(message : msg, ui : self,
                       cbOK: {
                        print("OK callback")
            })
        }
    }
```

Add the following function to the BleAdapter class:

```
    func centralManager(_ central: CBCentralManager, didDiscover peripheral: CBPeripheral,
advertisementData: [String : Any], rssi RSSI: NSNumber) {


        printDeviceDetails(peripheral);
        var i = 0
        while i < self.peripherals.count
        {
            let p = self.peripherals.object(at: i)
            if((p as AnyObject).identifier == peripheral.identifier)
            {
                self.peripherals.replaceObject(at: i, with: peripheral)
                return
            }
            i = i + 1
        }
        // did not find device in our array so it must be a new device
        self.peripherals.add(peripheral)
        scan_results_consumer?.onDeviceDiscovered(peripheral)
    }
```

didDiscover is a delegate function, called by the CBCentralManager object. In here, we add it to our list of peripherals, provided it has not already been discovered and then make a call to the DeviceListViewController's onDeviceDiscovered function to tell it to add the newly discovered device to its list.

12. Checkpoint

Build, install and test. Pressing the Scan button should cause scanning for Bluetooth Low Energy devices to be performed for 10 seconds and any devices in range discovered and listed. Make sure you have at least one Bluetooth Low Energy peripheral switched on, in range and advertising for this test!

13. Device Filtering

All we have to do now to complete implementation of device discovery is to filter out the devices we are evidently not interested in and list only those we think we should be interested in. There are various ways of doing this, depending on how the target device type is advertising. We could have specified required service UUIDs for example. In our case though, the BDSK device is advertising its name in the advertising packet so we'll use that to perform application layer filtering.

In BleAdapter add the following variable declaration:

```
var required_name : String?
```

Update findDevices so that the specified device name argument is stored in the new variable after checking that Bluetooth is on, like this:

```
    func findDevices(_ timeout: Int, _ name: String, _ consumer: ScanResultsConsumer)->Int
    {
        if(self.central_manager?.state != CBManagerState.poweredOn)
        {
            print("Bluetooth is not powered on");
            return -1
        }

        peripherals.removeAllObjects()

        required_name = name
```

Update the didDiscover function to filter on device name like this:

```
    func centralManager(_ central: CBCentralManager, didDiscover peripheral: CBPeripheral,
advertisementData: [String : Any], rssi RSSI: NSNumber) {
        if let localname = advertisementData["kCBAdvDataLocalName"] {
            print(advertisementData["kCBAdvDataLocalName"] as! String)
            if ((advertisementData["kCBAdvDataLocalName"]  as! String) != required_name) {
                return;
            }
        } else {
            return;
        }
```

Note that advertisementData is a dictionary containing advertising data fields from the received advertising packet.

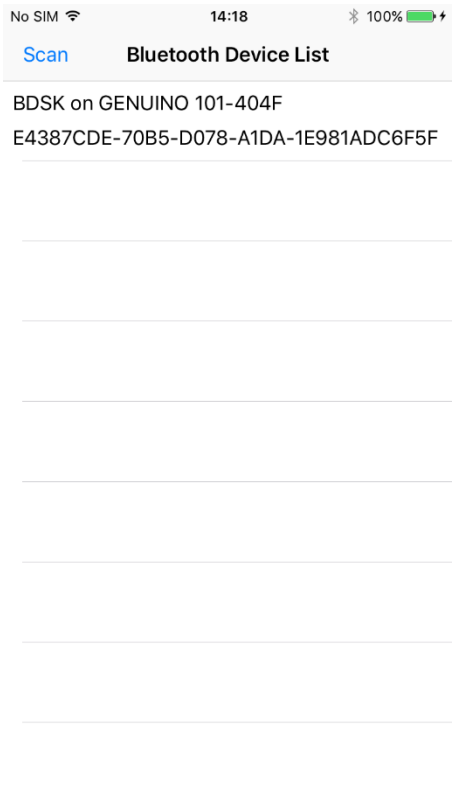
In DeviceListViewController, replace the onDeviceDiscovered function with the following code so that device name in table cells is prefixed by "BDSK on ". Note that we no longer need to be concerned about the optionality of peripheral.name either, so we can simplify our code and use forced unwrap instead.

```
    func onDeviceDiscovered(_ peripheral: CBPeripheral) {
        var device_name = "BDSK on " + peripheral.name!
        let device = Device(peripheral: peripheral, deviceAddress: peripheral.identifier,
deviceName: device_name)
        devices.insert(device, at: 0)
        let indexPath = IndexPath(row: 0, section: 0)
        var indexesPath:[IndexPath] = [IndexPath]()
```

```
        indexesPath.append(indexPath)
        self.tableView.insertRows(at: indexesPath, with: UITableViewRowAnimation.automatic)
    }
```

14. Checkpoint

The application should now be able to scan for 10 seconds and list only devices which are advertising with "BDSK" as the Complete Local Name advertising data field.



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 View Controller and we'll implement any required Bluetooth operations in the BleAdapter class.

1. Creating and Navigating to DeviceViewController

The next task is to add a new scene which will allow the control and monitoring of a selected BDSK device. Touching a listed BDSK device in the DeviceListViewController will cause navigation to the new scene.

Add a new ViewController to the storyboard to the right of the DeviceListViewController

Create a segue from the device list scene's prototype cell component to the new view controller. Do this by selecting the DeviceTableViewCell component and ctrl+drag to the new view controller. From the pop-up dialogue which appears, select Show.

Select the resultant segue in the storyboard and in the Attributes inspector, set the identifier attribute to 'showDeviceControlSegue'

2. Custom View Controller Class

Create a new cocoa touch class with the name 'DeviceViewController'. Ensure it is a subclass of UIViewController

In the storyboard, select the new View Controller and in the Identity Inspector set the class name to 'DeviceViewController'.

3. Checkpoint

Test by scanning and then select any BDSK device that is listed. The new view controller, currently blank, should appear. You'll probably also get a dialogue indicating scanning has finished popping up over the new scene. This is because we didn't cancel scanning when we navigated away from the device list scene. We'll do that next.

4. Cancelling Scanning

Uncomment the ‘prepare’ function in DeviceListViewController and replace it with the following code:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    // Get the new view controller using segue.destinationViewController.
    // Pass the selected object to the new view controller.

    if (adapter.scanning == true) {
        print("stopping scanning")
        adapter.stopScanning()
        scan_timer.invalidate()
    }
}
```

5. Passing the selected CBPeripheral object to the BleAdapter

Update the prepare function in DeviceListViewController to pass the selected CBPeripheral object to the BleAdapter.

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    // Get the new view controller using segue.destinationViewController.
    // Pass the selected object to the new view controller.

    if (adapter.scanning == true) {
        print("stopping scanning")
        adapter.stopScanning()
        scan_timer.invalidate()
    }

    if let index = tableView.indexPathForSelectedRow?.row {
        let selected_device = devices[index] as! Device
        adapter.selected_peripheral = selected_device.peripheral
    }
}
```

6. The DeviceViewController

In the main storyboard, add a Label to the top of the controller.

Open the Assistant Editor and select the DeviceViewController swift file.

Ctrl+drag from the new label to the swift file and create an outlet. Name it 'device_details'.

Create a member variable for an instance of BleAdapter and obtain a shared instance near the start of viewDidLoad.

Create a member variable for an instance of Utils and obtain a shared instance near the start of viewDidLoad.

Update the viewDidLoad function to set the new device_details text attribute to a suitable string which contains the UUID of the selected device (which the BleAdapter object contains).

The top part of DeviceViewController should now look something like this:

```
class DeviceViewController: UIViewController {


    var adapter: BLEAdapter!
    var utils: Utils!


