Bluetooth Developer Starter Kit

Creating a Bluetooth low energy application for iOS

Version: 3.1.0

Last updated: 2nd December 2016

Contents

Revision History

Version	Date	Author	Changes
1.0	1st May 2013	Matchbox	Initial version
2.0	25th November 2014	Vincent Gao and 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	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	2nd December 2016	Martin Woolley, Bluetooth SIG	Name change and version number increment to sync with changes to the Android lab for V3.1.

Overview

This document is a step-by-step guide to creating a Bluetooth Smart Ready app on iOS, using a Bluetooth Smart device implementing a GATT profile. For details on the specific profile and how to configure your Bluetooth Smart Device, please refer to the *Smart Starter Kit - Getting Started With Arduino* document downloaded with this package. In this lab we will **not** cover any basics of iOS programming, nor is the resulting code suitable for production – the lab, tasks and code are designed to provide instruction in the principles and practice of Bluetooth Smart.

NOTE: This lab is not a complete commercial solution. All instructions relate directly to the kit list as published.

Objectives

This lab provides instructions to achieve the following:

- Discover a nearby Bluetooth Smart Device.
- Communicate with a Bluetooth Smart device.
- Implement a "key finder" app on iOS, using a Bluetooth Smart device.

System Requirements

- Apple OSX 10 and up
- Xcode 5 and up
- Apple Developer License

Equipment Requirements

- Arduino Uno
- A-to-B type USB cable
- RedBearLab Bluetooth Low Energy Shield
- iPhone 4S and up with iOS 8.

Exercises

This hands-on lab contains the following exercises:

1. Scanning for Bluetooth Smart devices
2. Communicating with a Bluetooth Smart device

Estimated time to complete this lab: 45 to 60 minutes

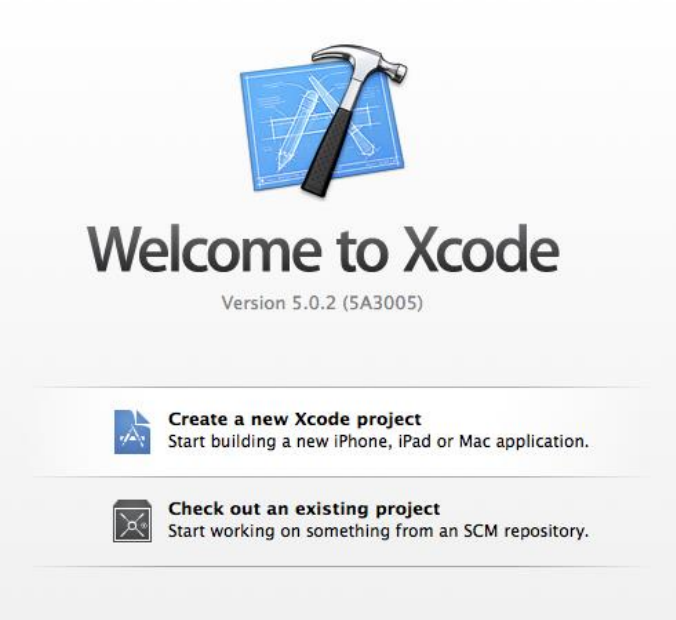
Exercise 1: Getting Started

This lab is all about scanning, connecting and ccomunicating with a Bluetooth Smart device. We will begin with a new Xcode project and import some ready-made wrapper files that will help us interface with a Bluetooth Smart device.

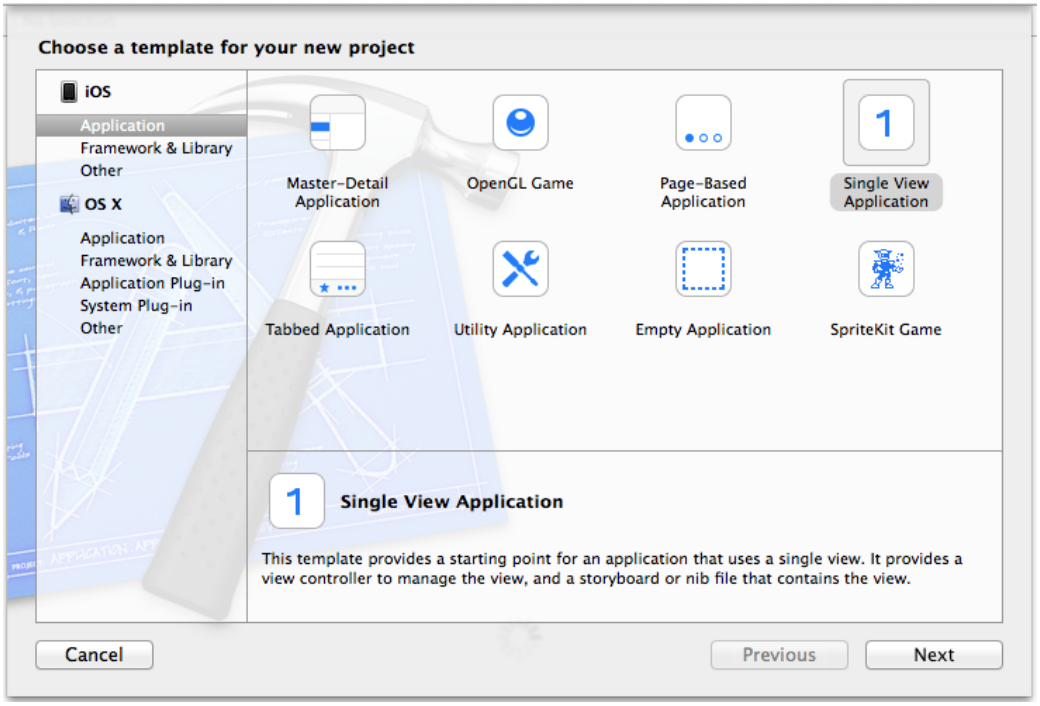
Note: to complete this lab you will first need to configure the Arduino Uno and Bluetooth Smart Shield. Please refer to the *Smart Starter Kit - Getting Started With Arduino* document downloaded with this package.

Task 1 – Create a new Xcode Project

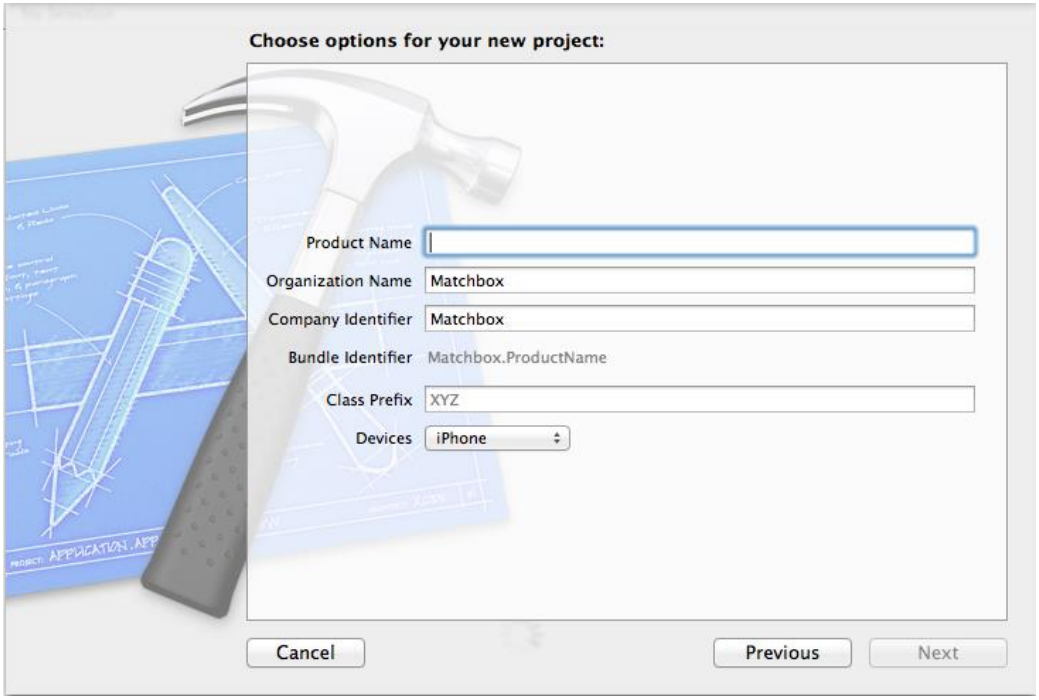
1. Download the latest version of Xcode from the App Store. At the time of writing, this was Xcode 6.1.
2. Run Xcode.
3. On the splash screen, select “Create a new Xcode project.”



4. Select “Single View Application” from the template selection screen



5. Give the project a suitable product name, fill in your company details and select *iPhone* as the target device.



6. Set the project location and create the project

Task 2 – Importing Wrapper Files

Now that we have a basic empty project, we will import the Bluetooth Smart wrapper files that we mentioned at the start of task 1.

Overview of wrapper files

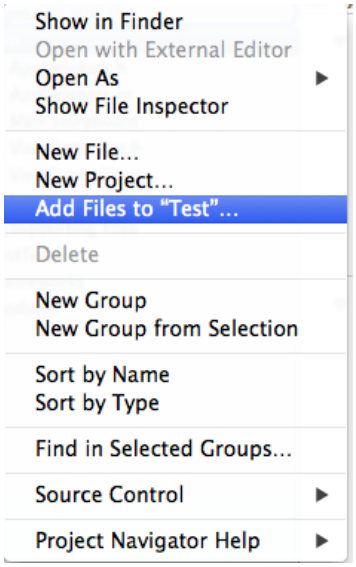
For the purposes of the tutorial, we have supplied wrapper files that define and implement a class called BLEAdapter. BLEAdapter will give you easier access to the basic Bluetooth Smart functions of iOS.

We will reference these functions throughout the tutorial and we will explain them fully as we go. The BLEAdapter wrapper class is for instructional purposes – it is not something you should need in your own live projects. Eventually you will gain enough familiarity with the underlying iOS APIs that you will no longer need the wrapper class.

> For more details on BLEAdapter and how to use it, you can download the *Application Accelerator for Apple* from Bluetooth.org here[1].

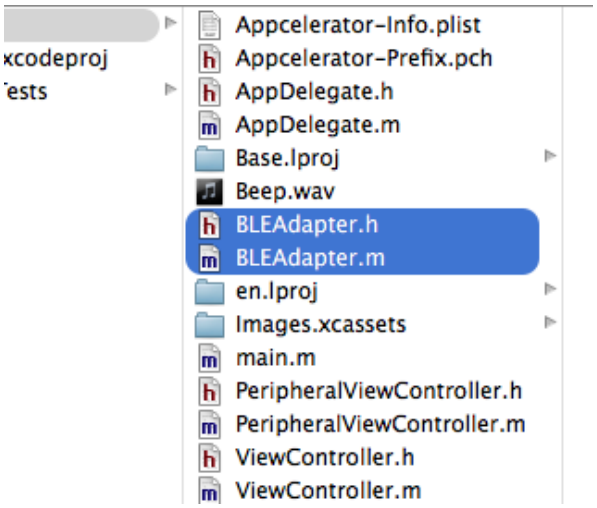
Importing the BLEAdapter wrapper files

1. In the project navigator, select the destination project.
2. Choose File > Add Files to "<Project_Name>".

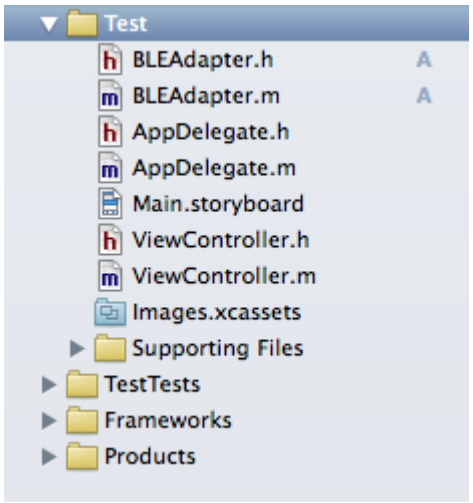


3. Browse to the *[Lab Install Folder] iOS/Source/* folder.
4. Select *BLEAdapter.m* and *BLEAdapter.h* from this folder.

[1] https://developer.bluetooth.org/Pages/Bluetooth-Apple-Developers.aspx



5. Leave the import options as they are and click Add.
6. Once imported, the files will appear below the top-level named folder, if not select them and drag them in, so they match the image below



7. Now that you have the BLEAdapter files, you can start working with a Bluetooth Smart device. We will cover this in the next task.

Task 3 – Scanning for peripherals

For this task, we will need an iPhone 4S or above, your Mac OS development machine, and a Bluetooth Smart peripheral powered up and in range.

