Bluetooth®

Developer Study Guide

An introduction to Bluetooth® Beacons – iOS

Release: 1.3.2

Document Version : 1.3.2

Last Updated: 21st December 2020

Running Time: 90 to 120 minutes

CONTENTS

Revision History

Version	Date	Author	Changes
1.0.0	16th December 2016		Initial version
1.1.0	1st August 2018	Kai Ren Bluetooth SIG	Raspberry Pi Tutorial: removed all the BDS references in the Raspberry Pi Tutorial. Raspberry Pi Tutorial: user doesn't need to download shell file on the host computer, user can just download the shell file by command on R Pi3 through SSH remote access or local access. Android Tutorial: add a "template" Android project for user, they can start copy and paste on it. User doesn't need to create a new Android Studio project. Android Tutorial: add a "fullSolution" Android project for user, user can build and install the App on their Android devices. This project is for user to debug if user has any problem for the implementation on "template" project. Android Tutorial: update Android project gradle, make the project can be compiled by latest Android Studio. Android Tutorial: double checked, the Android projects can be built on Windows and macOS host computer. iOS Tutorial: update some APIs which are deprecated. iOS Tutorial: optimize the work flow.
1.2.0	18th December 2018	Kai Ren Bluetooth SIG	Name changed to "Develoepr Study Guide" Fixed an Android Studio compilation warning.
1.3.0	11th October 2019	Kai Ren Bluetooth SIG	iOS retrofit the whole study guide to make it be compatible with Swift, instead of Objective-C. add new template Swift source code Raspberry Pi: Add latest Raspberry Pi4 support Update BlueZ installation from v5.49 to v5.50
1.3.1	28th October 2020	Martin Woolley Bluetooth SIG	Updated Android tutorial to incorporate permissions requests and be compatible with Android 10.

1.3.2	21st December 2020	Martin Woolley Bluetooth SIG	Language changes

Overview

In this guide, we will show you how to create a simple application that can discover Bluetooth® beacons – AltBeacons – close to your iPhone/iPad. The application you build will present a list of nearby beacons with details:

- manufacturer UUID (first 16 bytes of BEACON ID)
- additional data (arguments) from AltBeacon (last 4 bytes of BEACON ID)
- current RSSI and reference RSSI

The application will refresh this list every few seconds, adding newly discovered beacons and removing any beacons that are no longer in range.

Beacons and AltBeacons

Beacons are low-power Bluetooth devices that advertise a data packet containing some unique identifier and secondary bits of information. The AltBeacon specification is an open and interoperable specification for proximity beacons using Bluetooth. AltBeacons, by design, are not tied to any particular manufacturer or chipset.

Prerequisites

This tutorial is tested under below conditions:

- macOS High Sierra v10.13.5
- Xcode v9.4
- iOS v11.4
- An AltBeacon. This can be a Raspberry Pi3 configured as per our first guide in this series, or some other device that conforms to the AltBeacon open specification.

Running Time

The two exercises in this lab should take 1 to 2 hours to complete.

How this works

Follow the exercises, tasks and steps below to get to the finished product. If you want to skip ahead, or if you want to check your work, or if you get stuck, refer to the completed source project provided as part of the guide.

Please note: To complete this lab you will need to be comfortable with Xcode and Objective-C.