    // MARK: properties
    @IBOutlet weak var device_details: UILabel!



    override func viewDidLoad() {
        super.viewDidLoad()
        adapter = BLEAdapter.sharedInstance
        utils = Utils.sharedInstance
        device_details.text = "Device: BDSK [" +
adapter.selected_peripheral!.identifier.uuidString + "]"
    }
```

7. Checkpoint

Test your application now and make sure the DeviceViewController displays details of the selected device.

8. UI Completion

Next we need to build the remainder of the UI. This means adding an RSSI indicator and 'proximity classification panel', some buttons which will allow Alert Levels to be set, a switch that lets 'proximity sharing' be switched on or off, a second switch with which to enable or disable temperature monitoring, a button which will cause the BDSK device to make a noise (Immediate Alert) and last but not least, a button we can press to connect our iOS device to the BDSK device over Bluetooth.

Proceed as follows, working from the top to the bottom of the screen:

Add a label with the text "RSSI: unavailable" in it. Create an outlet for it in DeviceViewController called 'rssi'.

Add a View and make it a rectangle which occupies the full width of the screen. Create an outlet called 'proximity_classification'. Set the background colour to grey.

Add three buttons arranged horizontally with titles 'Low', 'Medium' and 'High' respectively. Create actions for each called onLow, onMedium and onHigh. Create outlets for each entitled btn_low, btn_medium and btn_high.

Add a label with the text "Share" and next to it a Switch with the default value Off. Create an action for the switch called 'onShareChanged'. Create an outlet called switch_share.

Add a label with text value "Monitor Temperature" and next to it a Switch component with value Off. Create an outlet called switch_temperature and create an action called onTemperatureMonitoringChanged. Add another label, centred and under the previous label and switch with the default text "not available”. Create an outlet for this label called temperature.

Add a label not far from the bottom of the screen (leave room for a couple of buttons under it). Size it to fill the full width and clear its default value. Create a outlet called 'status'.

Add a button with text "MAKE A NOISE" and centre it under the status label. Create an action called onMakeNoise. Create an outlet called btn_noise.

Add a button with text "CONNECT" and centre it under the MAKE A NOISE button. Create an action called onConnect. Create an outlet called btn_connect.

Add print statements to each of the new actions to provide an easy way of verifying they are working.

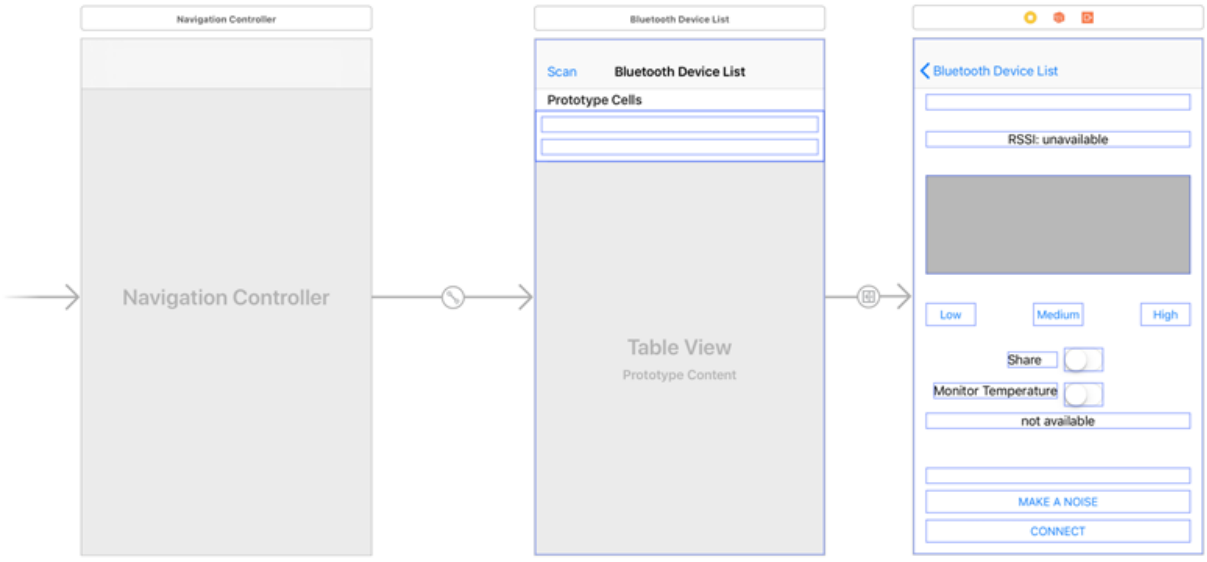
Your DeviceViewController code should currently look like this:

```
import UIKit
class DeviceViewController: UIViewController {
    var adapter: BLEAdapter!
    var utils: Utils!
    @IBOutlet weak var device_details: UILabel!
    @IBOutlet weak var rssi: UILabel!
    @IBOutlet weak var proximity_classification: UIView!
    @IBOutlet weak var btn_low: UIButton!
    @IBOutlet weak var btn_medium: UIButton!
    @IBOutlet weak var btn_high: UIButton!
    @IBOutlet weak var switch_share: UISwitch!
    @IBOutlet weak var switch_temperature: UISwitch!
    @IBOutlet weak var temperature: UILabel!

    @IBOutlet weak var status: UILabel!
    @IBOutlet weak var btn_noise: UIButton!
    @IBOutlet weak var btn_connect: UIButton!

    override func viewDidLoad() {
        super.viewDidLoad()
        adapter = BLEAdapter.sharedInstance
        utils = Utils.sharedInstance
        device_details.text = "Device: BDSK [" +
           adapter.selected_peripheral!.identifier.uuidString + "]"
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }

    @IBAction func onLow(_ sender: UIButton) {
        print("onLow")
    }

    @IBAction func onMedium(_ sender: UIButton) {
        print("onMedium")
    }

    @IBAction func onHigh(_ sender: UIButton) {
```

```
        print("onHigh")
    }

    @IBAction func onShareChanged(_ sender: UISwitch) {
        print("onShareChanged")
    }

    @IBAction func onTemperatureMonitoringChanged(_ sender: UISwitch) {
        print("onTemperatureMonitoringChanged \(sender.isOn)")
    }
    @IBAction func onMakeNoise(_ sender: UIButton) {
        print("onMakeNoise")
    }