> Note – you can type the code you see below into Xcode, or you can simply copy and paste from this document. Alternatively you can reference the completed project and follow along.

Set up the first page of the application, creating a button to allow us to scan for available Bluetooth Smart Peripherals

1. In Xcode, open up the project containing the BLEAdapter wrapper files.
2. Updating the .h file, add the following statements to your viewcontroller.h

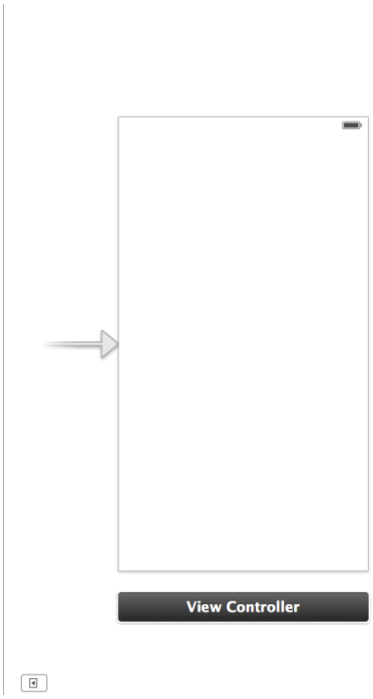
Objective C

```
#import <UIKit/UIKit.h>
#import "BLEAdapter.h"

@interface ViewController : UIViewController<UITableViewDelegate,
UITableViewDataSource>

@property (strong, nonatomic) BLEAdapter* bleWrapper;

@end
```

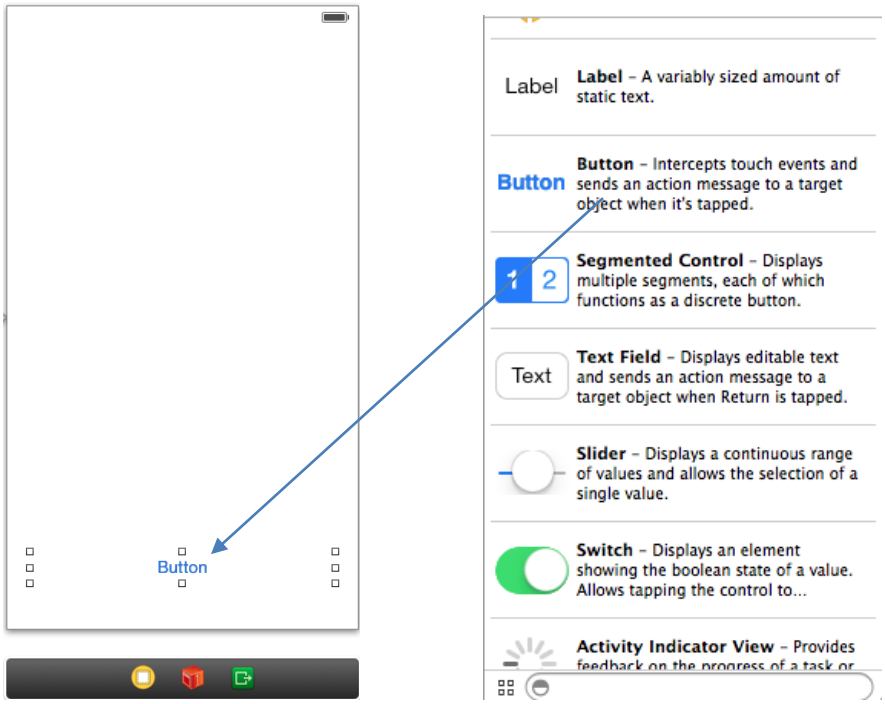
We now have a property, that will allow us to use functionality from the class.

We have also specified that this class is deriving from UITableViewDelegate, UITableViewDataSource so we are able to override some table functionality, which we will be using later.

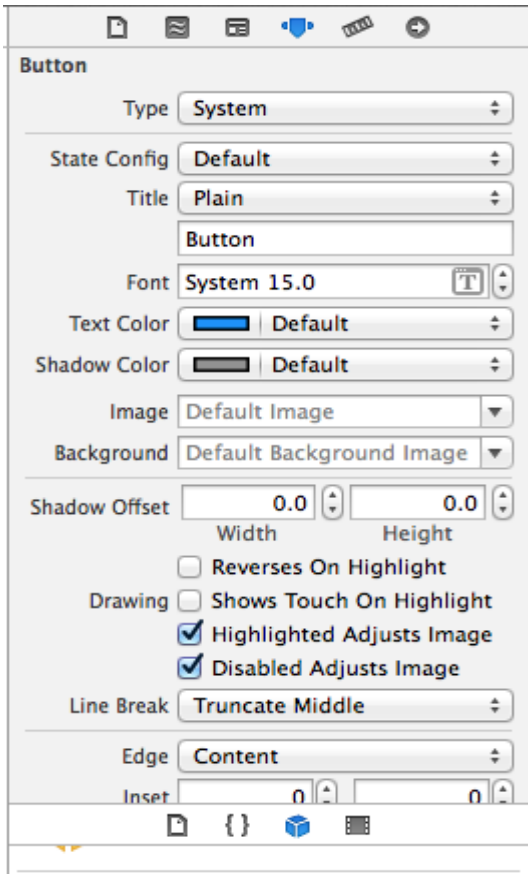
3. **Adding a button**. Click on the main storyboard filename in the left-hand pane. You should be presented with a blank workspace.



4. On the bottom right of the screen, you should see some template controls that can be added. Drag a button in from the bottom right pane, placing it in the middle of the blank storyboard at the bottom.



5. Above the template items dock we can see attributes for our selected items. Click on the Attributes Inspector, marked in blue at the top of the dock shown below.

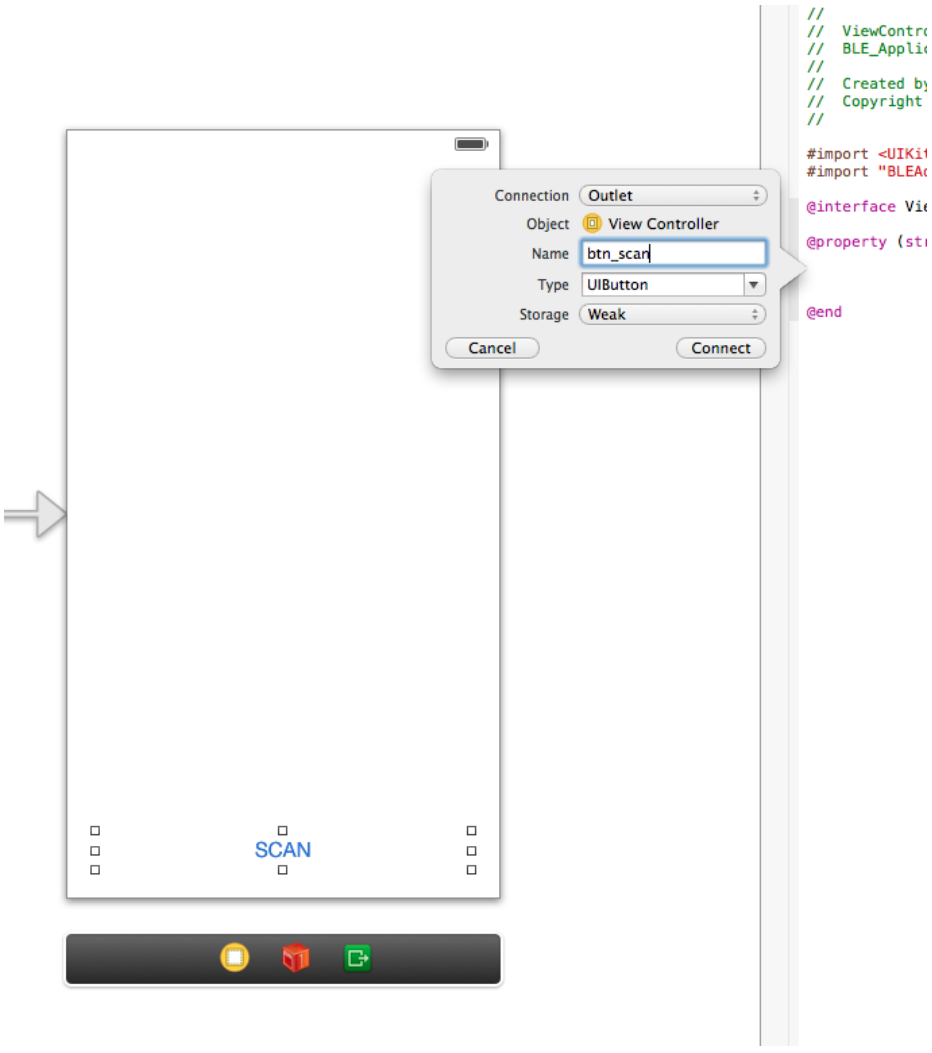


6. Give the button an easily recognizable title, e.g. “SCAN”
7. Select Assistants Editor, represented by a small tux and bowtie in the toolbar

8. Ensure you are viewing "viewcontroller.h"

Connecting UI to Events

9. We now need to connect the button to the View Controller, for which we’ll need to create an outlet. To do this, select the button on the storyboard, hold ctrl and drag it to the .h file. This will allow you to access the button from your code.

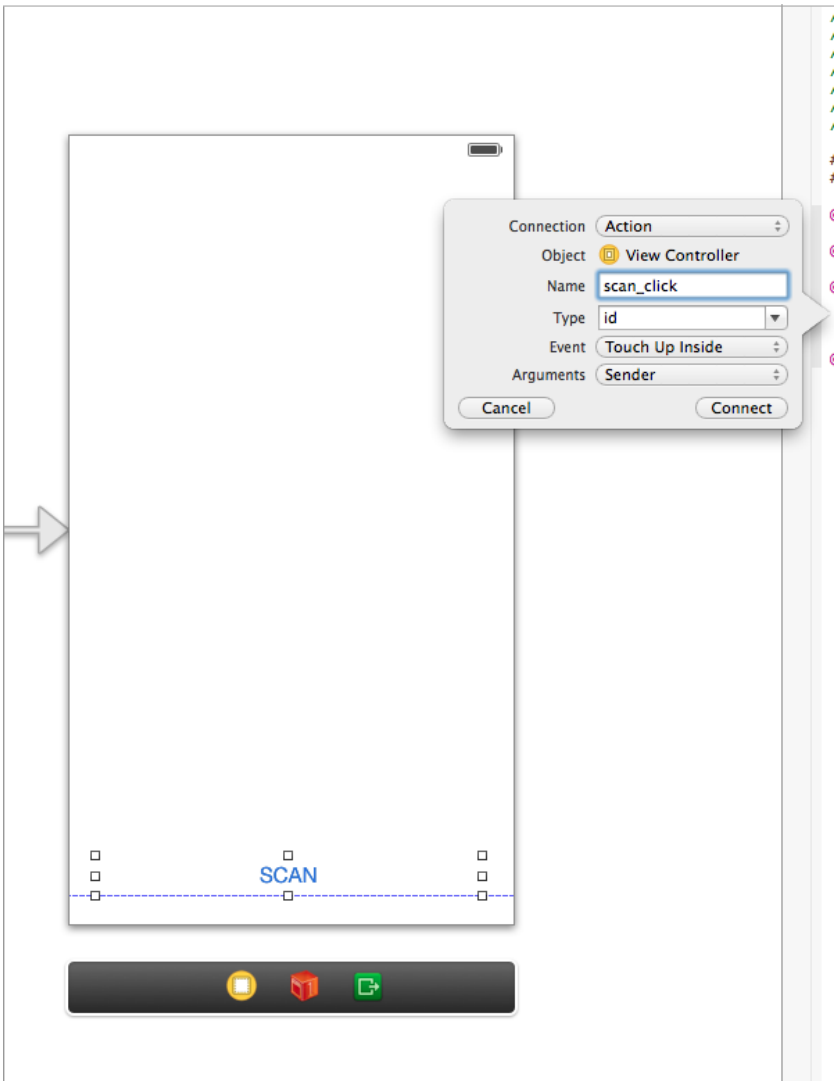


10. Next, we need to create an action. This is similar to creating an outlet.

    Select the button on the storyboard, hold the Control key and drag the button to your .h file.

    This time, select “Action” from the “Connection” drop down.

    In the “Name” field, enter something recognizable, e.g. “scan_click”.



    Finally, Click “connect” and Xcode will generate the following code in your .m file:

Objective C

```
(IBAction)scan_click:(id)sender
{
}
```

11. We now have a button, but in order to be able to scan for devices we need to add some code to handle a button press. We will step through the below code block to explain what is happening.