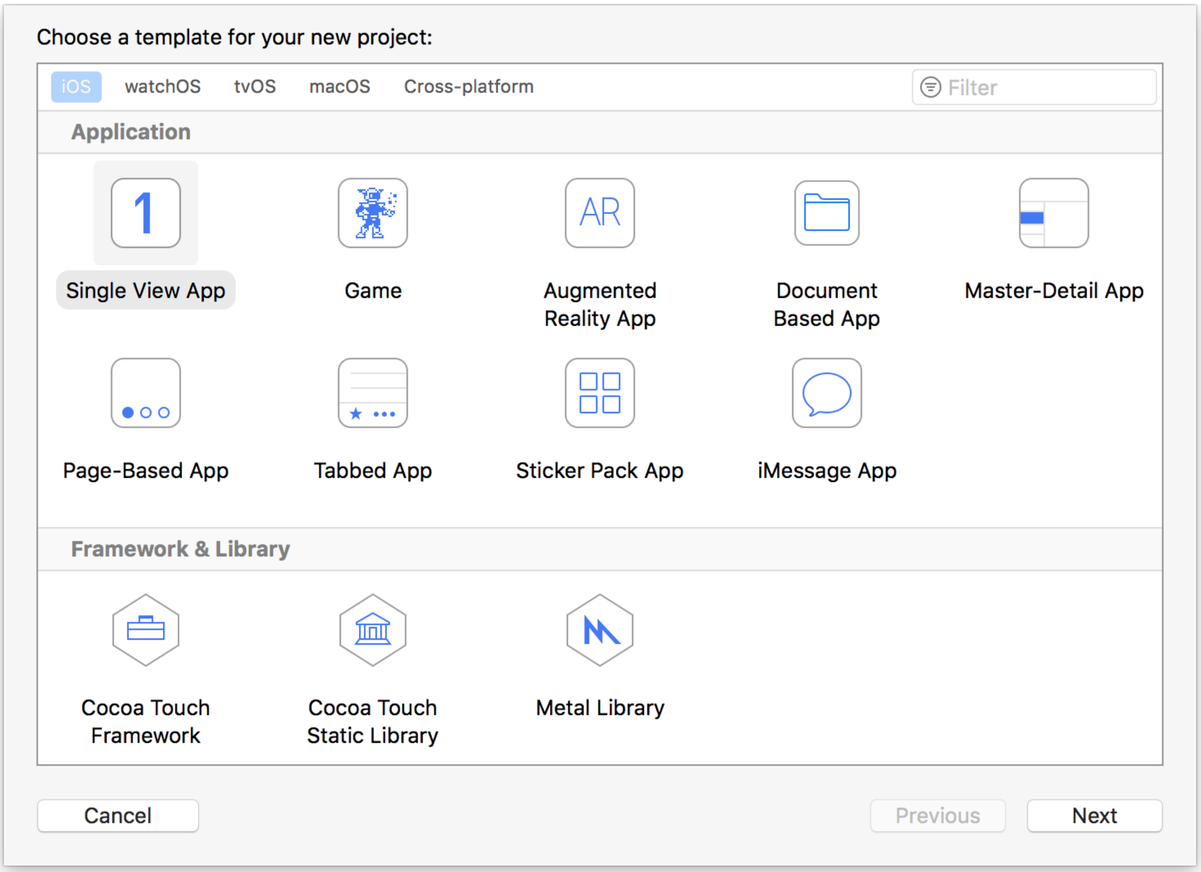
Exercise 1 – A simple app to show AltBeacon data