    @IBAction func onConnect(_ sender: UIButton) {
        print("onConnect")
    }
}
```

9. Checkpoint

The main storyboard should now look like this:



And details of each scene like this:

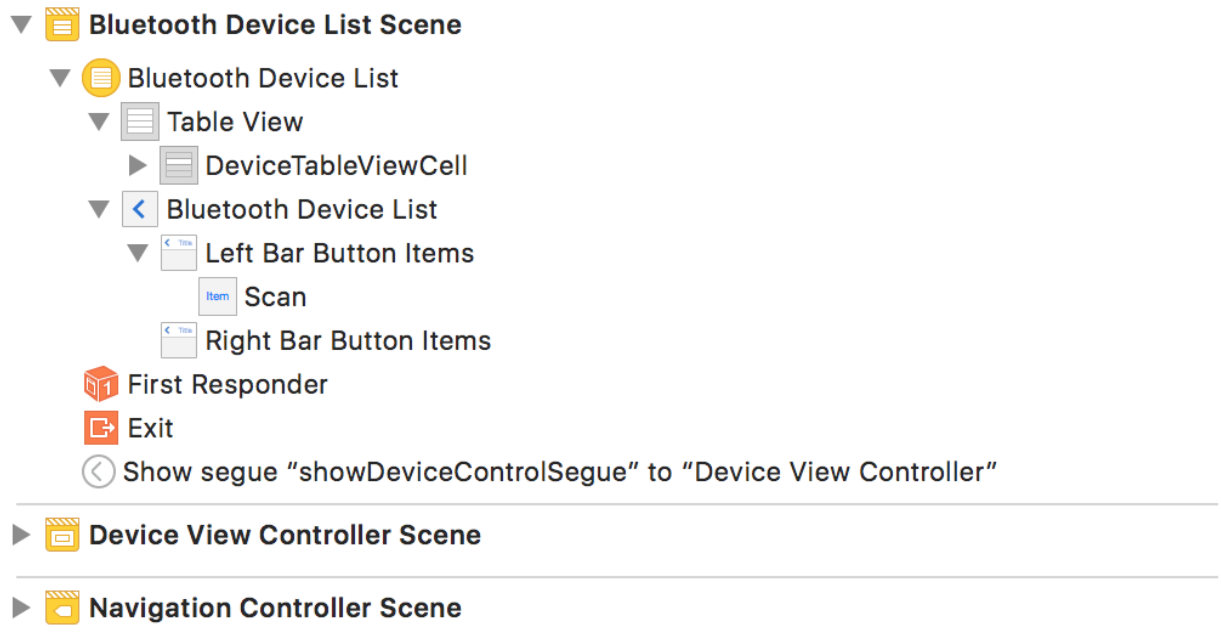


Figure 1 - device list scene composition

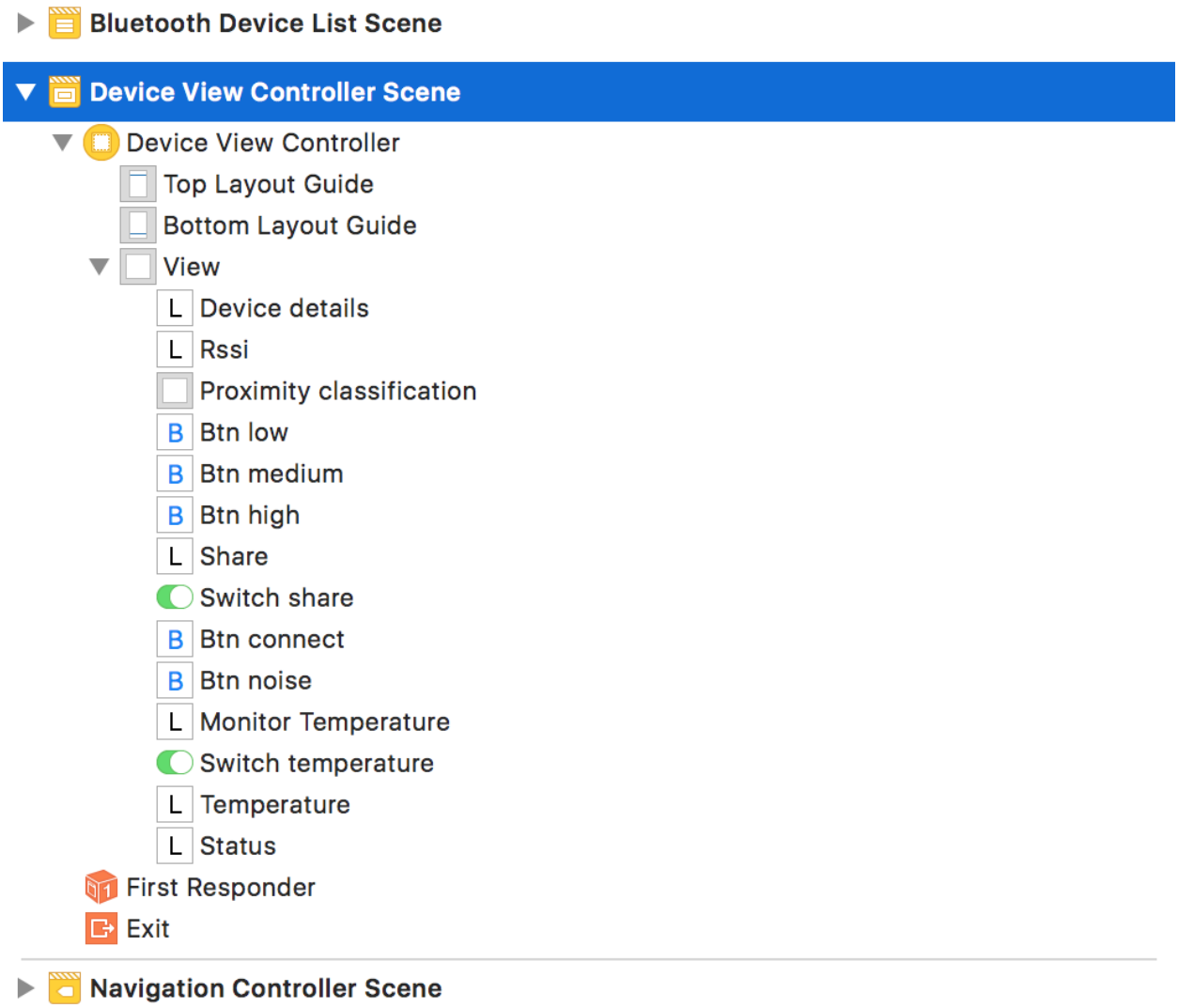


Figure 2 - device view controller composition

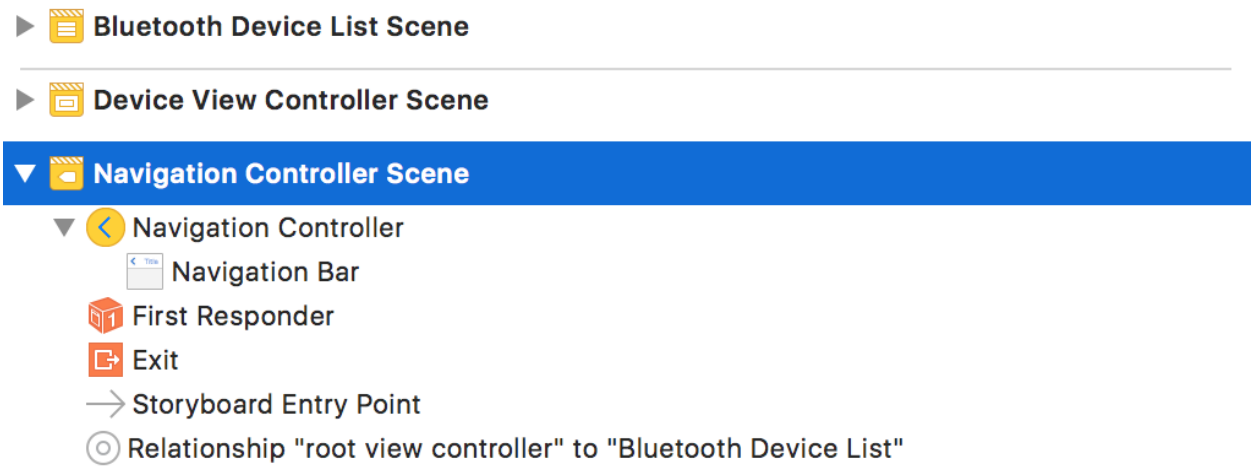


Figure 3 - navigation controller composition

Test the application now, exercising all the new UI components which should be connected to actions and watching the Xcode console to verify the expected result.

10. Bluetooth Operations

DeviceViewController will initiate various Bluetooth operations via calls to new functions we'll need to add to BleAdapter. It will need to know the results of these operations in many cases. Add the following protocol declaration to BleAdapter.

```
protocol BluetoothOperationsConsumer {
    func onConnected()
    func onFailedToConnect(_ error: Error?)
    func onDisconnected()
}
```

Add an instance variable of this type to BleAdapter

```
var bluetooth_operations_consumer: BluetoothOperationsConsumer?
```

Update the DeviceViewController class declaration so that it indicates it adopts this protocol.

```
class DeviceViewController: UIViewController, BluetoothOperationsConsumer {
```

And add skeleton implementations of the functions defined by the protocol (include print statements so you can perform a quick test if you want to).

11. Connecting and Disconnecting

UI State Management

BleAdapter will track whether or not we're connected to the BDSK device and we'll use this state information to adjust the availability and appearance of controls on the UI

Most UI components should be disabled until we've established a Bluetooth connection. Set defaults as shown in the DeviceViewController's viewDidLoad function:

```
override func viewDidLoad() {
    super.viewDidLoad()
    device_details.text = "Device: BDSK [" + peripheral!.identifier.uuidString + "]"
    btn_low.isEnabled = false
    btn_medium.isEnabled = false
    btn_high.isEnabled = false
    btn_noise.isEnabled = false
    switch_share.isEnabled = false
    switch_temperature.isEnabled = false
}
```

Checkpoint

Test and verify that the UI initial states are as expected.

Adapter API - Connecting and Disconnecting

Add functions to BleAdapter that will allow connections to be established or cancelled.