Objective C

```
#import "ViewController.h"

@interface ViewController ()
@end

@implementation ViewController

@synthesize bleWrapper;
NSTimer* scannerTimer;
static NSString * const kServiceUUID = @"1803";

- (void)viewDidLoad
{
    [super viewDidLoad];
      // Do any additional setup after loading the view, typically from a nib.

    self.bleWrapper = [BLEAdapter sharedBLEAdapter];
    [self.bleWrapper controlSetup:1];
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

- (IBAction)scan_click:(id)sender
{
    // Stop button from being able to be clicked again untilscan is finished
    [self.btn_scan setEnabled:false];

    // call the scan function in the BLEAdapter, specifiying a 2 second scan time
    // and the service UUID we require.
    [self.bleWrapper findBLEPeripherals:2 serviceUUID:KServiceID ];

    // set progress spinner showing we are loading something
    [UIApplication sharedApplication].networkActivityIndicatorVisible = YES;

    // Initiate timer to go off in 2 seconds
    [NSTimer scheduledTimerWithTimeInterval:(float)2.0 target:self
selector:@selector(connectionTimer:) userInfo:nil repeats:NO];
}

// Called when scan period is over
-(void) connectionTimer:(NSTimer *)timer
{
    // stop progress spinner
    [UIApplication sharedApplication].networkActivityIndicatorVisible = NO;

    NSString *message;

    // Display alert to show if we have found devices
    if ( self.bleWrapper.peripherals.count > 0)
```

```
    {
        message = @"We have found devices";
    }
    else
    {
       message = @"We have not found devices";
    }
    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Result" message:message delegate:nil
cancelButtonTitle:@"Ok" otherButtonTitles: nil];
    [alert show];

    // enable scan button
    [self.btn_scan setEnabled:true];
}
@end
```

In the code above, the first thing we do is synthesize the bleWrapper and add a variable for a timer.

```
@synthesize bleWrapper;
NSTimer* scannerTimer;
```

In the *viewDidLoad* function we initialize the *BLEAdapter* class

```
self.bleWrapper = [BLEAdapter sharedBLEAdapter];
 [self.bleWrapper controlSetup:1];
```

Next, we add some code to the *scan_click* function, which will disable the scan button after it has been pressed. This is to prevent multiple scans from occurring simultaneously.

```
[self.btn_scan setEnabled:false];
```

We then make a call to the *BLEAdapter* to find peripherals; we also pass through an integer value that represents the number of seconds we want the scan to run.

It is good practice to keep these scans as short as possible, as they have a direct effect on battery drain.

```
[self.bleWrapper findBLEPeripherals:2];
```

At this point, we'll show the network activity indicator as useful UI feedback.

Finally, we are setting up a timer to go off at the same time we have told the BLEAdapter to stop scanning.

This means we can then query the adapter class and see if any peripherals were found.

```
[NSTimer scheduledTimerWithTimeInterval:(float)2.0 target:self
selector:@selector(connectionTimer:) userInfo:nil repeats:NO];
```

Whenever the BLEAdapter finds a peripheral that fits the Bluetooth Smart device description, it will add the peripheral object to the collection called CBPeripherals. We'll use this object to display our found peripherals and their information.

The timer is set to call the *connectionTimer* function after 2 seconds. This function turns off the network indicator, as we are no longer processing background data, and then checks how many peripherals are in the collection within the *BLEAdapter*. **It may be that 2 seconds is not long enough to successfully find a device. If you are having trouble connecting try increasing this value to 20 seconds.**

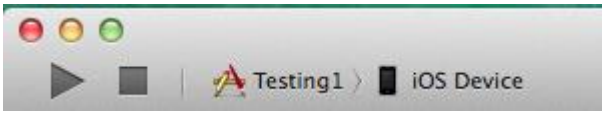
Objective C

```
NSString *message;

    // Display alert to show if we have found devices
    if ( self.bleWrapper.peripherals.count > 0)
    {
        message = @"We have found devices";
    }
    else
    {
       message = @"We have not found devices";
    }
    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Result" message:message delegate:nil
cancelButtonTitle:@"Ok" otherButtonTitles: nil];
    [alert show];
```

In the code block above, we initialize a string variable. This will hold the messages we want to display; if the collection size is greater than 0, we have found something, so we display “We have found devices”.

If this is not the case we display “We have not found devices”.

At this point you should be able to run your project on your iOS device and you should see a blank screen, except for the button. Tapping it will bring up the small network spinner animation and, finally, a message box.



When running the application, make sure you are running on the device, and not using the emulator. The above image shows the option which needs to be selected in order to run on device.

To successfully scan and find a device the Arduino board needs to have had the proximity code uploaded and to be running. To perform this step please refer to the ‘Getting started with Arduino’ document.

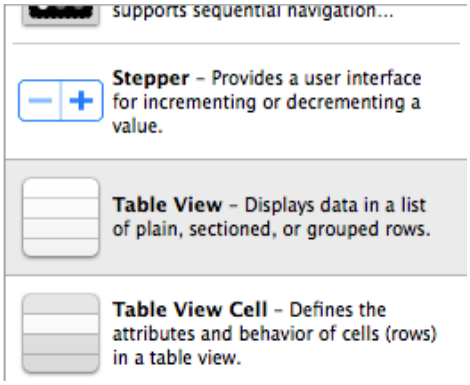
If scanning for an Arduino device fails to show a nearby device, try uploading the sketch again and resetting the board (the board can be reset by connecting a wire between pin 7 and GND then pressing the button). Also, make sure that no other device is connected, and is thus preventing you from connecting. If you are still having problems try increasing the scan timeout value from 2 to 20 seconds.

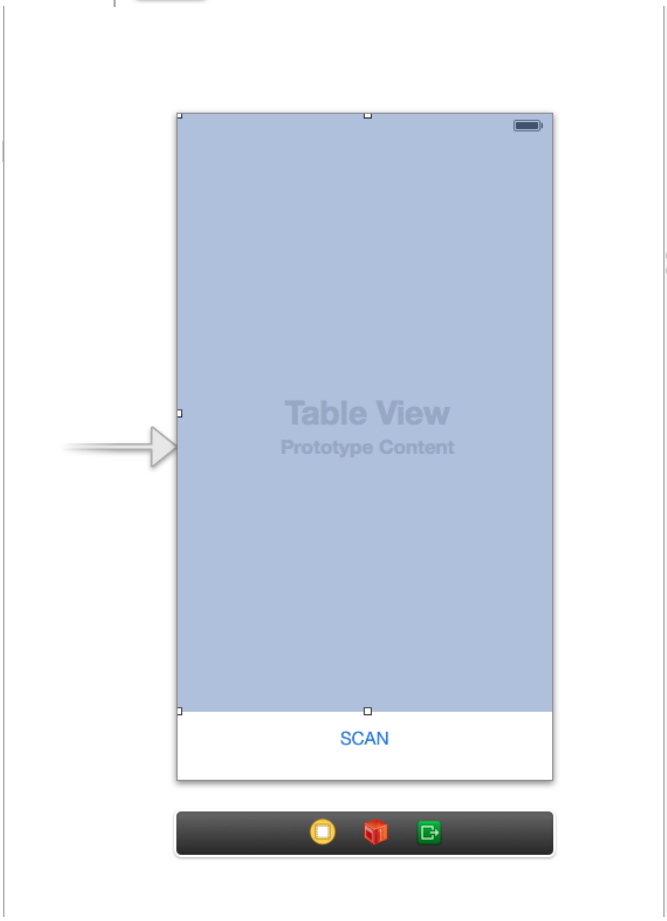
12. At this point, hopefully, we have found some devices. Now we need to see the details of what we have found: the services and characteristics for

    To do this we are first going to add a table and some table functionality.

    We need to add a UITable to our view, so clicking *Main.storyboard* will open interface builder.

As we did with the button, we can drag a Table View from the templates dock and add it to our main UI.





Once this has been placed onto the page, we must create an outlet for it.

As before, Control-Click on the Table View and drag it into our .h file.

Select “Outlet” from the “Connection” drop down.

Give the table a recognizable name in the “Name” field e.g. “tbl_data”

Your .h file should now look something like this:

Objective C

```
#import <UIKit/UIKit.h>
#import "BLEAdapter.h"

@interface ViewController : UIViewController<UITableViewDelegate, UITableViewDataSource>

@property (strong, nonatomic) BLEAdapter* bleWrapper;

@property (weak, nonatomic) IBOutlet UIButton *btn_scan;
- (IBAction)scan_click:(id)sender;

@property (weak, nonatomic) IBOutlet UITableView *tbl_data;

@end
```

Back in the .m file we need to add a few table functions and delegate functions. These are:

Objective C

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath
*)indexPath

- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath
```

The first of these tells the UI how many items we want to display in our table

The second tells the UI how we want each cell in our table to look

The last function is the logic we want to run when we click a cell.

We know that BLEAdapter has a collection of devices so if we get the count value of that collection then we can complete the first function like so.

Objective C

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
    return [self.bleWrapper.peripherals count];
}
```

Next, we want a default cell template and we want to display the name of the devices we find. We can do this by adding the following:

Objective C

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath
*)indexPath
{
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];

    if(cell == nil)
    {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
reuseIdentifier:CellIdentifier];
    }

    CBPeripheral * temp = (CBPeripheral *)[self.bleWrapper.peripherals objectAtIndex:indexPath.row];
    cell.textLabel.text = temp.name;

    return cell;
}
```

The indexPath value iterates over the collection, each item in the collection is selected, the name is retrieved and used to set the *cell.textLabel.text* value.

We can then add a UIAlert that will pop up when we select a table cell into the method we just defined:

Objective C

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    CBPeripheral *temp =(CBPeripheral *)[ self.bleWrapper.peripherals objectAtIndex:indexPath.row];
    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Result" message:temp.name delegate:nil
cancelButtonTitle:@"Ok" otherButtonTitles: nil];
    [alert show];

}
```

Finally we need to setup the table's delegate functionality so the data can be accessed by the table on the view. We do this by adding the following to the viewDidLoad function into ViewController.m

Objective C

```
    [self.tbl_data setDelegate:self];

    [self.tbl_data setDataSource: self];
```

This means we can now update the table data when we have scanned for devices by editing the the connectionTimer function in the VIewController.m file to be the same as the following, adding in the [self.tbl_data reloadData]; line.

Objective C

```
// Called when scan period is over
-(void) connectionTimer:(NSTimer *)timer
{
    // stop progress spinner
    [UIApplication sharedApplication].networkActivityIndicatorVisible = NO;
    NSString *message;
    // Display alert to show if we have found devices
    if ( self.bleWrapper.peripherals.count > 0)
    {
        [self.tbl_data reloadData];
    }
    else
    {
       message = @"We have not found devices";
        UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Result" message:message delegate:nil
cancelButtonTitle:@"Ok" otherButtonTitles: nil];
        [alert show];
    }
    // enable scan button
    [self.btn_scan setEnabled:true];
}
```

Summary of Exercise 1

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

1. Created a blank iOS project
2. Imported the BLEWrapper files to help us make some Bluetooth Smart calls
3. Added the ability to scan for, and list, nearby Bluetooth Smart devices

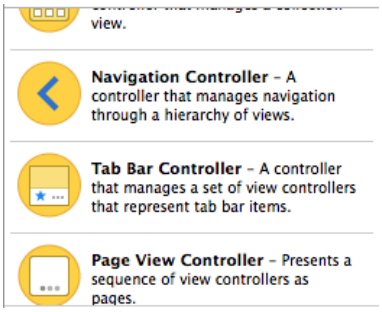
To proceed to Exercise 2, continue below with your existing project, or you may open the XCode project under the *[Lab Install Folder]/iOS/Source/ex2* folder. The XCode project under Begin is a snapshot of the work up to this point, i.e. with Exercise 1 completed and ready to start on Exercise 2.