Task 1 – Create a basic iOS app test this on a real device

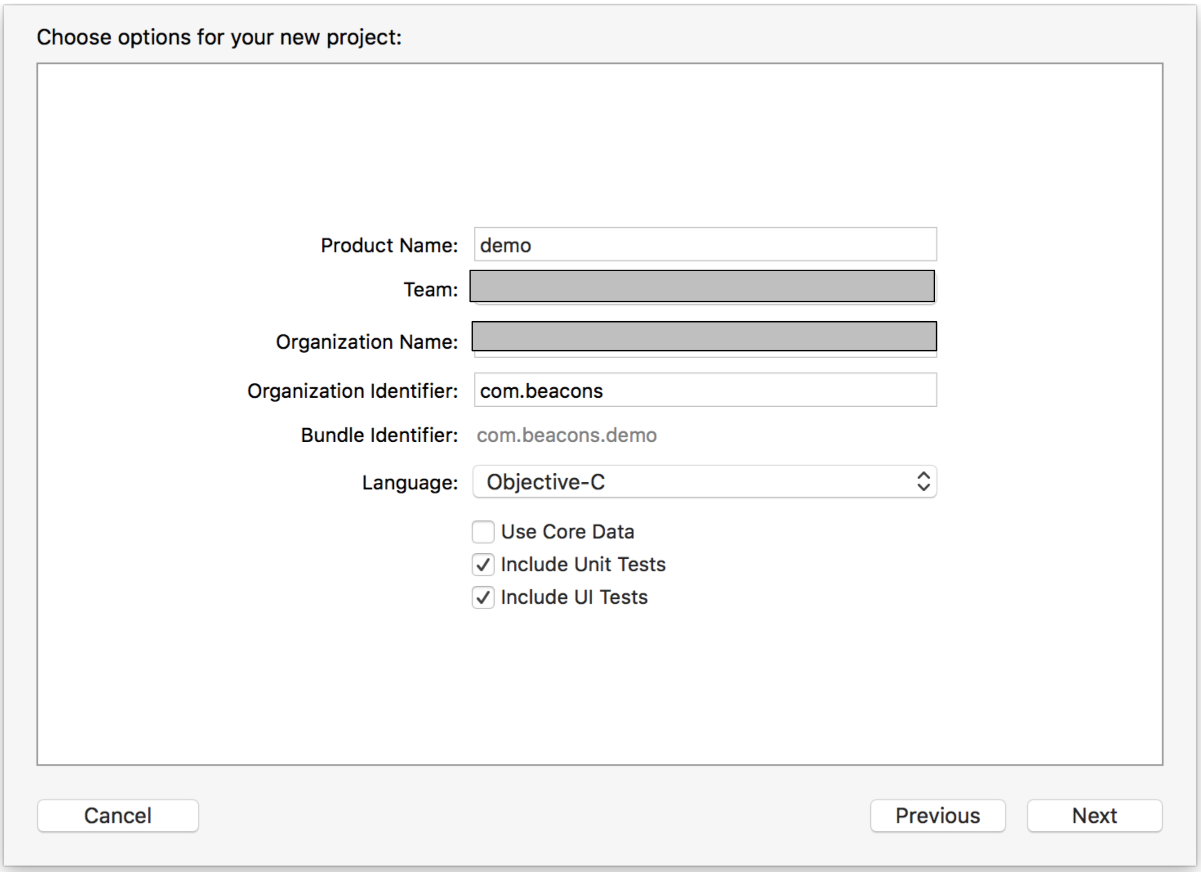
Create a new project

Start Xcode and create a new project with these parameters:

- iOS Single View Application


Choose a template for your new project:

- name: demo
- organization identifier: com.beacons
- language: Objective-C
- devices: Universal
- use core data: un-ticked


Choose options for your new project:

Product Name:	demo
Team:	[redacted]
Organization Name:	[redacted]
Organization Identifier:	com.beacons
Bundle Identifier:	com.beacons.demo
Language:	Objective-C

- [ ] Use Core Data
- [✓] Include Unit Tests
- [✓] Include UI Tests

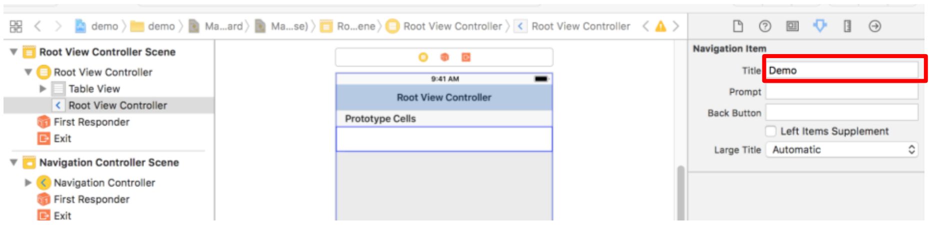
Click Next button and store this project to a suitable location.

Now you can run the application on an iPhone or iPad. The app will show an empty screen, as expected.

Set up the main scene for our application

When you open Main.storyboard, you will find one standard controller. We will set up the storyboard from scratch, so we need to delete everything and then add new elements:

- Delete anything you see on the canvas.
- Drag and drop a Navigation Controller onto the canvas. This should also add a Table View Controller inside – which is good, as we need to show a list of items.
- Select the Navigation Controller (not Navigation Controller Scene) and mark "Is Initial View Controller" in the Attributes Inspector panel.
- Select “Root View Controller” and change Title to “Demo” (or anything else you like) in Navigation Item inside the Table View Controller (the bar at the top) like below picture shown.



Navigation Item
Title Demo
Prompt
Back Button
☐ Left Items Supplement
Large Title Automatic

- Select the cell prototype inside table view, and change the following properties in the Attributes Inspector as below picture shown:
  - Style to Subtitle.
  - Identifier to "BeaconItemCell" (we'll need this later).