```
func connect(_ result_consumer: BluetoothOperationsConsumer)
{
    if (selected_peripheral != nil) {
        selected_peripheral?.delegate = self
        bluetooth_operations_consumer = result_consumer
        central_manager?.connect(selected_peripheral!, options: nil)
    }
}

func disconnect(_ result_consumer: BluetoothOperationsConsumer)
{
    if (selected_peripheral != nil) {
        bluetooth_operations_consumer = result_consumer
        central_manager?.cancelPeripheralConnection(selected_peripheral!)
```

```
        }
    }
```

Note that a BluetoothOperationConsumer object is required so that the result of the requested operation can be sent back to the originator object.

Note that we set the BleAdapter object as the delegate for the CBPeripheral selected_peripheral object.

Add implementations of the following functions of the CBCentralManagerDelegate class like this:

```
    func centralManager(_ central: CBCentralManager, didConnect peripheral: CBPeripheral) {
        print("Connected")
        connected = true
        bluetooth_operations_consumer?.onConnected()
    }


    func centralManager(_ central: CBCentralManager, didFailToConnect peripheral: CBPeripheral,
error: Error?) {
        print("didFailToConnect")
        bluetooth_operations_consumer?.onFailedToConnect(error)
    }


    func centralManager(_ central: CBCentralManager, didDisconnectPeripheral peripheral:
CBPeripheral, error: Error?) {
        print("Disconnected")
        connected = false
        bluetooth_operations_consumer?.onDisconnected()
    }
```

Checkpoint

Build to check your code has no compilation errors.

UI state and connection state

As stated, we intend to manage the state of various UI components according to the state of our Bluetooth connection. Change the onConnect action so that it triggers connecting or disconnecting appropriately.

```
    @IBAction func onConnect(_ sender: UIButton) {
        print("onConnect")
        if (adapter.connected == false) {
            adapter.connect(self)
        } else {
            adapter.disconnect(self)
        }
```

```
}
```

Change onConnected and onDisconnected so that the text on the Connect button is set appropriately and the state of some of the UI components also set correctly and display an error dialogue in onFailedToConnect.

```
func onConnected() {
    print("onConnected")
    btn_connect.setTitle("DISCONNECT", for: .normal)
}

func onFailedToConnect(_ error: Error?) {
    print("onFailedToConnect")
    utils.error(message : "Failed to connect:  \(error)",
                ui : self,
                cbOK: {
    })
}

func onDisconnected() {
    print("onDisconnected")
    btn_connect.setTitle("CONNECT", for: .normal)
    btn_low.isEnabled = false
    btn_medium.isEnabled = false
    btn_high.isEnabled = false
    btn_noise.isEnabled = false
    switch_share.isEnabled = false
    switch_temperature.isEnabled = false
}
```

Navigating back to the device list scene

When the user selects the back button to navigate back to the scene where we scan for device, we need to disconnect from the peripheral device if we're currently connected. Add the following function, which will take care of this, to DeviceViewController.swift

```
override func viewWillDisappear(_ animated : Bool) {
    super.viewWillDisappear(animated)

    if self.isMovingFromParentViewController {
        if (adapter.connected == true) {
            print("disconnecting")
            adapter.disconnect(self)
        }
```

```
        }
    }
```

Checkpoint

Build to ensure no compilation errors have sneaked in.

12. RSSI Polling

Next we're going to periodically poll the RSSI value and use this to set the colour of the proximity_classification view to a colour which indicates how near or far we are from the BDSK device:

Green (near), yellow (medium distance) and red (far away but still in range).

BleAdapter will implement new functions which allow the DeviceViewController to start and stop RSSI polling and to read RSSI values periodically. Having read an RSSI value though, it will need to communicate that value to the DeviceViewController and so we need to add a new function to the BluetoothOperationConsumer protocol. Add the onRssiValue function as shown here:

```
protocol BluetoothOperationsConsumer {
    func onConnected()
    func onFailedToConnect(_ error: Error?)
    func onDisconnected()
    func onRssiValue(_ rssi: NSNumber)
}
```

We'll use a Timer object to control requesting the RSSI value every 1 second. Add a Timer variable declaration to BleAdapter:

```
    var rssi_timer = Timer()
```

Next, we'll add functions to BleAdapter which can be called by DeviceViewController when a connection is established or lost, respectively. The point here is that RSSI values are only available when in a connection.

```
    func startPollingRssi(_ result_consumer: BluetoothOperationsConsumer) {
        rssi_timer = Timer.scheduledTimer(withTimeInterval: 1, repeats: true) {
            (_) in
            self.selected_peripheral?.readRSSI()
        }
    }

    func stopPollingRssi() {
```

```
        rssi_timer.invalidate()
    }
```

Calls to readRSSI() will asynchronously return a value to the delegate method didReadRSSI. Implement it in BleAdapter as shown:

```
    func peripheral(_ peripheral: CBPeripheral, didReadRSSI RSSI: NSNumber, error: Error?) {
        print("didReadRssi")
        bluetooth_operations_consumer?.onRssiValue(RSSI)
    }
```

All we do here is pass the rssi value to DeviceViewController so it can make use of it.

In DeviceViewController add the following function which we'll make responsible for updating the UI whenever we have a new RSSI value.

```
 func updateProximityClassification(_ rssi_value: NSNumber) {
        var proximity_band = 3;
        rssi.text = "RSSI: \(rssi_value.floatValue.description) dBm"
        if (rssi_value.floatValue < -80.0) {
            proximity_classification.backgroundColor = UIColor.red
        } else if (rssi_value.floatValue < -50.0) {
            proximity_classification.backgroundColor = UIColor.yellow
            proximity_band = 2;
        } else {
            proximity_classification.backgroundColor = UIColor.green
            proximity_band = 1;
        }
}
```

Don't worry about the unused proximity_band variable. We'll be using it later.

Implement onRssiValue in the view controller and call the updateProximityClassification function as shown:

```
    func onRssiValue(_ rssi: NSNumber) {
        print("Received RSSI value \(rssi)")
        updateProximityClassification(rssi)
    }
```

We need to reset the RSSI UI text and the colour of the proximity classification rectangle when the connection drops so update onDisconnected with the following additional lines of code.

```
        rssi.text = "RSSI: unavailable"
        proximity_classification.backgroundColor = UIColor.gray
    }
```

Update onConnected in the view controller so that it starts RSSI polling using our new function.

```
    func onConnected() {
        print("onConnected")
        btn_connect.setTitle("DISCONNECT", for: .normal)
        adapter.startPollingRssi(self)
    }
```

Update onDisconnected so that we stop RSSI polling when disconnected. Add a call to the stopPollingRssi function like this:

```
        adapter.stopPollingRssi()
```

13. Checkpoint

Test the application. You should see the proximity classification rectangle change colour as you move away from the BDSK device and back towards it and you should also see the RSSI value updated on the screen above it. When you disconnect, the rectangle should go gray in colour and the RSSI text be replaced with "RSSI: unavailable".

14. Service and Characteristic Discovery

Before we can start working with the Alert Level characteristic or any other characteristic for that matter we need to perform service and characteristic discovery. To that end, we'll add some new functions to BleAdapter and make use of them from DeviceViewController as the state of our application and its use of Bluetooth progresses. We'll also implement delegate functions to handle the results of the new operations and modify our BluetoothOperationsConsumer protocol so that BleAdapter can communicate the new types of result to the view controller.

Add the following variable and constant declarations to BleAdapter:

```
    let IMMEDIATE_ALERT_SERVICE_UUID      = "00001802-0000-1000-8000-00805F9B34FB"
    let LINK_LOSS_SERVICE_UUID            = "00001803-0000-1000-8000-00805F9B34FB"
    let TX_POWER_SERVICE_UUID             = "00001804-0000-1000-8000-00805F9B34FB"
    let PROXIMITY_MONITORING_SERVICE_UUID = "3E099910-293F-11E4-93BD-AFD0FE6D1DFD"
    let HEALTH_THERMOMETER_SERVICE_UUID   = "00001809-0000-1000-8000-00805F9B34FB"
    let TEMPERATURE_MEASUREMENT_CHARACTERISTIC = "00002A1C-0000-1000-8000-00805F9B34FB"
    let ALERT_LEVEL_CHARACTERISTIC        = "00002A06-0000-1000-8000-00805F9B34FB"
    let CLIENT_PROXIMITY_CHARACTERISTIC   = "3E099911-293F-11E4-93BD-AFD0FE6D1DFD"
```