Exercise 2: Communicating with the Bluetooth Smart device

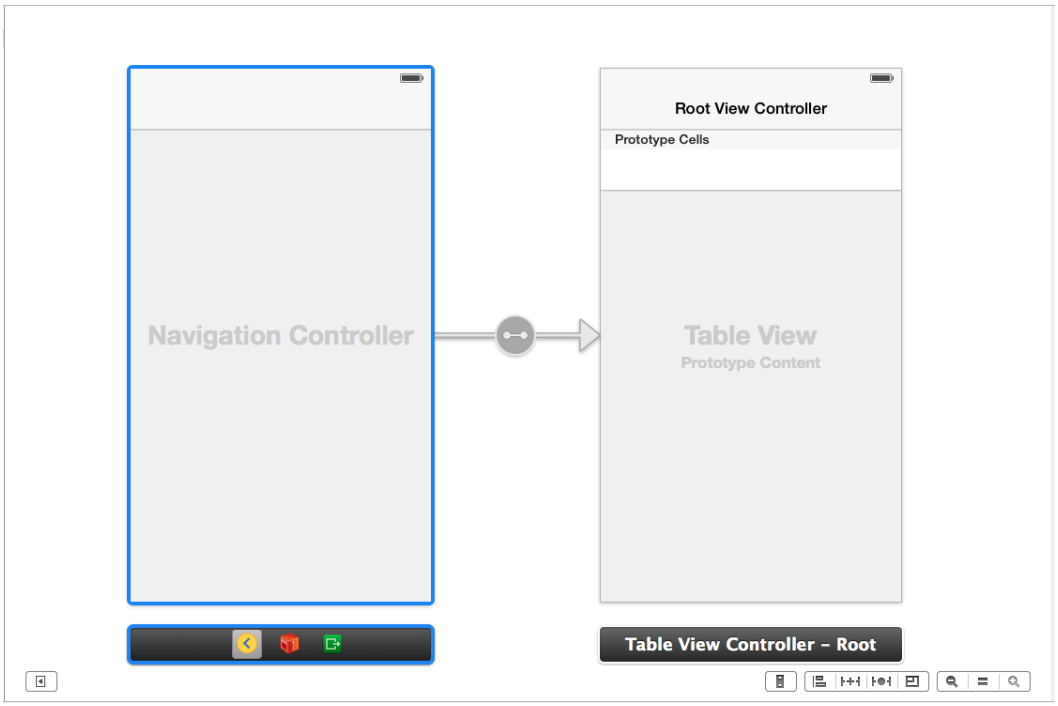
Task 1 – Connecting to the device

In this exercise we will be adding a new page to the application to handle connecting to and interaction with the device. We first need to add a Navigation Controller to handle the navigation between the main page and the interaction page.

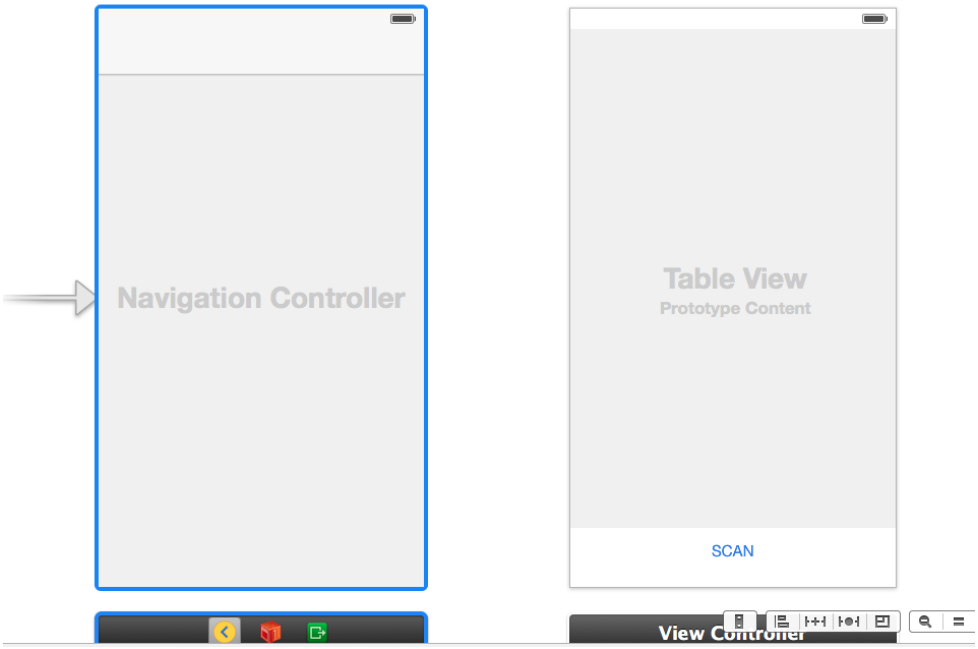
1. First, select Main.storyboard and from the right hand side drag a Navigation Controller to the page.



Once this lands on the page, you will see a representation of two new views on the screen.



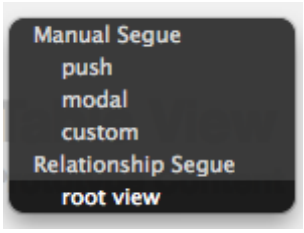
2. We only need the Navigation Controller, because we already have a main page, so here we delete the current Root View Controller.



3. Next, we move the Navigation Controller next to the current scan page.
4. Now we need to tell the application that we want to open the Navigation Controller first when the application opens. We do this by dragging this arrow:

so it is in front of the Navigation Controller. If you were to run the application now you would see that the application would start and open the Navigation Controller, but the application would never get to our scan page. Therefore, we need to create a Segue between the Navigation Controller and the scan page.

5. To create a segue, click the Navigation Controller hold Ctrl and drag to the scan page. This opens a dialog asking which type of relationship we would like.

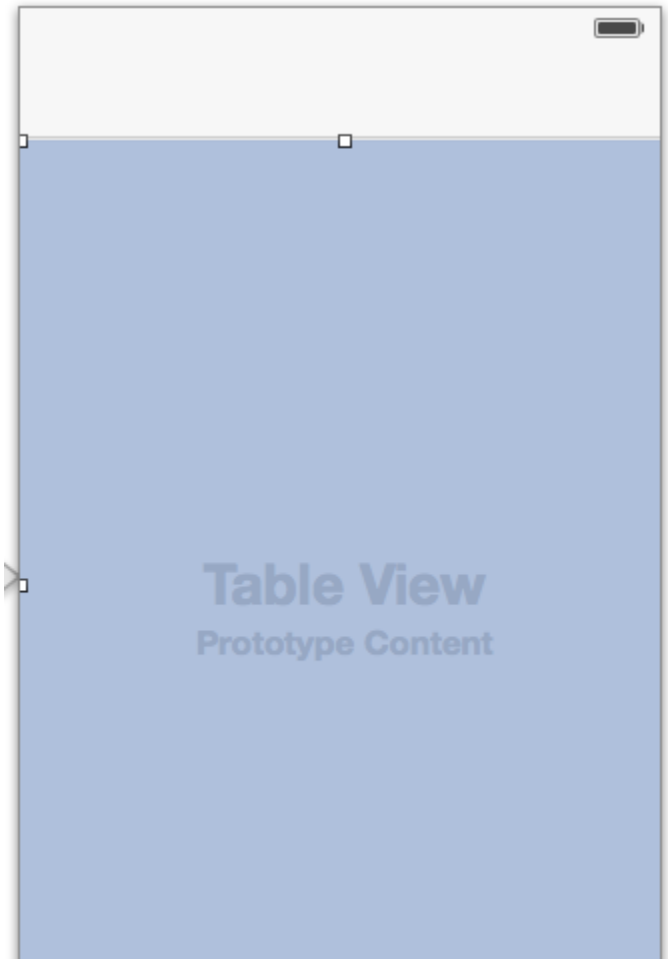


In this case we should select that we want the scan page to be the root view.

Status check: if you run the application you should see it initialize and navigate to the scan page.

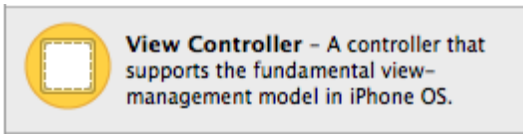
After you have added the Navigation Controller you may notice the visual representation in interface builder has added a Navigation Controller bar to the scan page. This is great, however we need to move any controls which are behind it, otherwise they will not be visible.

6. Resize the table to start at the bottom of the navigation bar like so:

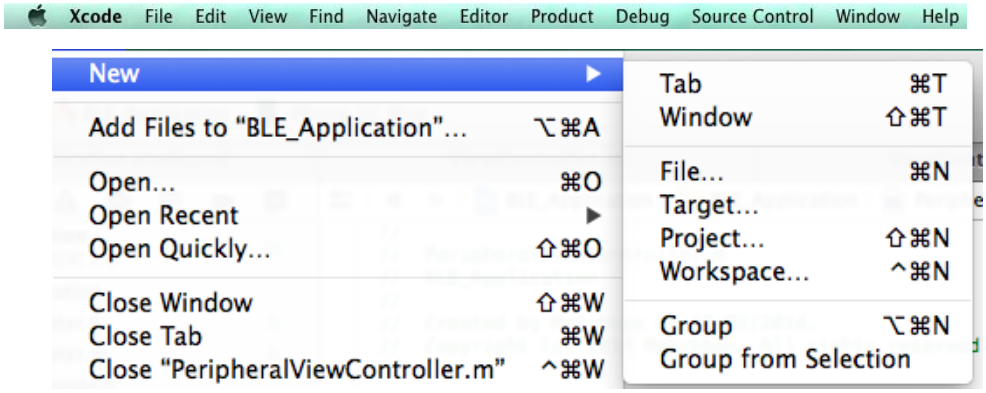


The Navigation Controller is now in place.

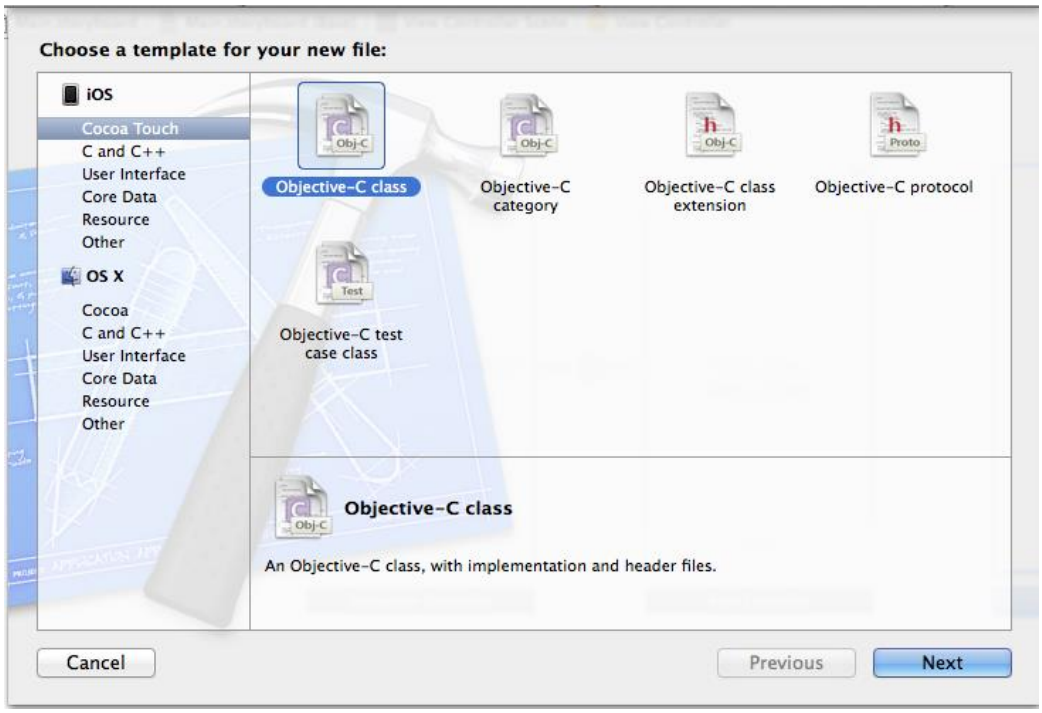
7. Now we need to add a new View Controller to handle device interaction and data. To do this, drag another View Controller from the bottom right menu.