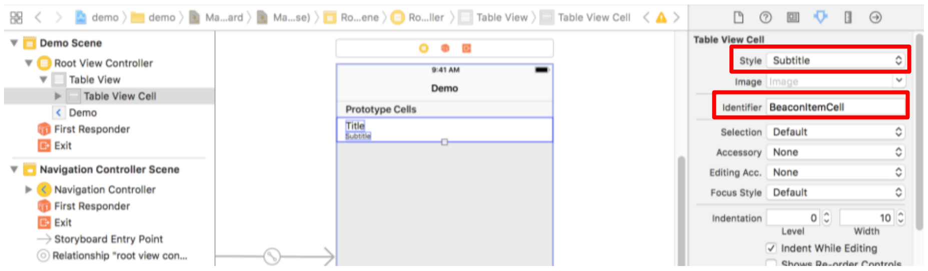
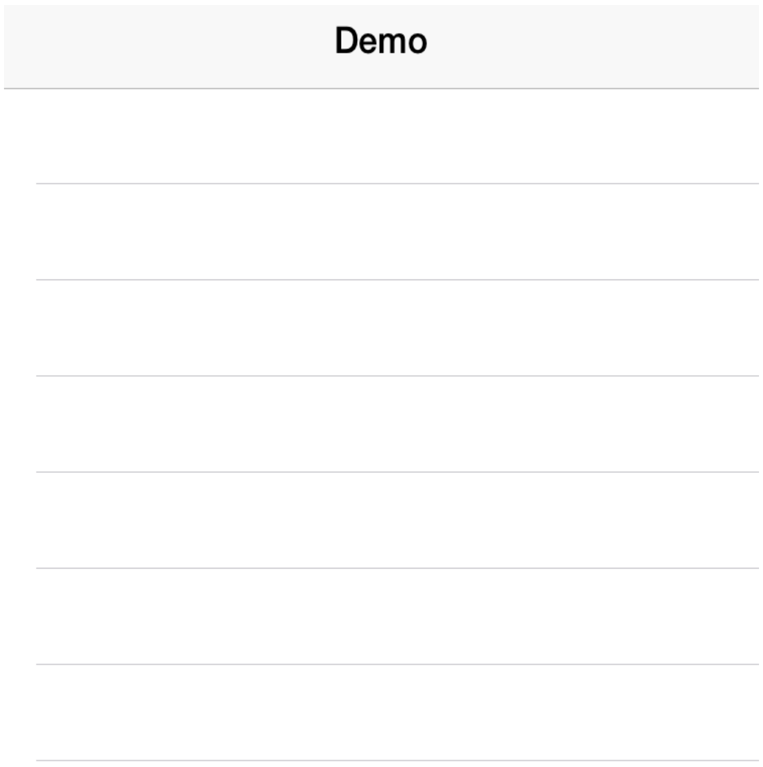


Table View Cell
Style Subtitle
Image
Identifier BeaconItemCell
Selection Default
Accessory None
Editing Acc. None
Focus Style Default
Indentation 0 Level 10 Width
✓ Indent While Editing
☐ Shows Re-order Controls

Now run the application again. You should see an empty list and a title at the top of the application screen, as shown below:



Empty project, ready to be implemented, running on iPhone 6

Task 2 – Implement the Model and View elements

We need to prepare a model for storing the beacon data, and we need to update our view to be able to show and handle that data.

Adding a BeaconModel

Add a new Cocoa Touch class to the project which is called `BeaconModel`. Go to the header file and make the following changes, adding the properties for all the data we want:

In `BeaconModel.h`

```
@interface BeaconModel : NSObject

@property (nonatomic, strong) NSString* uuid;
@property (nonatomic)             short arg1;
@property (nonatomic)             short arg2;
@property (nonatomic)               int referenceRssi;
@property (nonatomic)               int currentRssi;

@property (nonatomic, strong) NSString* beaconId;
@property (nonatomic)              long timestamp;

@end
```

This is all we need for now in our model. Let's move onto the view part.

Adding DataSource methods

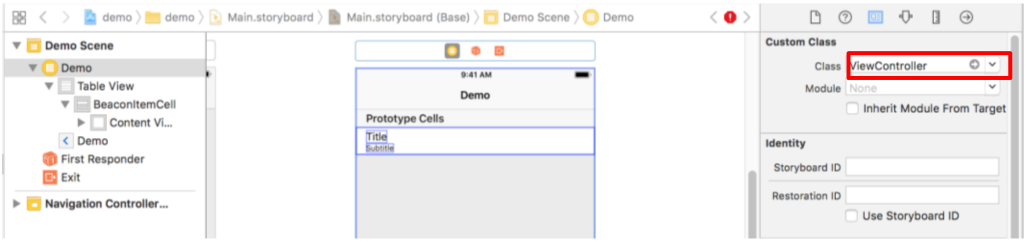
We will make our view controller act as a DataSource for the table view.

Edit the `ViewController` class header (`ViewController.h`) to adopt the `UITableViewDataSource` protocol:

In `ViewController.h`

```
...
#import "BeaconModel.h"

@interface ViewController : UITableViewController <UITableViewDataSource>
```

Click `Main.storyboard` and select `Demo` Table View Controller and set `ViewController` as a Class in the Custom Class section in the Identity Panel like below picture shown.