```
let NUM_SERVICES_OF_INTEREST = 4

var ll_alert_level_characteristic: CBCharacteristic?
var ia_alert_level_characteristic: CBCharacteristic?
var client_proximity_characteristic: CBCharacteristic?
var temperature_measurement_characteristic: CBCharacteristic?

var service_char_discovery_count: Int?
```

This gives us definitions of the UUIDs of the services and characteristics we might need to use with the BDSK custom Bluetooth profile. We've also set up CBCharacteristic objects for the characteristic we'll need to interact with.

service_char_discovery_count will be used to count the number of services for which characteristic discovery has been performed so that we know when the process for all services has finished.

Next, add the following new methods to the BluetoothOperationsConsumer protocol:

```
func onServicesDiscovered()
    func onLlAlertLevelDiscovered()
    func onIaAlertLevelDiscovered()
    func onPmClientProximityDiscovered()
    func onHtTemperatureMeasurementDiscovered()
    func onDiscoveryFinished()
```

We'll just inform the view controller that these operations have completed. It can obtain the results direct from the selected_peripheral CBPeripheral object after this.

Add the function discoverServices:

```
func discoverServices() {
    self.selected_peripheral?.discoverServices(nil)
}
```

And its corresponding CBPeripheralDelegate function didDiscoverServices:

```
func peripheral(_ peripheral: CBPeripheral,
                didDiscoverServices error: Error?){
    print("didDiscoverServices")
    bluetooth_operations_consumer?.onServicesDiscovered()
}
```

Update the view controller's onConnected function so that after initiating RSSI polling, it initiates service discovery:

```
func onConnected() {
    print("onConnected")
    btn_connect.setTitle("DISCONNECT", for: .normal)
    adapter.startPollingRssi(self)
    adapter.discoverServices()
}
```

The view controller, on receiving a call to onServicesDiscovered, merely need to tell the BleAdapter to proceed with characteristic discovery for those services we need to work with. So add this function to the view controller:

```
func onServicesDiscovered() {
    print("onServicesDiscovered")
    adapter.discoverCharacteristics()
}
```

We haven't yet implemented discoverCharacterstics() in BleAdapter so do that now along with its delegate function:

```
func discoverCharacteristics() {
    if let services = selected_peripheral?.services {
        service_char_discovery_count = 0
        for service in services {
            if (service.uuid == CBUUID(string: LINK_LOSS_SERVICE_UUID) ||
                service.uuid == CBUUID(string: IMMEDIATE_ALERT_SERVICE_UUID) ||
                service.uuid == CBUUID(string: PROXIMITY_MONITORING_SERVICE_UUID) ||
                service.uuid == CBUUID(string: HEALTH_THERMOMETER_SERVICE_UUID)) {
                selected_peripheral?.discoverCharacteristics(nil, for: service)
            }
        }
    }
}
```

The delegate function corresponding to discoverCharacteristics will be called once for each service. Our implementation will pick out those characteristics our application in interested in, store the CBCharacteristic object as an instance variable and inform the view controller that the characteristic has been found. It, in turn will track the discovery of each of these characteristics using Bools.

```
   func peripheral(_ peripheral: CBPeripheral,
                   didDiscoverCharacteristicsFor service: CBService,
                   error: Error?) {
        let ht_uuid = CBUUID(string: HEALTH_THERMOMETER_SERVICE_UUID)
        let ll_uuid = CBUUID(string: LINK_LOSS_SERVICE_UUID)
        if let characteristics = service.characteristics {
            for characteristic in characteristics {
                print("service: \(service.uuid) characteristic: \(characteristic.uuid)")
                if service.uuid == CBUUID(string: LINK_LOSS_SERVICE_UUID) && characteristic.uuid
== CBUUID(string: ALERT_LEVEL_CHARACTERISTIC) {
                    ll_alert_level_characteristic = characteristic
                    print("discovered link loss alert level")
                    service_char_discovery_count = service_char_discovery_count! + 1
                    bluetooth_operations_consumer?.onLlAlertLevelDiscovered()
                } else if service.uuid == CBUUID(string: IMMEDIATE_ALERT_SERVICE_UUID) &&
characteristic.uuid == CBUUID(string: ALERT_LEVEL_CHARACTERISTIC) {
                    ia_alert_level_characteristic = characteristic
                    print("discovered immediate alert alert level")
                    service_char_discovery_count = service_char_discovery_count! + 1
                    bluetooth_operations_consumer?.onIaAlertLevelDiscovered()
                } else if service.uuid == CBUUID(string: PROXIMITY_MONITORING_SERVICE_UUID) &&
characteristic.uuid == CBUUID(string: CLIENT_PROXIMITY_CHARACTERISTIC) {
                    client_proximity_characteristic = characteristic
                    print("discovered proximity monitoring client proximity")
                    service_char_discovery_count = service_char_discovery_count! + 1
                    bluetooth_operations_consumer?.onPmClientProximityDiscovered()
                } else if service.uuid == CBUUID(string: HEALTH_THERMOMETER_SERVICE_UUID) &&
characteristic.uuid == CBUUID(string: TEMPERATURE_MEASUREMENT_CHARACTERISTIC) {
                    temperature_measurement_characteristic = characteristic
                    print("discovered temperature measurement client proximity")
                    service_char_discovery_count = service_char_discovery_count! + 1
                    bluetooth_operations_consumer?.onHtTemperatureMeasurementDiscovered()
                }
            }
            if (service_char_discovery_count == NUM_SERVICES_OF_INTEREST) {
                bluetooth_operations_consumer?.onDiscoveryFinished()
            }
        }
    }
```

Add the following Bools to the view controller and initialise them to false in viewDidLoad:

```
    var got_ll_alert_level: Bool?
```

```
    var got_ia_alert_level: Bool?
    var got_pm_client_proximity: Bool?
    var got_ht_temperature_measurement: Bool?
```

```
    override func viewDidLoad() {
        super.viewDidLoad()
        adapter = BLEAdapter.sharedInstance
        utils = Utils.sharedInstance
        device_details.text = "Device: BDSK [" +
adapter.selected_peripheral!.identifier.uuidString + "]"
        adapter.connected = false
        btn_low.isEnabled = false
        btn_medium.isEnabled = false
        btn_high.isEnabled = false
        btn_noise.isEnabled = false
        switch_share.isEnabled = false
        switch_temperature.isEnabled = false
        got_ia_alert_level = false
        got_ll_alert_level = false
        got_pm_client_proximity = false
        got_ht_temperature_measurement = false
    }
```

Implement the functions required to track characteristic discovery, validate the results of the discovery process and update UI state:

```
    func setUiStateValidBdsk() {
        print("setUiStateValidBdsk")
        btn_low.isEnabled = true
        btn_medium.isEnabled = true
        btn_high.isEnabled = true
        btn_noise.isEnabled = true
        switch_share.isEnabled = true
        switch_temperature.isEnabled = true
    }

    func setUiStateInvalidBdsk() {
        print("setUiStateInvalidBdsk")
        btn_low.isEnabled = false
        btn_medium.isEnabled = false
        btn_high.isEnabled = false
```