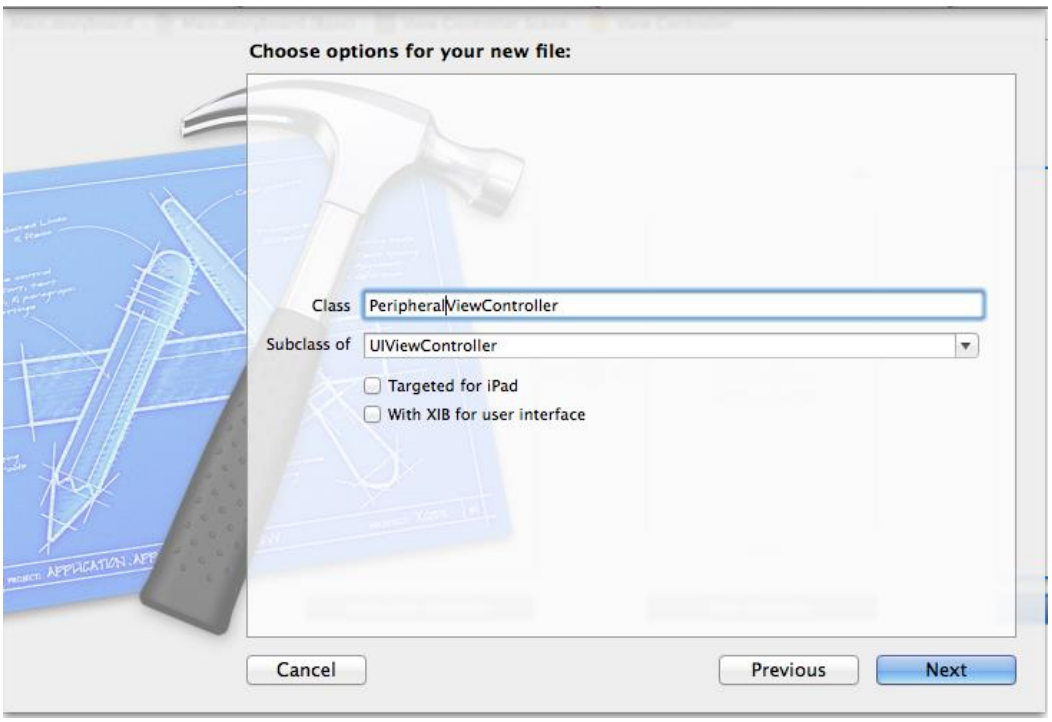
This gives us a new view on the scan page. Next we need to create a View Controller class to connect to the view. To do this we add a new file by navigating to File > New > File:



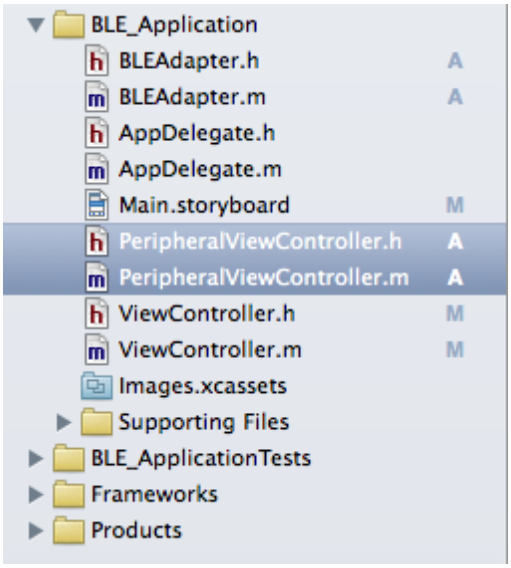
Choose the Objective-C class template:



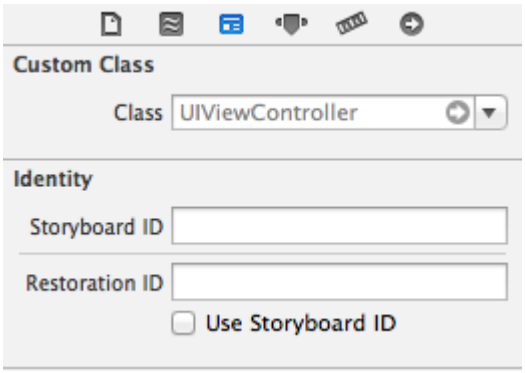
Hit Next, and give the class a name, making sure it is a subclass of UIViewController. We have called it PeripheralViewController.



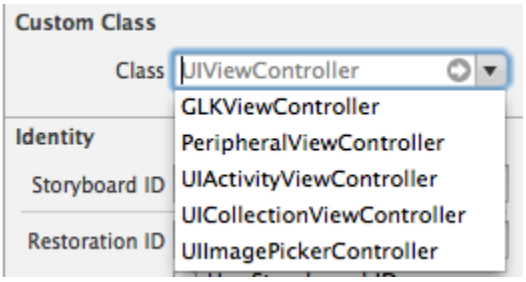
Leave the other default options, click Next and then Create, and you should see the files in your project solution.



8. We now need to connect the view to the View Controller. We can do this by opening Main.storyboard and clicking on the new view. Using the identity inspector, we can change the class connected to the view.

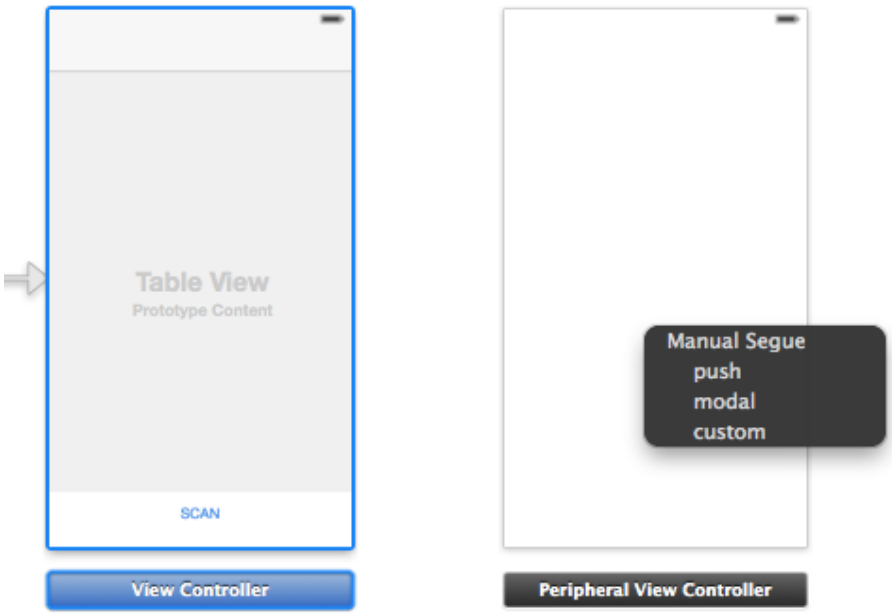


Click the class drop-down box to see a choice of available View Controllers. We want to select the one we just created (in our case, PeripheralViewController).



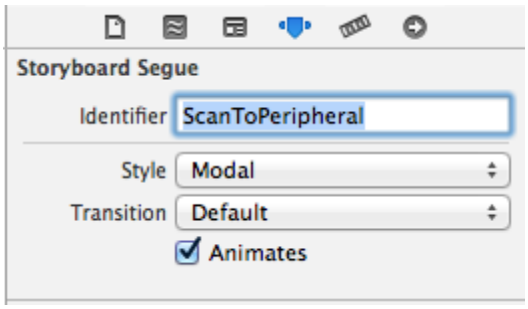
We now have a new page and a new controller.

9. Next, we need to create a Segue between the scan page and the new page. We do this the same way we create an output or action: by selecting the scan page, holding down the Ctrl button and dragging to the new page. This will open a Segue options page, where we want to select “Modal”.

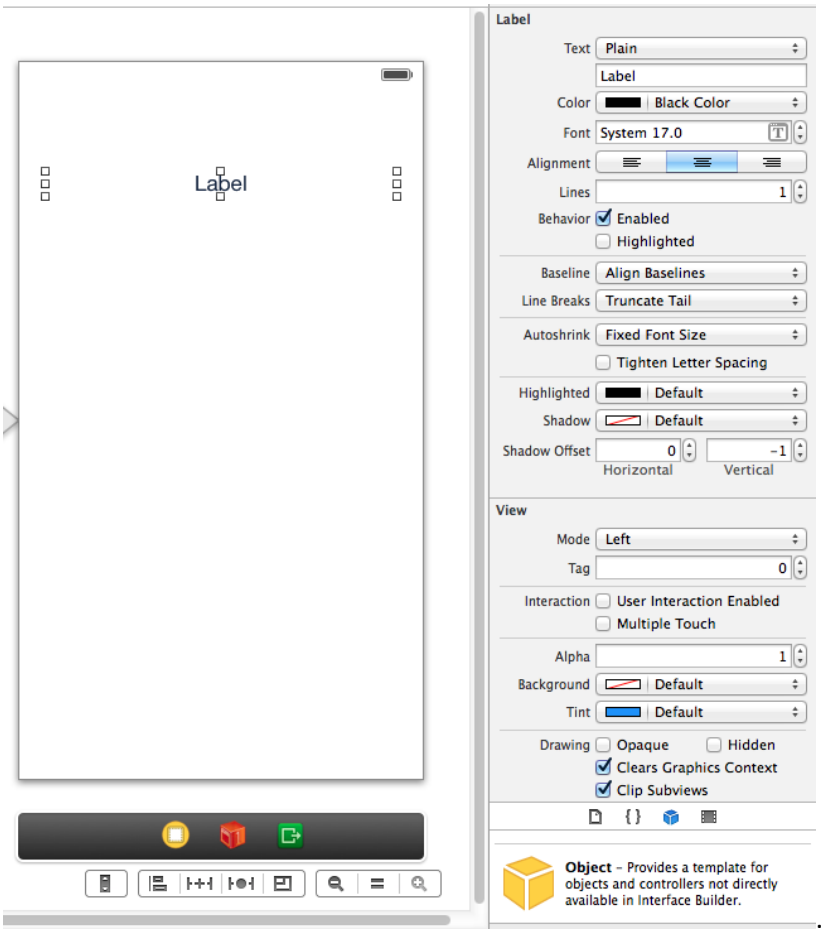


Note: When trying to create the Segue make sure you are zoomed out using the scale buttons at the bottom right of the screen.

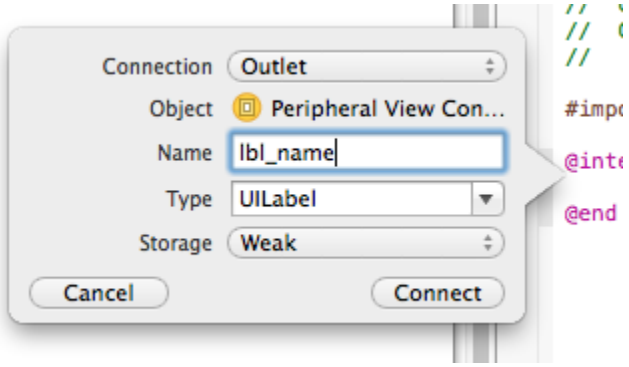
We also want to name our new Segue something memorable, like ScanToPeripheral.



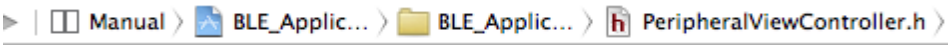
10. Next we need a label to display the name of the connected device. Drag out a new label object and place it in the middle of the screen.



11. Once you have placed the label, create an outlet by clicking the control, holding Ctrl and dragging to the .h file



Note: when creating an outlet, make sure you connect to the correct .h file. In this case the new viewcontroller we just created.



This sets up the UI we need for now.

12. Next, we will add code to pass through the device information from the scan page to the peripheral page, adding some connection logic.

Open your PeripheralViewController .h file and replace with the following:

Objective C

```
#import <UIKit/UIKit.h>
#import "BLEAdapter.h"

@interface PeripheralViewController : UIViewController

@property (weak, nonatomic) IBOutlet UILabel *lbl_name;

- (void) setData: (BLEAdapter *) adapter : (CBPeripheral *) peripheral;

@property (strong, nonatomic) BLEAdapter *bleAdapter;
@property (strong, nonatomic) CBPeripheral *toConnect;

@end
```

Here we are again importing the BLEAdapter class, which encapsulates all of the BLE APIs in a single class. We are also defining a setData function, which we will use to send the reference of the device we want to connect to from the scan page. Finally, we declare two variables to hold the local references of the BLEAdapter class, and the peripheral we want to connect to.

Add the following code to the PeripheralviewController.m file:

Objective C

```
@synthesize toConnect;
@synthesize bleAdapter;

-(void) setData: (BLEAdapter *) adapter : (CBPeripheral *) peripheral
{
    self.bleAdapter = adapter;
    self.toConnect = peripheral;
}
```

Here we are synthesizing the two variables, and implementing the function declared in the .h file. This function will be called when we are performing the Segue on the scan page to pass data from one View Controller to another. In our case this is going to be the peripheral and the BLEAdapter.

Finally we can change the viewDidLoad function to set the label text to the selected peripheral device name.

13. Now we need to write the code to perform the Segue and pass this data through. Go back to your ViewController.h file and add the following property:

Objective C

```
@property (strong, nonatomic) CBPeripheral* selected;
```

This is going to be used to hold a reference the peripheral we select from the table.