Custom Class
Class: ViewController
Module: None
☐ Inherit Module From Target

Identity
Storyboard ID:
Restoration ID:
☐ Use Storyboard ID

Add methods required by table view into implementation file in `ViewController.m`:

In `ViewController.m`

```
- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    return 0;
}

- (UITableViewCell*) tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString* cellIdentifier = @"BeaconItemCell";

    UITableViewCell* cell = [tableView
         dequeueReusableCellWithIdentifier:cellIdentifier
                              forIndexPath:indexPath];

    return cell;
}
```

That is giving us empty list view, with empty cells for this moment. Let's add a backend `NSMutableArray` which will hold all items. Add a new property to the header (`ViewController.h`):

In `ViewController.h`

```
...
@interface ViewController : UITableViewController <UITableViewDataSource>
@property (nonatomic, strong) NSMutableArray* beacons;

@end
```

Allocate in the viewDidLoad callback (replace or edit as required):

In `ViewController.m`

```
...
@implementation ViewController


- (void)viewDidLoad {
    [super viewDidLoad];

    _beacons = [[NSMutableArray alloc] init];
}
```

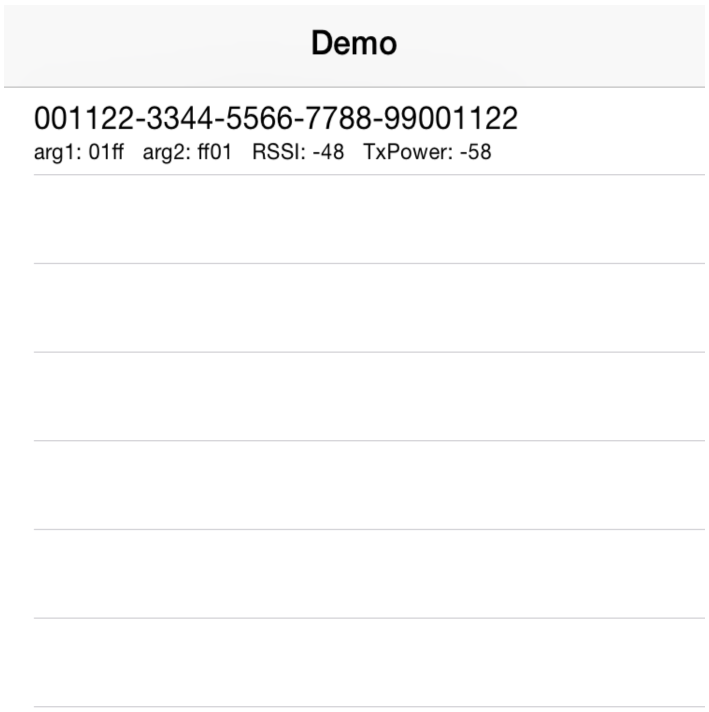
Now you can update both methods connected with `UITableViewDataSource` protocol:

```
- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    return _beacons.count;
}

- (UITableViewCell*) tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString* cellIdentifier = @"BeaconItemCell";

    UITableViewCell* cell = [self.tableView
         dequeueReusableCellWithIdentifier:cellIdentifier
                              forIndexPath:indexPath];

    BeaconModel* beacon = [self.beacons objectAtIndex:indexPath.row];

    cell.textLabel.text = beacon.uuid;
    cell.detailTextLabel.text = [NSString
             stringWithFormat:@"arg1: %04x arg2: %04x RSSI: %d TxPower: %d",
                  beacon.arg1 & 0xffff,
                  beacon.arg2 & 0xffff,
                  beacon.currentRssi,
                  beacon.referenceRssi];
    return cell;
}
```

Now, when you add any BeaconModel to that list and reload data, it should be visible in the UI. As a quick test, add a dummy beacon in the `viewDidLoad` callback:

```
BeaconModel* test = [[BeaconModel alloc] init];
test.uuid = @"001122-3344-5566-7788-99001122";
test.arg1 = 0x01ff;
test.arg2 = 0xff01;
test.referenceRssi = -58;
test.currentRssi = -48;
[_beacons addObject:test];
```

Run the app and you should see the dummy beacon in the list, as shown in the screenshot below.


Demo

001122-3344-5566-7788-99001122
arg1: 01ff   arg2: ff01   RSSI: -48   TxPower: -58

Application UI displaying test, fake data

Exercise Complete

Well done! You have successfully created an app that is ready to show our AltBeacon data in the right format. In the next exercise you will connect your app to a real AltBeacon and show the real data from that AltBeacon.

Exercise 2 – Connecting to a real AltBeacon

Task 1 – Add Bluetooth Hardware management code

Now that we have everything ready to show our AltBeacons, we need our app to make sure Bluetooth is available and running on the iOS device.

The main API we need for running Bluetooth Low Energy (LE) scanning is the `CBCentralManager` class from the `CoreBluetooth` library.