```
        btn_noise.isEnabled = false
        switch_share.isEnabled = false
        switch_temperature.isEnabled = false
        utils.error(message : "Connected device does not have all the required Bluetooth services
and characteristics",
            ui : self,
            cbOK: {
        })
    }

    func onIaAlertLevelDiscovered() {
        print("onIaAlertLevelDiscovered")
        got_ia_alert_level = true
    }

    func onLlAlertLevelDiscovered() {
        print("onLlAlertLevelDiscovered")
        got_ll_alert_level = true
    }

    func onPmClientProximityDiscovered() {
        print("onPmClientProximityDiscovered")
        got_pm_client_proximity = true
    }

    func onHtTemperatureMeasurementDiscovered() {
        print("onHtTemperatureMeasurementDiscovered")
        got_ht_temperature_measurement = true
    }

    func onDiscoveryFinished() {
        print("onDiscoveryFinished")
        if (got_ia_alert_level == false || got_ll_alert_level == false || got_pm_client_proximity
== false || got_ht_temperature_measurement == false) {
            setUiStateInvalidBdsk()
        } else {
            setUiStateValidBdsk()
        }
  }
```

15. Checkpoint

Test again and watch the console message to ensure discovery is proceeding as expected. For extra points, change your peripheral code so that not all services and/or characteristics are implemented in its profile and test again. The code you just added should detect this and inform you via a dialogue.

16. Link Loss Alert Level

Our next job is to implement the Low/Medium/High buttons. Their purpose is to write a value of 0, 1 or 2 to the Alert Level characteristic of the Link Loss service on the BDSK device and also to set the alert level value which will be written to the Immediate Alert service's Alert Level characteristic when we click the MAKE A NOISE button (that will be our next job).

Read current alert level value

We need to find out what the current value of the Link Loss alert level characteristic is so that the UI can reflect this and we'll do this first. Then we'll make it possible to change the value from those buttons.

Add the following function to the BluetoothOperationsConsumer protocol:

```
    func onLlAlertLevelRead(_ ll_alert_level: UInt8)
```

Add a member variable to hold the link loss service's alert level in the view controller and a value for the default colour of the low|medium|high buttons.

```
    var ll_alert_level: UInt8?
    let default_btn_colour = UIColor(red: 247.0/255.0, green: 249.0/255.0, blue: 249.0/255.0,
alpha: 1.0)
```

Initialise ll_alert_level to 0 in viewDidLoad:

```
        ll_alert_level = 0
```

Implement onLlAlertLevelRead which will store its argument in the member variable and update the UI to highlight the b=low|medium|high button which represents the current value.

```
    func onLlAlertLevelRead(_ ll_alert_level: UInt8) {
        print("onLlAlertLevelRead \(ll_alert_level)")
        self.ll_alert_level = ll_alert_level
        let ll_al_int = Int8(bitPattern: ll_alert_level)
        switch ll_al_int {
        case 0:
            btn_low.backgroundColor = UIColor.green
            btn_medium.backgroundColor = default_btn_colour
            btn_high.backgroundColor = default_btn_colour
        case 1:
```

```
            btn_medium.backgroundColor = UIColor.yellow
            btn_low.backgroundColor = default_btn_colour
            btn_high.backgroundColor = default_btn_colour
        case 2:
            btn_high.backgroundColor = UIColor.red
            btn_medium.backgroundColor = default_btn_colour
            btn_low.backgroundColor = default_btn_colour
        default:
            btn_high.backgroundColor = default_btn_colour
            btn_medium.backgroundColor = default_btn_colour
            btn_low.backgroundColor = default_btn_colour
        }
    }
```

In BleAdapter add a function which will initiate reading of the Link Loss service's Alert Level characteristic and a delegate function for the result of all characteristic values received from the remote peripheral. Include conditional logic which will send the value read to the view controller if it comes from the alert level characteristic.

```
    func getLlAlertLevel() {
        selected_peripheral?.readValue(for: ll_alert_level_characteristic!)
    }
    func peripheral(_ peripheral: CBPeripheral,
                    didUpdateValueFor characteristic: CBCharacteristic,
                    error: Error?){
        print("didUpdateValueFor characteristic:
service=\(characteristic.service.uuid.uuidString)
characteristic=\(characteristic.uuid.uuidString)")
        if characteristic.service.uuid == CBUUID(string: LINK_LOSS_SERVICE_UUID) &&
characteristic.uuid == CBUUID(string: ALERT_LEVEL_CHARACTERISTIC) {
            if let data = characteristic.value {
                var values = [UInt8](repeating:0, count:data.count)
                data.copyBytes(to: &values, count: data.count)
                bluetooth_operations_consumer?.onLlAlertLevelRead(values[0])
            } else {
                print("ERROR: NO DATA from characteristic")
            }
        }
    }
```

Update onDiscoveryFinished to make a call to our new getLlAlertLevel function so it looks like this:

```
    func onDiscoveryFinished() {
        print("onDiscoveryFinished")
        if (got_ia_alert_level == false || got_ll_alert_level == false || got_pm_client_proximity
== false) {
            setUiStateInvalidBdsk()
        } else {
            setUiStateValidBdsk()
            adapter.getLlAlertLevel()
        }
    }
```

Checkpoint

Test the application. One of the three Low|Medium|High buttons should have its background colour set to Green|Yellow|Red respectively according to the current value of the Link Loss service's Alert Level characteristic. Unless you've changed this with some other Bluetooth tool, it should be zero and so the Low button will be highlighted.

Write to the link loss service alert level characteristic

Next we'll implement those three button so that a value of 0, 1 or 2 is written to the Link Loss service's Alert Level characteristic.

Add another function to the BluetoothOperationsConsumerprotocol in BleAdapter:

```
    func onLlAlertLevelWritten()
```

This function will inform the view controller when a write with response operation has completed.

Add the following function to BleAdapter to allow the view controller to update the link loss service's alert level:

```
    func setLlAlertLevel(alert_level: UInt8) {
        let new_alert_level = [alert_level]
        let alert_level_byte = Data(bytes: new_alert_level)
        selected_peripheral?.writeValue(alert_level_byte, for: ll_alert_level_characteristic!,
type: .withResponse)
    }
```

Also in BleAdapter, add the following delegate function to receive calls from the CBPeripheral object when write operations have finished:

```
    func peripheral(_ peripheral: CBPeripheral,
                    didWriteValueFor characteristic: CBCharacteristic,
```

```
                    error: Error?){
        print("didWriteValueFor characteristic: service=\(characteristic.service.uuid.uuidString)
characteristic=\(characteristic.uuid.uuidString)")
        if characteristic.service.uuid == CBUUID(string: LINK_LOSS_SERVICE_UUID) &&
characteristic.uuid == CBUUID(string: ALERT_LEVEL_CHARACTERISTIC) {
            bluetooth_operations_consumer?.onLlAlertLevelWritten()
        }
    }
```

In the view controller, add this variable:

```
var new_ll_alert_level: UInt8?
```

Update the alert level button click handlers as shown here:

```
    @IBAction func onLow(_ sender: UIButton) {
        print("onLow")
        self.new_ll_alert_level = UInt8(0)
        adapter.setLlAlertLevel(alert_level: new_ll_alert_level!)
    }

    @IBAction func onMedium(_ sender: UIButton) {
        print("onMedium")
        self.new_ll_alert_level = UInt8(1)
        adapter.setLlAlertLevel(alert_level: new_ll_alert_level!)
    }

    @IBAction func onHigh(_ sender: UIButton) {
        print("onHigh")
        self.new_ll_alert_level = UInt8(2)
        adapter.setLlAlertLevel(alert_level: new_ll_alert_level!)
    }
```

As you can see, we make a note of the new alert level and then invoke the write operation using the BleAdapter object.

Finally, add this function to the view controller:

```
    func onLlAlertLevelWritten() {
        print("onLlAlertLevelWritten \(new_ll_alert_level)")
        ll_alert_level = new_ll_alert_level
```

```
        let ll_al_int = Int8(bitPattern: new_ll_alert_level!)
        switch ll_al_int {
        case 0:
            btn_low.backgroundColor = UIColor.green
            btn_medium.backgroundColor = default_btn_colour
            btn_high.backgroundColor = default_btn_colour
        case 1:
            btn_medium.backgroundColor = UIColor.yellow
            btn_low.backgroundColor = default_btn_colour
            btn_high.backgroundColor = default_btn_colour
        case 2:
            btn_high.backgroundColor = UIColor.red
            btn_medium.backgroundColor = default_btn_colour
            btn_low.backgroundColor = default_btn_colour
        default:
            btn_high.backgroundColor = default_btn_colour
            btn_medium.backgroundColor = default_btn_colour
            btn_low.backgroundColor = default_btn_colour
        }
    }
```

It will be called when the write operation has completed and we can then use the new alert level value to highlight the appropriate button in the UI.

Update the onDisconnected function so that all three of the low|medium|high buttons are coloured grey when we disconnect.