Now open the ViewController.m file and import the BLEAdapter.h and PeripheralViewController .h like so

Objective C

```
#import "BLEAdapter.h"
#import "PeripheralViewController.h"
```

And synthesize the new variable

Objective C

```
@synthesize selected;
```

We also need to change the didSelectRowAtIndexPath function to utilize this new variable

Objective C

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    CBPeripheral *temp =(CBPeripheral *)[self.bleWrapper.peripherals
objectAtIndex:indexPath.row];
```

```
    self.selected = temp;
    [self performSegueWithIdentifier:@"ScanToPeripheral" sender:self];
}
```

Here we save a local reference to the selected to peripheral and the manually call the Segue to navigate to the new page. Just before the actual navigation happens another function is called.

Objective C

```
-(void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    if([segue.identifier isEqualToString:@"ScanToPeripheral"])
    {
        PeripheralViewController *destViewController = segue.destinationViewController;
        [destViewController setData: self.bleWrapper : self.selected];
    }
}
```

Just before we navigate to the new page we get the destination View Controller (in this case peripheralViewController) and call the set data function on it, passing through our reference to the BLEAdapter and the selected peripheral.

Running your app now you should be able to scan, get a list of results, select a result and navigate to a new page and see the selected device name.

Connecting to device

14. Now we can connect to the device!

We have the reference to the device in the peripheral page, so we will continue to add the connect logic. First thing to do would be to add a button to the view, set its title to “Connect”, create an outlet with the name btn_connect and finally create an action called connect_click.

Your PeripheralViewController .h file should now look like this

Objective C

```
#import <UIKit/UIKit.h>
#import "BLEAdapter.h"

@interface PeripheralViewController : UIViewController

@property (weak, nonatomic) IBOutlet UILabel *lbl_name;

- (void) setData: (BLEAdapter *) adapter : (CBPeripheral *) peripheral;

@property (strong, nonatomic) BLEAdapter *bleAdapter;
@property (strong, nonatomic) CBPeripheral *toConnect;

@property (weak, nonatomic) IBOutlet UIButton *btn_connect;
- (IBAction)connect_click:(id)sender;

@end
```

15. You should also amend your PeripheralViewController.m file as follows:

Objective C

```
#import "PeripheralViewController.h"

@interface PeripheralViewController ()

@end

@implementation PeripheralViewController

@synthesize toConnect;
@synthesize bleAdapter;

// Set data from a Segue from outside of this View Controller
-(void) setData: (BLEAdapter *) adapter : (CBPeripheral *) peripheral
{
    self.bleAdapter = adapter;
    self.toConnect = peripheral;
}

- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil
{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
    if (self) {
        // Custom initialization
    }
    return self;
}

- (void)viewDidLoad
{
    [super viewDidLoad];
            // Do any additional setup after loading the view.

    // Set device name to label tet
    self.lbl_name.text = self.toConnect.name;
    [self.bleAdapter setDelegate:self];
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

- (IBAction)connect_click:(id)sender
{
    // If the CBPeripheral is n ot null try to connect to it
    if (self.toConnect != Nil)
    {
        [self.bleAdapter connectPeripheral:self.toConnect status:TRUE];
        [self.btn_connect setEnabled:false];
    }
```

```
}

// Call back function, which will fire when we know if we are connected or not
-(void) OnConnected:(BOOL)status
{
    if (status)
    {
        [self.btn_connect setEnabled: false];
        [self displayMessage:@"We are connected!"];
    }
    else
    {
        [self connectionFailed];
    }
    [self.btn_connect setEnabled:true];
}

// If we do not connect display error
-(void)connectionFailed
{
    [self displayMessage:@"Could not connect to device"];
    [self.navigationController popToRootViewControllerAnimated:TRUE];
}

// Helper method
-(void)displayMessage:(NSString*)message
{
    UIAlertView *alert = [[UIAlertView alloc] initWithTitle: @"DEBUG" message: message delegate:
nil cancelButtonTitle:@"OK" otherButtonTitles:nil];
    [alert show];
}

@end
```

The main changes to this file are the addition of this line of code in the *viewDidLoad* function:

```
[self.bleAdapter setDelegate:self];
```

This sets the delegate target of the BLEAdapter to the peripheralViewController. In other words, when the BLEAdapter has to fire a callback method it will fire only in the peripheralViewController. This means we can override certain functionality from the BLEAdapter. You will see this when we try to connect.

We also implemented the connect_click event

Objective C

```
- (IBAction)connect_click:(id)sender
{
    // If the CBPeripheral is not null try to connect to it
    if (self.toConnect != Nil)
    {
        [self.bleAdapter connectPeripheral:self.toConnect status:TRUE];
        [self.btn_connect setEnabled:false];
```

```
    }
}
```

First we make sure the peripheral we want to connect to is valid, i.e. not null. If this is true we can call the connectPeripheral function on the BLEAdapter. This function will try and connect to the device. If the connection succeeds, the function didConnectPeripheral in the BLEAdapter will be called. If, however, the connection fails, didFailToConnectPeripheral in the BLEadapter will be called. Both of these functions call the delegate method onConnected, passing through the connection state as a Boolean value. This onConnected function fires the function with the same name in PeripheralViewController.

Objective C

```
-(void) OnConnected:(BOOL)status
{
    if (status)
    {
        [self.btn_connect setEnabled: false];
        [self displayMessage:@"We are connected!"];
    }
    else
    {
        [self connectionFailed];
    }
   }
```

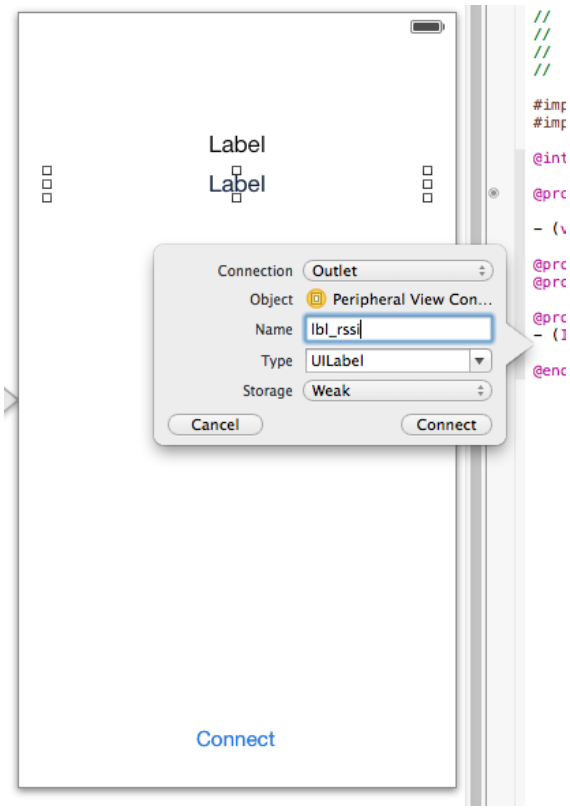
If the status is true we know we are connected. If not, we let the user know we are not connected.

If you run your application now you should be able to connect to the BLE board from your phone.

Task 2 – Reading RSSI

Now we are connected to the device, we will read and display the RSSI value (signal strength).

1. First we want to add another label on the UI. Using interface builder, create an outlet and give it the name “lbl_rssi”.



2. Back in the PeripheralViewController.m file we are going to set up a timer and every time it fires we will read the RSSI value of the connection

   First you should define a new NSTimer, so under where you synthesize your variables you should add

Objective C

```
NSTimer* rssiTimer;
```

3. We want to hide the label when we are not connected. In the viewDidLoad function, add:

Objective C

```
[self.lbl_rssi setHidden:true];
```

4. To actually initialize the timer, we need to update the onConnected function to look like this:

Objective C

```
-(void) OnConnected:(BOOL)status
{
    if (status)
    {
        [self.btn_connect setEnabled: false];
```

```
    [self.lbl_rssi setHidden:false];
     rssiTimer = [NSTimer scheduledTimerWithTimeInterval:(float)3 target:self
selector:@selector(readRSSITimer:)      userInfo:nil repeats:YES];

    }
    else
    {
        [self connectionFailed];
    }
    [self.btn_connect setEnabled:true];
}
```

The main difference here is that instead of creating a UIAlert and letting the user know we are connected, we are starting a timer to fire a function every 3 seconds. This function which will be called is:

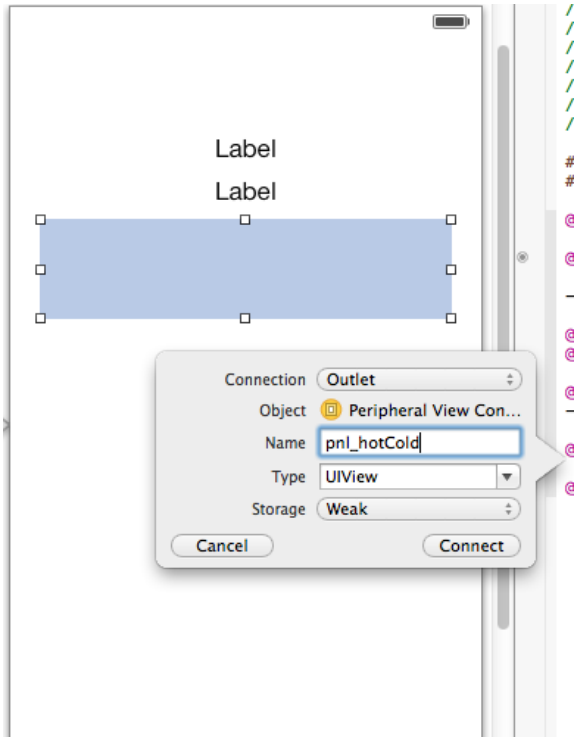
Objective C

```
-(void) readRSSITimer:(NSTimer *)timer
{
    [self.bleAdapter.activePeripheral readRSSI];
    if (self.bleAdapter.activePeripheral.RSSI != NULL)
    {
        self.lbl_rssi.text = [NSString stringWithFormat: @"RSSI : %@",
self.bleAdapter.activePeripheral.RSSI];
    }
}
```

The first thing we do is call readRSSI on the connected peripheral. This will update the RSSI property, which we can then access by calling the BLEAdapter activePeripheral RSSI. We then change the string value to reflect the changes.

Now that we can see the RSSI value updating, we can create a “hot and cold” panel. This will be a simple way to tell the user how close they are to the Bluetooth Smart device.

5. To create this hot/cold panel, we need to add a new UIView to your peripheral page. Drag up a UIView and create an outlet called pnl_hotCold.



With the .h file updated, we can add the code to change the color of the panel depending on the RSSI value.

6. Now we will edit the realRSSITimer function so that it reads as follows:

Objective C

```
-(void) readRSSITimer:(NSTimer *)timer
{
    NSLog(@"Reading RSSI value");

    [self.bleAdapter.activePeripheral readRSSI];

    if (self.bleAdapter.activePeripheral.RSSI != NULL)
    {
        self.lbl_rssi.text = [NSString stringWithFormat: @"RSSI : %@",
self.bleAdapter.activePeripheral.RSSI];

        Byte rssi = [self.bleAdapter.activePeripheral.RSSI integerValue] * -1;
        Byte proximity_band;

        if (rssi <= 50)
        {
            proximity_band = 0;
        }
        else if (rssi > 50 && rssi < 80)
        {
            proximity_band = 1;
        }
        else
        {
            proximity_band = 2;
        }
        [self ChangeRSSIColour : proximity_band];
```

```
        NSLog(@"PROXIMITY BAND : %d  RSSI : %d", proximity_band , rssi);

    } else {
        NSLog(@"RSSI not available");
    }

}
```

This inverts the value of the RSSI to be positive rather than negative, and there are a range of if statements which determine the alert. We can then pass this new alert level to a color change function like this:

Objective C

```
-(void) ChangeRSSIColour: (int) level
{
    switch (level)
    {
        case 1:
            [self.pnl_hotCold setBackgroundColor:[UIColor orangeColor]];
            break;
        case 2:
            [self.pnl_hotCold setBackgroundColor:[UIColor redColor]];
            break;
        default:
            [self.pnl_hotCold setBackgroundColor:[UIColor greenColor]];
            break;
    }
    [self.pnl_hotCold setNeedsDisplay];
}
```

Adding the above code to your PeripheralviewController.m file gets the current alert level and changes the background of the new UIView we placed on the page.

Task Complete – Run, Scan, Connect, Read RSSI.

Build, deploy and run the app now. You should be able to scan for devices, connect to a device, read the RSSI value and have a basic hot and cold monitor.

In the next task we will scan the Bluetooth Smart device for available services.

Task 3 – Scanning for services

We scan for services on a Bluetooth Smart device by calling getAllServicesFromPeripheral in our BLEAdapter wrapper, passing in a reference to the currently connected peripheral.

1. Go to PeripheralViewController.m and edit the OnConnected function inside the `if (status)` block, adding this line:

Objective C

```
[bleAdapter getAllServicesFromPeripheral:self.bleAdapter.activePeripheral];
```

getAllServicesFromPeripheral will scan the device for all available services and return to a function called *OnDiscoverServices*.

2. We need to override this function by copying the following code into PeripheralViewController.m:

Objective C