In your ViewController implementation file (`ViewController.m`) import the core Bluetooth module, and add one new property of type `CBCentralManager`:

In `ViewController.m`

```
...
@import CoreBluetooth;

@interface ViewController ()

@property (nonatomic, strong) CBCentralManager* bleManager;
...

@end

@implementation ViewController

...

@end
```

When initializing our `bleManager`, we will need to provide an object which implements all callbacks from the Bluetooth hardware for scanning results, Bluetooth state changes etc. This object needs to adopt the `CBCentralManagerDelegate` protocol. We will implement this directly inside our view controller.

Put the `CBCentralManagerDelegate` protocol inside the list of protocols for our view controller:

In `ViewController.h`

```
...
@import CoreBluetooth;

@interface ViewController : UITableViewController <UITableViewDataSource,
                                                   CBCentralManagerDelegate>
```

And add an implementation of the required methods:

In `ViewController.m`

```
- (void) centralManagerDidUpdateState:(CBCentralManager *)central {

}

- (void) centralManager:(CBCentralManager *)central
  didDiscoverPeripheral:(CBPeripheral *)peripheral
      advertisementData:(NSDictionary *)advertisementData
                   RSSI:(NSNumber *)rssi
{
    // reporting callback
}
```

Now we are able to initialize the BLE manager. Add the code below into the `viewDidLoad` method (and remove the code to add a dummy beacon):

In `ViewController.m`

```
- (void)viewDidLoad {
    ...

    _bleManager = [[CBCentralManager alloc] initWithDelegate:self queue:nil];
}
```

We will receive a `centralManagerDidUpdateState:` callback any time the status of the Bluetooth hardware changes. We can use this callback to find out and react to the current state of Bluetooth LE on the device. For our simple demo application, we will want to start scanning as soon as BLE is available and enabled. Outside of this event, we want our application to wait and do nothing.

Extend the `centralManagerDidUpdateState:` callback as follows:

In `ViewController.m`

```
- (void) centralManagerDidUpdateState:(CBCentralManager *)central {
    if ([central state] == CBManagerStatePoweredOn) {
        [self startScanning];
    }
    else {
        [self stopScanning];
    }
}
```

We have used two functions (`startScanning` and `stopScanning`) which are not defined yet. Let's declare them here

In `ViewController.m`

```
- (void) startScanning {

}

- (void) stopScanning {

}
```

Whenever the user leaves our app we would like to stop scanning, and we want to start scanning again as soon as the user returns to our application. In order to do this, we need additional view controller callbacks where we will call our (just declared) functions:

In `ViewController.m`

```
- (void) viewWillDisappear:(BOOL)animated {
    [self stopScanning];
}

- (void) viewDidAppear:(BOOL)animated {
    [self startScanning];
}
```

Stopping discovery mode is very easy. Implement `stopScanning` method with this code:

In `ViewController.m`

```
- (void) stopScanning {
    [_bleManager stopScan];
}
```

Starting discovery mode requires a little bit more code. First, we need to define an NSDictionary which will contain all parameters/options for our scanning:

In `ViewController.m`

```
- (void) startScanning {
    static NSMutableDictionary* options = nil;
    if(options == nil) {
        options = [[NSMutableDictionary alloc] init];
        [options setValue:[NSNumber numberWithBool:YES]
                   forKey:CBCentralManagerScanOptionAllowDuplicatesKey];
    }

    // we will put scanning start code below . . .
}
```

We requested the Bluetooth LE hardware to report the same device multiple times. By default, iOS reports every device only once. In most cases that is fine. In our case however, we are constantly interested in the current signal strength (RSSI). We also want to be sure that any peripheral which we find is still near the device (so we can remove it from the list if it’s out of range).

With that options dictionary, we can ask `bleManager` to start searching for nearby beacons, insert below code into `startScanning()`:

In `ViewController.m`

```
- (void) startScanning {
    ...

    // we will put scanning start code below . . .
    // start scanning:
    [_bleManager scanForPeripheralsWithServices:nil options:options];
}
```

That is all we need to get Bluetooth running and scanning for our AltBeacons. The next step is where it gets more exciting!

Task 2 – Filter for AltBeacons and parse the data

If you run the application, you will still see an empty screen. This is because we are not doing anything with the data returned in the `centralManager:didDiscoverPeripheral:...` callback.

In that callback we would like to know if a reported device is an AltBeacon. If it is, we would like to get and parse the advertisement data and insert/update the model in the `beacons` list.