```
        btn_low.backgroundColor = default_btn_colour
        btn_medium.backgroundColor = default_btn_colour
        btn_high.backgroundColor = default_btn_colour
```

Checkpoint

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

1. Scan, select and connect, 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 device you should see the green LED flash by way of acknowledgement.
2. 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 device is “LOW” (0).
3. 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 device you should see the yellow LED flash by way of acknowledgement.

4. Go back and disconnect. Select and connect again.
- 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.
- 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.
- The HIGH button should be highlighted red to indicate the current alert level read back from the peripheral device is “HIGH” (2).

17. Immediate Alert

Pressing the MAKE A NOISE button needs to trigger writing the current selected value for alert level to the Immediate Alert service's alert level characteristic. So that's the next job to do.

Add the following function to BleAdapter:

```
    func setIaAlertLevel(alert_level: UInt8) {
        print("setIaAlertLevel arg=\(alert_level)")
        var new_alert_level = [UInt8]()
        new_alert_level.append(alert_level)
        let alert_level_byte = Data(bytes: new_alert_level)
        selected_peripheral?.writeValue(alert_level_byte, for: ia_alert_level_characteristic!,
type: .withoutResponse)
    }
```

Note that this time we're using Write Without Response. as such there's no call back from CBPeripheral to implement as a delegate method. All we have to do now is change the view controller to call our new function when the MAKE A NOISE button is pressed.

```
    @IBAction func onMakeNoise(_ sender: UIButton) {
        print("onMakeNoise")
        // yes this should be the ll_alert_level. We use the same values for link loss and
immediate alert.
        adapter.setIaAlertLevel(alert_level: ll_alert_level!)
    }
```

18. Checkpoint

Test the MAKE A NOISE button. Select each of Low, Medium and High before pressing it to perform 3 tests in total. The BDSK device should flash all LEDS, 3, 4 or 5 times depending on the selected alert level. At medium and high levels it will also sound the buzzer.

19. 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.

Add the following Bool to the view controller:

```
var sharing: Bool?
```

Initialise it to false in the viewDidLoad function:

```
sharing = false
```

Add the following function to BleAdapter:

```
    func setClientProximity(proximity_band: UInt8 , rssi: UInt8) {
        let cp_bytes = [proximity_band , rssi]
        let cp_data = Data(bytes: cp_bytes)
        selected_peripheral?.writeValue(cp_data, for: client_proximity_characteristic!, type:
.withoutResponse)
    }
```

Back in the view controller, complete the onShareChanged function so that it toggles our new sharing variable and uses the new BleAdapter function to write the special value combination of proximity_band=0 and rssi=0 to inform the BDSK device we've switched sharing off, whenever that is the case:

```
    @IBAction func onShareChanged(_ sender: UISwitch) {
        print("onShareChanged \(sender.isOn)")
        sharing = sender.isOn
        if (!sharing!) {
            // tell BDSK device we're no longer sharing
            adapter.setClientProximity(proximity_band: UInt8(0), rssi: UInt8(0))
        }
    }
```

Add code to the updateProximityClassification function so that if sharing is enabled, we call the new adapter function to write the current proximity_band and RSSI values to the Client Proximity characteristic:

```
    func updateProximityClassification(_ rssi_value: NSNumber) {
        var proximity_band = 3;
        rssi.text = "RSSI: \(rssi_value.floatValue.description) dBm"
        if (rssi_value.floatValue < -80.0) {
            proximity_classification.backgroundColor = UIColor.red
        } else if (rssi_value.floatValue < -50.0) {
            proximity_classification.backgroundColor = UIColor.yellow
            proximity_band = 2;
        } else {
            proximity_classification.backgroundColor = UIColor.green
            proximity_band = 1;
        }

        if (sharing!) {
            adapter.setClientProximity(proximity_band: UInt8(proximity_band), rssi:
UInt8(rssi_value))
        }
    }
```

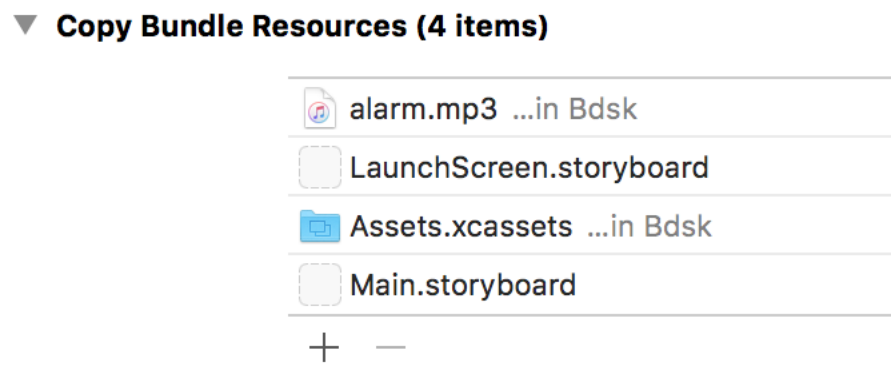
20. Checkpoint

That should be it for this feature! Test now and check that enabling sharing causes RSSI values to be displayed on the BDSK LCD display and that the proximity_band value causes the corresponding LED to be lit on the circuit board. Check that when you switch sharing off, this is reflected in both the LEDs on the circuit board and on the LCD display.

21. Link Loss

Our final task is to respond to the Bluetooth connection being lost by playing an audible sound whenever this occurs with the selected alert level at a value > 0

The project resources include a sound file called alarm.mp3. Add it to your project using Project navigator, Build Phases, Copy Bundle Resources, +, Add Another, Finish.



Create a new Swift file called AlarmManager and import AVFoundation. Add stub implementations of the three functions we're going to need, as shown:

```
import AVFoundation

class AlarmManager  : NSObject, AVAudioPlayerDelegate {

    var player: AVAudioPlayer?

    static let sharedInstance = AlarmManager()

    func soundAlarm() {

    }

    func alarmIsSounding()->Bool {
        return false
    }

    func stopAlarm() {

    }
}
```

Add an instance of AlarmManager to our device view controller:

```
    var alarm: AlarmManager!
```

and obtain an instance of the class in viewDidLoad:

```
alarm = AlarmManager.sharedInstance
```

Now complete the soundAlarm() function so that it plays the alarm.mp3 file when invoked:

```
    func soundAlarm() {
        print("soundAlarm")
        let url = Bundle.main.url(forResource: "alarm", withExtension: "mp3")!
        do {
            try AVAudioSession.sharedInstance().setCategory(AVAudioSessionCategoryPlayback)
            try AVAudioSession.sharedInstance().setActive(true)
            player = try AVAudioPlayer(contentsOf: url)
```

```
            if let plyr = player {
                plyr.delegate = self
                plyr.prepareToPlay()
                plyr.play()
            }
        } catch let error {
            print("soundAlarm \(error.localizedDescription)")
        }
    }
```

In the view controller, update the onDisconnected function so that it has the following, final lines of code:

```
        if (Int8(ll_alert_level!) > 0) {
            alarm.soundAlarm()
        }
```

If we're playing an alarm when the user reconnects, we want to stop the alarm. To achieve this, update the onConnected function in our view controller so that it uses AlarmManager to check whether the alarm is currently sounding and if it is, stop it.