```
-(void) OnDiscoverServices:(NSArray *)s
{
    for (CBService *service in s)
    {
        NSLog(@"SERVICE : %s", [self.bleAdapter CBUUIDToString: service.UUID]);
    }
}
```

This will iterate through all of the services available and print out the UUID (unique id) to the console window.

Objective: We now want to filter the list of returned services to look for two specific services.

Navigating to the Bluetooth.org website will give you a list of predefined services and their corresponding UUID.

https://developer.bluetooth.org/gatt/services/Pages/ServicesHome.aspx

In this tutorial we are looking for the Link loss service with UUID 1803 and the Immediate Alert service with UUID 1802 as well as a custom service called the Proximity Monitoring service which has 128 bit UUID 3E099910-293F-11E4-93BD-AFD0FE6D1DFD. We will adapt our onDiscoverServices function to find and save a reference to any service which has one of these UUIDs.

3. Add three new string variables to hold the UUID values of the services we are looking for by placing these in the PeripheralViewController.m file.

Objective C

```
NSString *IMMEDIATE_ALERT_SERVICE_UUID = @"1802";
NSString *LINK_LOSS_SERVICE_UUID = @"1803";
NSString  *PROXIMITY_MONITORING_SERVICE_UUID =  @"3E099910-293F-11E4-93BD-
AFD0FE6D1DFD";
```

We also need somewhere to put the services when we have found them, so we can also add these variables to the PeripheralViewController.m file.

Objective C

```
CBService *LinkLoss;
CBService *ImmediateAlert;
CBService  *ProximityMonitoring;
```

4. Next we need to change the onDiscoverServices to check each service it comes across against these UUID values and if it is a match save a reference to it.

Objective C

```
-(void) OnDiscoverServices:(NSArray *)s
{
    for (CBService *service in s)
    {
        NSLog(@"SERVICE : %s", [self.bleAdapter CBUUIDToString: service.UUID]);

        if ([service.UUID isEqual:[CBUUID UUIDWithString: LINK_LOSS_SERVICE_UUID]])
        {
            NSLog(@"Got Link Loss Service");
            LinkLoss = service;
        }
        if ([service.UUID isEqual:[CBUUID UUIDWithString: IMMEDIATE_ALERT_SERVICE_UUID]])
        {
            NSLog(@"Got Immediate Alert Service");
            ImmediateAlert = service;
        }
        if ([service.UUID isEqual:[CBUUID UUIDWithString: PROXIMITY_MONITORING_SERVICE_UUID]])
        {
            NSLog(@"Got Proximity Monitoring Service");
            ProximityMonitoring = service;
        }
    }
}
```

Task Complete – Enumerating services

We should now have a local reference to all of the services we need: one to handle a button click, one to set the link loss alert level and one to allow sharing of proximity data with the Arduino. However, although we have the services, we still need to get the actual characteristics to read / write.

Task 4 – Querying characteristics

In this task, we will query the Bluetooth Smart device for specific characteristics that are part of its services.

We're interested in the Alert Level characteristics belonging to each of the Link Loss and Immediate Alert services and the Client Proximity characteristic which belongs to the custom Proximity Monitoring service.

NOTE: A list of predefined GATT characteristics can be found on Bluetooth.org[2]

Looking up the UUID value of the Alert Level characteristic on Bluetooth.org, we can see it has a value of **2A06**. Don't forget though that both the Link Loss service and the Immediate Alert service have instances of this characteristic type, and we need access to both of them. The custom Client Proximity characteristic has a 128 bit UUID whose value was defined in the Arduino lab and which we'll use here.

[2] https://developer.bluetooth.org/gatt/characteristics/Pages/CharacteristicsHome.aspx

Armed with the UUIDs of the characteristics we are looking for, we can attempt to discover them. The process is similar to how we discovered services, but with a slightly different call.

1. First, we add NSString values to hold the UUIDs

Objective C

```
NSString *ALERT_LEVEL_CHARACTERISTIC_UUID = @"2A06";
NSString  *CLIENT_PROXIMITY_CHARACTERISTIC_UUID =  @"3E099911-293F-11E4-93BD-AFD0FE6D1DFD"
```

And we also add variables to hold the characteristics when we find them.

```
// Alert Level belonging to the Link Loss Service
CBCharacteristic *LL_AlertLevelCharacteristic;
// Immediate Alert Service
CBCharacteristic *IA_AlertLevelCharacteristic;
// Client Proximity custom characteristic
CBCharacteristic *ClientProximityCharacteristic;
```

2. Then we need to also define a function we can call when a characteristic is found

Objective C

```
-(void) OnDiscoverCharacteristics:(NSArray *)c
{
    for (CBCharacteristic *character in c)
    {
        if ([character.UUID isEqual:[CBUUID UUIDWithString: ALERT_LEVEL_CHARACTERISTIC_UUID]])
        {
            if ([character.service.UUID isEqual:[CBUUID UUIDWithString: LINK_LOSS_SERVICE_UUID]])
            {
                NSLog(@"Got Link Loss Service's Alert Level characteristic");
                LL_AlertLevelCharacteristic = character;
            } else {
                if ([character.service.UUID isEqual:[CBUUID UUIDWithString:
IMMEDIATE_ALERT_SERVICE_UUID]])
                {
                    NSLog(@"Got Immediate Alert Service's Alert Level characteristic");
                    IA_AlertLevelCharacteristic = character;
                }
            }
        }
        else if([character.UUID isEqual:[CBUUID UUIDWithString:
CLIENT_PROXIMITY_CHARACTERISTIC_UUID]])
        {
            NSLog(@"Got Client Proximity characteristic");
            ClientProximityCharacteristic = character;
        }
    }
}
```

This function iterates over the characteristics and saves a reference if the UUID value matches one of the NSString values.

3. Next up we need to call getAllCharacteristicsForService, passing in the currently connected peripheral and the service we want to explore. To do this we can make a few changes to the onDiscoverServices function:

Objective C

```
-(void) OnDiscoverServices:(NSArray *)s
{
    for (CBService *service in s)
    {
        NSLog(@"SERVICE : %s", [self.bleAdapter CBUUIDToString: service.UUID]);

        if ([service.UUID isEqual:[CBUUID UUIDWithString: LINK_LOSS_SERVICE_UUID]])
        {
            NSLog(@"Got Link Loss Service");
            LinkLoss = service;
            [self.bleAdapter getAllCharacteristicsForService:self.bleAdapter.activePeripheral
service:LinkLoss];
        }
        if ([service.UUID isEqual:[CBUUID UUIDWithString: IMMEDIATE_ALERT_SERVICE_UUID]])
        {
            NSLog(@"Got Immediate Alert Service");
            ImmediateAlert = service;
            [self.bleAdapter getAllCharacteristicsForService:self.bleAdapter.activePeripheral
service:ImmediateAlert];
        }
        if ([service.UUID isEqual:[CBUUID UUIDWithString: PROXIMITY_MONITORING_SERVICE_UUID]])
        {
            NSLog(@"Got Proximity Monitoring Service");
            ProximityMonitoring = service;
            [self.bleAdapter getAllCharacteristicsForService:self.bleAdapter.activePeripheral
service:ProximityMonitoring];
        }
    }
}
```

Task 5 – Reading and writing a characteristic

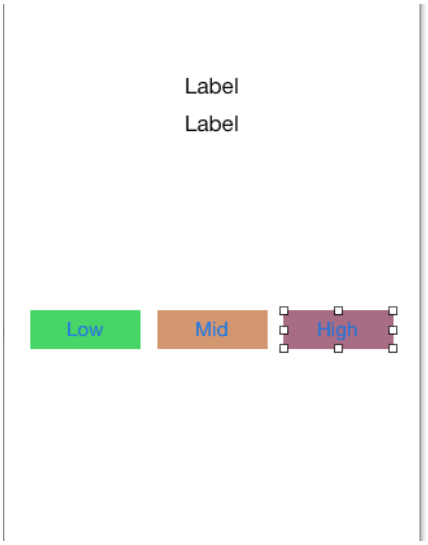
In general, once you have a reference to a characteristic, you can test for descriptors and permissions. In our case, however, we already know what the permissions are for the characteristics we're interested in, so will skip ahead to reading and writing.

NOTE: You can look up the details for a standard characteristic like Alert Level on the Bluetooth.org[3]

As we can see from the Bluetooth.org specification, the value for Alert Level must be a number between 0 and 2, corresponding to different alert levels. So, if we are reading this characteristic we can expect a number between 0 and 2.

1. Add three buttons to the peripheral view page and create three corresponding actions. Name the actions send_lowAlert, send_midAlert, send_highAlert.

After you have added the three buttons and their actions, your view should look like this:



and your peripheralViewController.h file should look like this:

Objective C

```
#import <UIKit/UIKit.h>
#import "BLEAdapter.h"

@interface PeripheralViewController : UIViewController
@property (weak, nonatomic) IBOutlet UILabel *lbl_name;
```

3 https://developer.bluetooth.org/gatt/characteristics/Pages/CharacteristicViewer.aspx?u=org.bluetooth.characteristic.alert_level.xml

```
- (void) setData: (BLEAdapter *) adapter : (CBPeripheral *) peripheral;

@property (strong, nonatomic) BLEAdapter *bleAdapter;
@property (strong, nonatomic) CBPeripheral *toConnect;
@property (weak, nonatomic) IBOutlet UIButton *btn_connect;

- (IBAction)connect_click:(id)sender;

@property (weak, nonatomic) IBOutlet UILabel *lbl_rssi;
@property (weak, nonatomic) IBOutlet UIView *pnl_hotCold;

- (IBAction)send_lowAlert:(id)sender;
- (IBAction)send_midAlert:(id)sender;
- (IBAction)send_highAlert:(id)sender;

@end
```

Adding these actions should have created the following lines of code in your PeripheralsViewController.m file:

Objective C

```
- (IBAction)send_lowAlert:(id)sender
{
}
- (IBAction)send_midAlert:(id)sender
{
}
- (IBAction)send_highAlert:(id)sender
{
}
```

2. Prepare functions to convert and send the data to the Bluetooth Smart device.

Using these three functions, we want to write an appropriate alert level of 0, 1 or 2 to the Alert Level characteristic belonging to the Link Loss service on the Arduino device. We also store the selected value locally for use with the “Make a Noise” button and the Immediate Alert service.

In order to do this we need to first create a suitable function which will take care of the establishing the data ain the right format and then sending the data via a characteristic write operation.

In PeripheralViewController.m, add the following code:

Objective C

```
-(void)changeAlertLevel:(int) level
{
    if (level <= 2 && link != NULL)
    {
        CurrentAlertLevel = level;

        char* bytes = (char*) &level;
        int len = 1;
        NSData* tmpData = [NSData dataWithBytes:bytes length:len];
```

```
        NSLog(@"Link loss UUID is %@", LinkLoss.UUID);
        NSLog(@"Alert Level characteristic UUID is %@", ALERT_LEVEL_CHARACTERISTIC_UUID);

        NSLog(@"incoming alert level %D", level);

        [self.bleAdapter.activePeripheral writeValue:tmpData
forCharacteristic:LL_AlertLevelCharacteristic type:CBCharacteristicWriteWithResponse];
    }
}
```

Make sure you add the following variable definition near the top of the source file too:

```
int CurrentAlertLevel;
```

This function takes an integer value as a parameter and converts it to NSData, which forms the data part of the write function. We have a reference to the alert level characteristic of the link loss service, so we are able to write directly to it using the call which is highlighted in the code fragment above.

The result of this function will trigger a function in the BLEAdapter called didWriteValueForCharacteristic. This in turn requires a function in our PeripheralViewController called OnWriteDataChanged and that takes a Boolean value as a parameter.

Now add the following function to our PeripheralViewController.m file

Objective C

```
-(void) OnWriteDataChanged:(BOOL)status
{
    if (status)
    {
        NSLog(@"Successful Write");
        // if we just updated the alert level then we should display the new current value in the
UI... if not it's simplest to do this anyway
        self.lbl_alert.text = [NSString stringWithFormat:@"alert: %d", CurrentAlertLevel];
    }else{
        NSLog(@"Error Writing value");
    }
}
```

This function will display a message to the output console stating if the write call was successful and update the UI label which displays the currently selected alert level value.

3. Hook up the button action to the data send commands.

We can now change the button actions set up earlier, as follows:

Objective C

```
- (IBAction)send_lowAlert:(id)sender
{
   [self changeAlertLevel:0];
}

- (IBAction)send_midAlert:(id)sender
{
   [self changeAlertLevel:1];
```

```
}

- (IBAction)send_highAlert:(id)sender
{
   [self changeAlertLevel:2];
}
```

4. Read the characteristic back.

We'll read the link loss service's alert level value back from the device periodically just so we have an example of reading a characteristic.

Add the following code to the end of the readRSSITimer function to read the Link Loss service's Alert Level characteristic:

Objective C

```
if (LL_AlertLevelCharacteristic != NULL)
{
    int serviceCode = [self.bleAdapter CBUUIDToInt: LinkLoss.UUID];
    int characteristicCode = [self.bleAdapter CBUUIDToInt: LL_AlertLevelCharacteristic.UUID];