Get advertisement data

First, we need to get advertisement data in a format suitable for us. We would like to have an `unsigned char*` array, which will be easy to parse. Inside the `centralManager:didDiscoverPeripheral:` callback, at the very beginning of it, add this:

In `ViewController.m`

```
NSData* data = [advertisementData
                    objectForKey:CBAdvertisementDataManufacturerDataKey];
if (data == nil) return;
unsigned char* bytes = (unsigned char*)[data bytes];
```

Define constants for parsing and validating AltBeacon

Next, we need to refer to the AltBeacon protocol specification (here on GitHub) and check if what we get conforms to that protocol. In order to do that, we need a few constants describing where particular fields should be placed inside the advertisement data. We also define what values should be used by an AltBeacon.

Add these constants to your view controller code just above `centralManager:didDiscoverPeripheral:` callback:

In `ViewController.m`

```
static const int BEACON_CODE_INDEX = 2;
static const int BEACON_CODE_VALUE = 0xbeac;
static const int UUID_START = 4;
static const int UUID_STOP = 19;
static const int CONTENT_START = 20;
static const int CONTENT_STOP = 23;
static const int REFERENCE_RSSI_START = 24;
```

Test if we have an AltBeacon

In order to find out if the Bluetooth device we are seeing is an AltBeacon, we will test BEACON_CODE and see if that has the right value. Add this check into our reporting callback (`centralManager:didDiscoverPeripheral:… function`):

In `ViewController.m`

```
int code = (((bytes[BEACON_CODE_INDEX] << 8) & 0x0000ff00) |
              (bytes[BEACON_CODE_INDEX+1]     & 0x000000ff) );
if(code != BEACON_CODE_VALUE) return;
```

If BEACON_CODE has an invalid value, we will just stop processing that device further.

Parse and save/update data about a particular beacon

Now it is time to get proper values and display them in the list. But in order to do that, we need a `BeaconModel` object into which we will put those values. There are two cases:

- A beacon was already reported before, and we need just to update it.
- A beacon is reported for the first time, and we need to create a new BeaconModel, fill it and add to the list.

iOS will not return the Bluetooth MAC address, for security reasons. To identify a particular peripheral, therefore, you need to use an ID generated and uniquely assigned to the beacon by iOS itself, we will use this as an ID value inside our BeaconModel; we need to update the currentRssi and timestamp values every time; and we would like list view to be fresh with new values. So, we need to add below code snippet at the end of reporting callback (`centralManager:didDiscoverPeripheral:… function`)

In `ViewController.m`

```
NSString* beaconId = [peripheral.identifier UUIDString];
BeaconModel* beacon = nil;
for (BeaconModel* tmp in _beacons) {
    if (tmp != nil && [tmp.beaconId isEqualToString:beaconId]) {
        beacon = tmp;
        break;
    }
}
if (beacon == nil) {
    // reported for the first time - create new model
    beacon = [[BeaconModel alloc] init];
    beacon.beaconId = beaconId;
    beacon.referenceRssi = (bytes[REFERENCE_RSSI_START] & 0xff) - 256;
    beacon.arg1 = ((bytes[CONTENT_START] << 8)   & 0x0000ff00) |
                  ((bytes[CONTENT_START+1])      & 0x000000ff);
    beacon.arg2 = ((bytes[CONTENT_START+2] << 8) & 0x0000ff00) |
                  ((bytes[CONTENT_START+3])      & 0x000000ff);
    beacon.uuid = [self getBeaconUuidFromAdvertisement:bytes];
    [self.beacons addObject:beacon];
}

// update current RSSI field and timestamp
beacon.currentRssi = rssi.intValue;
beacon.timestamp = @([NSDate date].timeIntervalSince1970).longValue;

[self.tableView reloadData];
```

PS Don't worry about a compile error for getBeaconUuidFromAdvertisement (above). We will fix this in a moment.

Now, we need to add the implementation for the `getBeaconUuidFromAdvertisement:` function, which is a little bit too long to be included inline in the above code. Add this function to your view controller code:

In `ViewController.m`

```
- (NSString*) getBeaconUuidFromAdvertisement:(unsigned char*)adv {
    NSMutableString* val = [NSMutableString stringWithString:@""];
    for (int i = UUID_START, offset = 0; i <= UUID_STOP; ++i, ++offset) {
        [val appendFormat:@"%02x", (adv[i] & 0x000000ff)];
        if (offset == 3 || offset == 5 || offset == 7 || offset == 9) {
            [val appendString:@"-"];
        }
    }
    return [NSString stringWithString:val];
}
```

That is the final bit of code for our view controller.

Now, when you run the application, you should see at least one AltBeacon in the list, provided there is at least one in range of your iOS device.

Task 3 – Remove old beacons from the list

So far, we have an application that uses Bluetooth to scan for and list nearby AltBeacons. Great work on getting to this point!