```
    func onConnected() {
        print("onConnected")
        if (alarm.alarmIsSounding()) {
            alarm.stopAlarm()
        }
        btn_connect.setTitle("DISCONNECT", for: .normal)
        adapter.startPollingRssi(self)
        adapter.discoverServices()
    }
```

Obviously we haven't yet implemented the last 2 functions in AlarmManager so let's do that now:

```
    func alarmIsSounding()->Bool {
        if (player == nil) {
            return false
        }
        if let plyr = player {
            return plyr.isPlaying
        } else {
            return false
        }
```

```
    }

    func stopAlarm() {
        if let plyr = player {
            plyr.stop()
        }
    }
```

22. Checkpoint

Test your application and ensure the features implemented so far are working as expected.

23. 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 a Swift API point of view, notifications and indications are dealt with in exactly the same way. There's no need to acknowledge indications explicitly. iOS will do that for you.

Add the following variables to DeviceViewController.swift

```
    var monitoring_temperature: Bool?
    var got_ht_temperature_measurement: Bool?
```

Initialise the new variables on viewDidLoad:

```
    override func viewDidLoad() {
        super.viewDidLoad()
        adapter = BLEAdapter.sharedInstance
        utils = Utils.sharedInstance
        alarm = AlarmManager.sharedInstance
        device_details.text = "Device: BDSK [" +
adapter.selected_peripheral!.identifier.uuidString + "]"
        btn_low.isEnabled = false
        btn_medium.isEnabled = false
        btn_high.isEnabled = false
        btn_noise.isEnabled = false
        switch_share.isEnabled = false
        switch_temperature.isEnabled = false
        got_ia_alert_level = false
        got_ll_alert_level = false
```

```
        got_pm_client_proximity = false
        got_ht_temperature_measurement = false
        sharing = false
        monitoring_temperature = false
        ll_alert_level = 0
    }
```

Add another function to the BluetoothOperationsConsumerprotocol in BleAdapter:

```
    func onTemperatureMeasurement(_ temperature_measurement: [UInt8])
```

This function will inform the view controller when a temperature measurement characteristic has been received and pass its value across.

Now add the following function to BleAdapter. It will allow the device view controller to enable to disable temperature measurement indications:

```
    func setTemperatureMonitoringIndications(state: Bool) {
        selected_peripheral?.setNotifyValue(state, for: temperature_measurement_characteristic!)
    }
```

Our existing didUpdateValueFor delegate function will receive temperature measurement indications, so we need to make some changes to accommodate this as shown next:

```
    func peripheral(_ peripheral: CBPeripheral,
                    didUpdateValueFor characteristic: CBCharacteristic,
                    error: Error?){
        print("didUpdateValueFor characteristic:
service=\(characteristic.service.uuid.uuidString)
characteristic=\(characteristic.uuid.uuidString)")
        if characteristic.service.uuid == CBUUID(string: LINK_LOSS_SERVICE_UUID) &&
characteristic.uuid == CBUUID(string: ALERT_LEVEL_CHARACTERISTIC) {
            if let data = characteristic.value {
                var values = [UInt8](repeating:0, count:data.count)
                data.copyBytes(to: &values, count: data.count)
                bluetooth_operations_consumer?.onLlAlertLevelRead(values[0])
            } else {
                print("ERROR: NO DATA from characteristic")
            }
            return
        }
        if characteristic.service.uuid == CBUUID(string: HEALTH_THERMOMETER_SERVICE_UUID) &&
characteristic.uuid == CBUUID(string: TEMPERATURE_MEASUREMENT_CHARACTERISTIC) {
```

```
            if let data = characteristic.value {
                var values = [UInt8](repeating:0, count:data.count)
                data.copyBytes(to: &values, count: data.count)
                bluetooth_operations_consumer?.onTemperatureMeasurement(values)
            } else {
                print("ERROR: NO DATA from characteristic")
            }
        }
    }
```

As you can see, on receiving a temperature measurement indication, we pass the received data as an array of unsigned bytes to the device view controller via the BluetoothOperationsConsumer protocol which it implements.

Next, let's complete the action associated with the UI temperature monitoring switch so that we can control subscribing or unsubscribing from indications:

```
    @IBAction func onTemperatureMonitoringChanged(_ sender: UISwitch) {
        print("onTemperatureMonitoringChanged \(sender.isOn)")
        adapter.setTemperatureMonitoringIndications(state: sender.isOn)
        if (!sender.isOn) {
            temperature.text = "not available"
        }
    }
```

All that's left to do now is to implement the onTemperatureMeasurement function which we just added to the BluetoothOperationConsumer protocol so that received indication values can be passed to our view controller and displayed.

```
    func onTemperatureMeasurement(_ temperature_measurement: [UInt8]) {


        if (temperature_measurement.count != 5) {
            print("ERROR: onTemperatureMeasurement expected 5 bytes but
received\(temperature_measurement.count)" )
            return
        }
        // first octet specifies a unit of Celsius (0) or Fahrenheit (1)
        var unit = "C"
        if (temperature_measurement[0] == 1) {
            unit = "F"
        }
```

```
        // 4 bytes required for Int32 conversion. Also, make big endian while we're at it.
        var mantissa_bytes: [UInt8] = [0x00, 0x00, 0x00 , 0x00]
        mantissa_bytes[1] = temperature_measurement[3]
        mantissa_bytes[2] = temperature_measurement[2]
        mantissa_bytes[3] = temperature_measurement[1]

        // propogate 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
        }
        let data = Data(bytes: mantissa_bytes)
        let mantissa_value = Int32(bigEndian: data.withUnsafeBytes { $0.pointee })
        print("mantissa_value=\(mantissa_value)")

        let exponent = temperature_measurement[4]
        print("exponent=\(exponent)")

        let signed_exponent = Int8(bitPattern: UInt8(exponent))
        print("signed exponent=\(signed_exponent)")

        let p = pow(Double(10), Double(signed_exponent))

        let new_temperature = Double(mantissa_value) * p
        print("new_temperature=\(new_temperature)")

        temperature.text = "\(new_temperature)"
    }
```

We expect to receive an array of 5 unsigned octets from BleAdapter, so the first thing we do is to check this. The 5 octets consist of:

Octet 0: indicates the temperature units with 0=Celsius and 1=Fahrenheit

Octets 1-3: A signed mantissa value

Octet 4: A signed exponent

The temperature value itself is therefore, a floating point value consisting of the mantissa and exponent. You can see how we derive the temperature from these two values, taking care with our conversion of the three byte mantissa into a signed 32-bit Swift Int32 variable.

Test your Application

You're now ready to take your Swift 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.

Troubleshooting

1. Why do I see 'api misuse cbperipheral can only accept commands while in the connected state' in the Scode console?

You have to wait until the [-CBCentralManagerDelegate centralManagerDidUpdateState:] callback has been called, then verify that the state is PoweredOn, before you start scanning for peripherals.

2. Why can't I drag / drop IBOutlets from my storyboard to my Swift class?

Make sure the thing to which you want to add outlets (e.g. Table View Cell) has a custom class and that this is specified in the class attribute in the Custom Class section of the Identity Inspector.

3. Why do I see a blank, black screen when I test my app?

You probably have no initial entry point set in your storyboard.

Select the initial view controller in the storyboard. In the Attributes Inspector select the Is Initial View Controller checkbox.

4. Why do I get a SIGABRT from AppDelegate.swift when I run my application and how can I fix it?

This can be caused by various fatal error conditions.

If this happens immediately on launching the app and leaves you looking at a blank screen, then there's a chance there's a non-existent outlet referenced from the initial view controller.

The Connections Inspector in the right hand navigation pane gives you a way to check what outlets and actions are defined

Incomplete implementation of delegates can also cause this. Here's an example:

```
2016-12-07 08:56:27.843228 BdskTest2[333:19467] *** Terminating app due to uncaught exception
'NSInternalInconsistencyException', reason: '<CBCentralManager: 0x170075b00> has provided a
```

```
restore identifier but the delegate doesn't implement the centralManager:willRestoreState:
method'
```

So the problem here was that a callback was failing because the willRestoreState method of the CBCentralManagerDelegate class has not been implemented.

The problem was fixed by adding this to the BleAdapter class:

```
func centralManager(_ central: CBCentralManager, willRestoreState dict: [String : Any]) {
    self.peripherals = dict[CBCentralManagerRestoredStatePeripheralsKey] as! NSMutableArray;
}
```

5. UIViewController compilation error 'has no initializers'

This can occur if you have a member variable which has no default value such as

```
var peripheral: CBPeripheral
```

This is because all Swift variables must have a value or be marked optional. The error message is confusing but is saying that if you do not specify a value or mark as optional, the class must have an initialiser in which to set the default value.

Fix by marking optional like this:

```
var peripheral: CBPeripheral?
```