    [self.bleAdapter readValue:(int)serviceCode characteristicUUID:(int)characteristicCode
p:(CBPeripheral *)self.bleAdapter.activePeripheral];
}
```

In the above code, we check to see if we have found the characteristic, then convert both the service and characteristic UUID to an int value. We can then call the readValue function in BLEAdapter. This follows the same pattern as the write function. When the data is read, we get a callback on the didUpdateValueForCharacteristic, which in turn is expecting our PeripheralViewController to implement an onReadDataChanged function,.

So now we need to add the following function to PeripheralViewController.m:

Objective C

```
-(void) OnReadDataChanged:(BOOL)status : (CBCharacteristic*) characteristic
{
    if (status)
    {
        NSLog(@"Successful Read %@", characteristic.value);
        self.lbl_alert.text = [NSString stringWithFormat:@"alert: %@", characteristic.value];
    }else{
        NSLog(@"Error Reading value");
    }
}
```

In the above, the Boolean value tells us if the read command succeeded. We also have the characteristic, which means we can read its 'value' property.

Task Complete – confirm write and read of Alert Level

If you run your application now you should be able to write a new value to the Alert Level on the Bluetooth Smart device, and you should also see the changed value in the output console. If you've

completed the Arduino lab or simply loaded the provided solution code onto it and attached the suggested circuit, you'll also see an LED flash to indicate receipt of that write operation.

Task 6 – Triggering Immediate Alert on the Bluetooth Smart device

Next we want to use the write functionality to trigger an "immediate alert" event on the Bluetooth Smart device.

We are going to create a UI button that will trigger the writing of the current alert level value to the Alert Level characteristic of the Immediate Alert service.

1. First, place a button on the Peripheral view, create an outlet and name it "btn_makeNoise".
2. Next, create an action called "noise_click". Creating that action will generate the following code in your PeripheralViewController.m file

Objective C

```
- (IBAction)noise_click:(id)sender
{
}
```

3. Now we will make the button click in the UI write a value to the Alert Level characteristic of the Immediate Alert service on the Bluetooth Smart device.

Add the following code to the noise_click function:

Objective C

```
- (IBAction)noise_click:(id)sender
{

    if (IA_AlertLevelCharacteristic != NULL)
    {

        char bytes[1];
        Byte val = CurrentAlertLevel;
        bytes[0] = val;
        int len = 1;
        NSData* tmpData = [NSData dataWithBytes:bytes length:len];

        [self.bleAdapter writeValueNoResponse :(CBCharacteristic *)IA_AlertLevelCharacteristic
p:(CBPeripheral *)self.bleAdapter.activePeripheral data:(NSData *)tmpData];

        NSLog(@"Requesting alert_level written to Immediate Alert Service's Alert Level without
response");

    }
}
```

In the above code, we are making sure we have the Immediate Alert, AlertLevel Characteristic. If yes, we then convert the current alert level int value of 0, 1 or 2 to the NSData type. We then the WriteValueNoResponse function on the BLEAdapter. We use this particular function because the specification for the Immediate Alert service indicates that the Attribute Protocol operation that this characteristic supports, is "write without response". Assuming you've implemented your Arduino code and circuit board in line with our Arduino lab, you should see all three LEDs flash and, if the alert level is 1 or 2, you should also hear the buzzer emit a noise.

Task complete – Push-button sound

Running the application now on your iOS device, you should be able to scan for Bluetooth Smart devices, connect to a device, set the alert level for the Arduino board, and read the newly-set level. You should also be able to trigger an immediate alert event by clicking a button on the phone.

Task 7 – Sound alarm on phone at link loss

The penultimate task is to implement the link loss on the phone. For the purposes of this demo we are using the following business logic: if we have set the alert level to its highest state AND the link is broken between the phone and the board, we will sound an alarm on the phone.

1. First thing we need is a sound file. Add this to your project by clicking File > Add files to <project name>, navigating to [Lab install folder/ iOS/Source] and selecting the Beep.wav file. Drag the file into the Supporting Files.
2. Next, import the following .h files into your peripheralVIewController.h file:

Objective C

```
#import <AVFoundation/AVAudioPlayer.h>
#import <AVFoundation/AVFoundation.h>
```

3. Now add a new AVAudioplayer property. This will be the player that makes the sound when the phone disconnects from the Bluetooth SMART device.

Objective C

```
@property (strong, nonatomic) AVAudioPlayer *player;
```

4. In your PeripheralViewController.m file , synthesize the player property

Objective C

```
@synthesize player;
```

5. Next we add two new functions which will play and stop the ‘Bleep’ sound. Add the following to the PeripheralViewController.m file:

Objective C

```
-(void) playSound
{
    NSString *soundFilePath = [[NSBundle mainBundle] pathForResource:@"Beep" ofType:@"wav"];
    NSURL *soundFileURL = [NSURL fileURLWithPath:soundFilePath];
    player = [[AVAudioPlayer alloc] initWithContentsOfURL:soundFileURL error:nil];
    player.numberOfLoops = -1; //infinite
    [player play];
}

-(void) stopSound
{
```

```
    [player stop];
}
```

This code initializes the player and sets play in a constant loop until the stopSound function is called.

6. Finally we need to handle the connection failed and check which alert we are in and whether we need to sound the alarm.

We can do this by adapting our connectionFailed function like so:

Objective C

```
-(void)connectionFailed
{
    NSLog(@"GETTING HERE CONNECTION CLOSED alert level %d",CurrentAlertLevel );
    [rssiTimer invalidate];
    [self.btn_connect setEnabled: true];
    [self.btnMakeNoise setEnabled:false];

    if (CurrentAlertLevel == 2)
    {
        [self playSound];
        // try to reconnect
        [self.bleAdapter connectPeripheral:self.toConnect status:TRUE];
    }
    else
    {
        [self displayMessage:@"Could not connect to device"];
        [self.navigationController popToRootViewControllerAnimated:TRUE];
    }
}
```

Task 8 – Proximity Monitoring

The final task is to implement a Switch control which allows us to toggle proximity monitoring on or off. “Proximity monitoring” entails sending the measured RSSI value and the proximity band (1=near, 2=middle distance, 3=far) corresponding to the color of our coded rectangle, to the custom Proximity Monitoring service via a characteristic write operation. When the Arduino receives the write request it will light one of its three LEDs, based on the proximity band value. If you’ve also rigged up a serial UART LCD display, the RSSI value will be displayed there too.

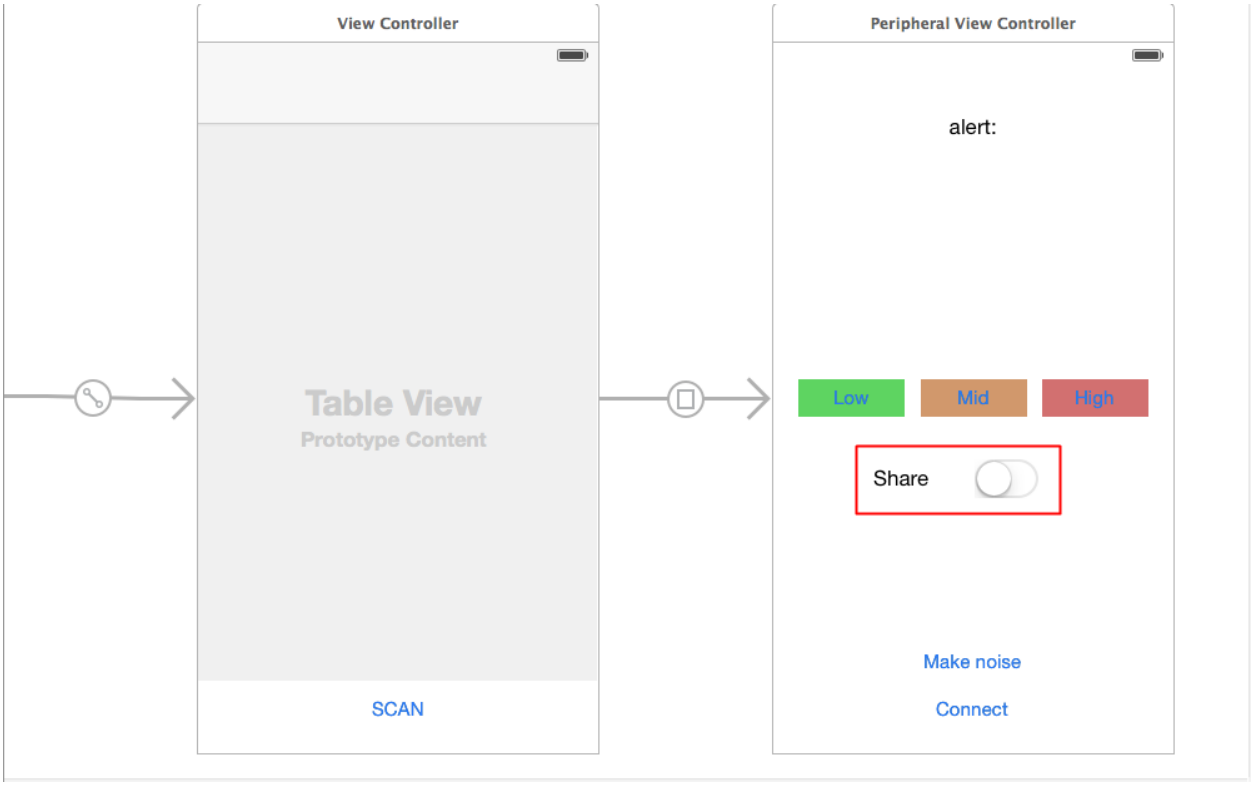
Add the following bool variable after your declaration for CurrentAlertLevel.

Objective C

```
int CurrentAlertLevel = 2;
bool share_proximity_data = false;
```

We need to set this variable in response to our slider control being changed.

Next, add a Switch to the UI:



Implement the following function, to be called when the state of the new Switch control changes and link the Switch to the function in your storyboard:

```
- (IBAction)share_switch_changed:(id)sender
{

    UISwitch *mySwitch = (UISwitch *)sender;
    if ([mySwitch isOn]) {
        NSLog(@"Proximity Sharing Enabled!");
        share_proximity_data = true;
    } else {
        NSLog(@"Proximity Sharing Disabled");
        share_proximity_data = false;
        // tell the arduino to switch off LEDS and clear LCD display
        char bytes[2];
        bytes[0] = 0;
        bytes[1] = 0;
        int len = 2;
        NSData* tmpData = [NSData dataWithBytes:bytes length:len];
        [self.bleAdapter writeValueNoResponse :(CBCharacteristic *)ClientProximityCharacteristic
p:(CBPeripheral *)self.bleAdapter.activePeripheral data:(NSData *)tmpData];

    }
}
```

Note that when proximity sharing gets switched off, we need to “tell” the Arduino that this has occurred so that it can switch off any LED that is currently lit. The code which accomplishes this is highlighted above and involves using a write with no response to send an rssi value of zero and a proximity band value of zero. The Arduino sketch has been programmed to interpret this as “sharing is now off”.

Now update the readRssiTimer function to incorporate proximity sharing. Add the following code in an appropriate place in the function:

```
if (share_proximity_data) {
    char bytes[2];
    bytes[0] = proximity_band + 1;
    bytes[1] = rssi * -1;
    int len = 2;
    NSData* tmpData = [NSData dataWithBytes:bytes length:len];
    [self.bleAdapter writeValueNoResponse :(CBCharacteristic *)ClientProximityCharacteristic
p:(CBPeripheral *)self.bleAdapter.activePeripheral data:(NSData *)tmpData];
}
```

Summary of Exercise 2

You now have all the pieces to run the full demonstration scenario.

Build and run the application now, and connecting to the Bluetooth Smart Device. As you take your phone further from the board you should notice the colour of the hot/cold panel change. Also, if you set the alert level to high and keep getting further from the board you will hear a siren. This will loop until you either connect the phone to the board again, or close the app. In this event, you should note the board will also make a noise until either it has looped around 100 times or the reset button is clicked. Selecting the Make a Noise button will make the Arduino flash all three LEDs several times. If the currently selected alert level value is set to medium (1) or high (2), it will also make a noise. Enabling proximity sharing via the Switch control will cause the same color LED as the hot spot rectangle to be illuminated and if the circuit has an LCD display attached, the RSSI value will also be displayed.

To review the final version of the code, open the Xcode project in the *Final* subfolder of *[Lab Install Folder]/ iOS/Source* Ex 2. You will also see a few tweaks in this final project for adding button states and updating a new label which displays the current alert level.

Well done! *By following this hands-on lab you have created an application that scans for Bluetooth Smart devices, scans their services, connects to them, reads and writes characteristic values, and monitors the RSSI value.*