For completeness, we should consider one more thing. If and when an AltBeacon moves out of range, the system will not report it again unless it comes back into device scanning range. But as things are, our model for that beacon will stay inside our list of beacons. Therefore, we need to implement a mechanism to verify if elements already on the list are still valid, and to clean the list of all 'dead' beacons.

To keep the list fresh, we will use the `timestamp` property of `BeaconModel` and an arbitrary timeout value. If we test an item and find its timestamp was more than X seconds ago, we will just remove that item from the list.

Add this `const` add the function below, `validateBeacons{}`, to ViewController at the bottom of `@implementation` in `ViewController.m`. This function does exactly what we just described: it iterates through all items and removes invalid ones.

In `ViewController.m`

```
static const long BEACON_DURATION = 8.5; // 8.5 [seconds]
- (BOOL) validateBeacons {
    BOOL anythingWasRemoved = NO;
    long earliestTimestampAllowed =
           @([NSDate date].timeIntervalSince1970).longValue -
BEACON_DURATION;

    NSMutableArray* newArray = [NSMutableArray
arrayWithCapacity:_beacons.count];

    for (BeaconModel* beacon in self.beacons) {
        if (beacon.timestamp >= earliestTimestampAllowed) {
            [newArray addObject:beacon];
        }
        else {
            anythingWasRemoved = YES;
        }
    }

    if (anythingWasRemoved) {
        [self.beacons setArray:newArray];
    }

    return anythingWasRemoved;
}
```

Now we need to make sure that this function is called at a specified interval, as long as the application is running. For this we will use an `NSTimer` scheduled to be repeated continuously.

First, let's add a wrapper function that can be fired by an `NSTimer` (put this at the bottom of `@implementation in ViewController.m`):

In `ViewController.m`

```
- (void) repeatableValidationTask:(NSTimer*)theTimer {
    if ([self validateBeacons]) {
        [self.tableView reloadData];
    }
}
```

Now we will schedule the timer to scan the beacons list periodically. First add a property to keep our timer:

In `ViewController.m`

```
@interface ViewController ()
...
@property (nonatomic, strong) NSTimer* timer;
...
@end
```

Now inside the `viewWillDisappear:` function make sure you invalidate the timer so it won't run in the background:

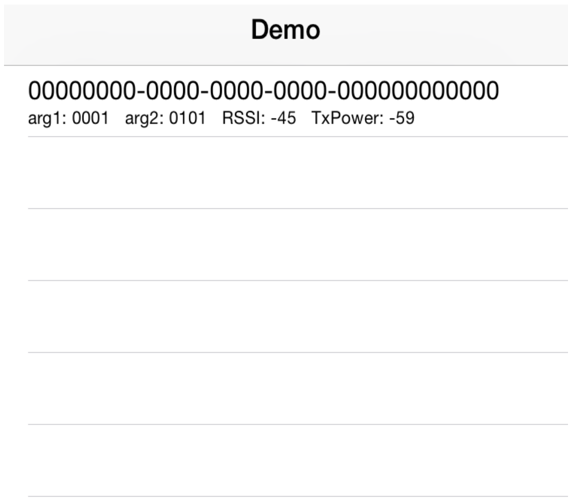
In `ViewController.m`

```
- (void) viewWillDisappear:(BOOL)animated {
    [self.timer invalidate];
    self.timer = nil;

    [self stopScanning];
}
```

And the last thing is to fire that timer inside the `viewWillAppear:` callback:

```
- (void) viewDidAppear:(BOOL)animated {
    [self startScanning];

    // fire it, every 4.3 seconds
    self.timer = [NSTimer scheduledTimerWithTimeInterval:4.3
                          target:self
                        selector:@selector(repeatableValidationTask:)
                        userInfo:nil
                         repeats:YES];
    [self.timer fire];
}
```

Now you can run and test your application on the device. All AltBeacons should be visible in the list. If you move out of range of any AltBeacon (or power down the Raspberry Pi), that item should disappear from the list within around 5 seconds (which was the timeout we chose). Come back in range and the beacon should reappear.



```
00000000-0000-0000-0000-000000000000
arg1: 0001   arg2: 0101   RSSI: -45   TxPower: -59
```

Final application showing one beacon around the device

Exercise Complete

Congratulations! You have created a working AltBeacon-ready app for iOS.

If you have any problem on the implementation of this iOS App, we provide completed source code in the zip file you downloaded, they are:

> *`ViewController.h`*
>
> *`ViewController.m`*
>
> *`BeaconModel.h`*
>
> *`BeaconModel.m`*

You can review the completed source code and track what the issue on your project.

Good luck!