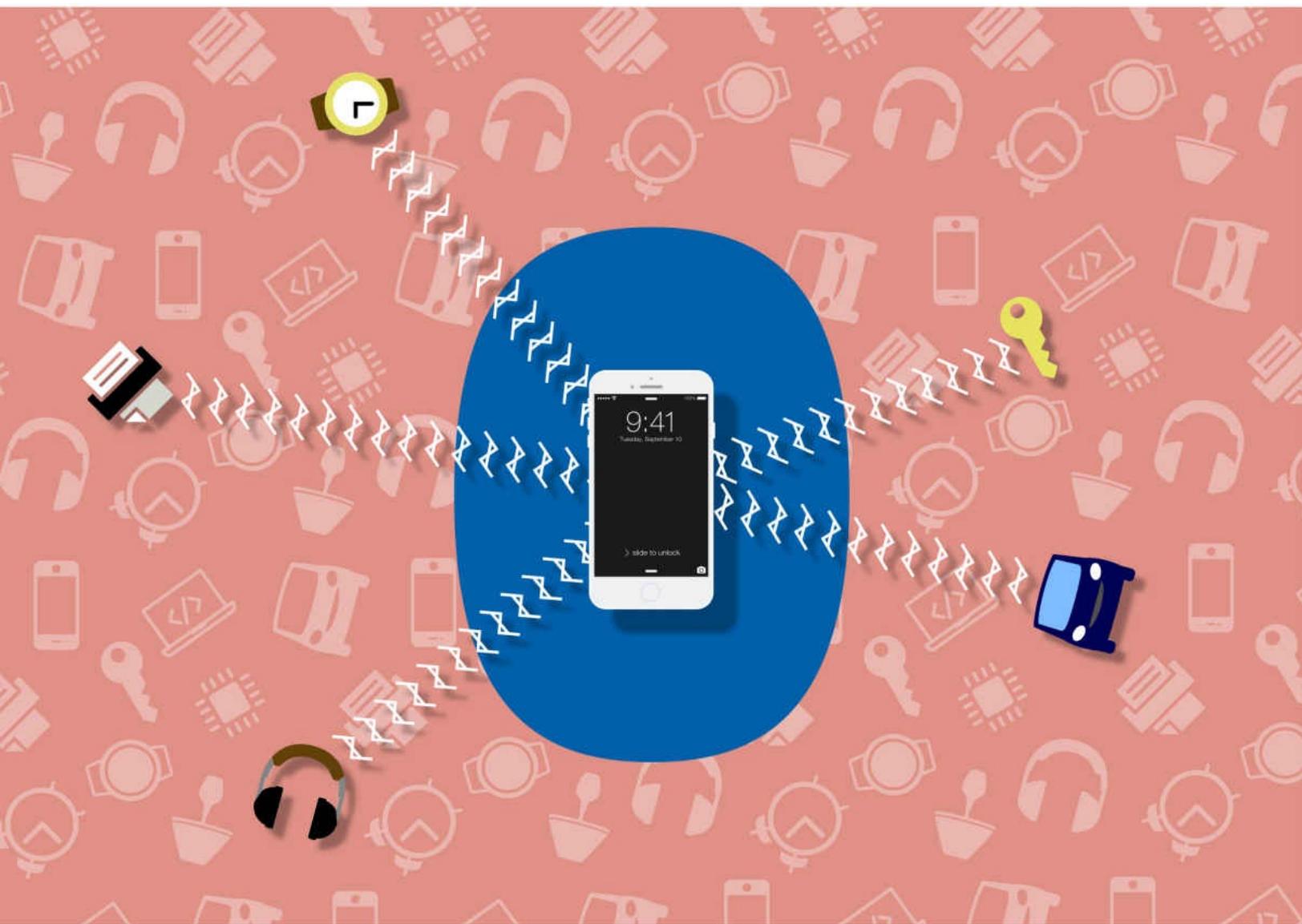


PROGRAM YOUR IPHONE TO BE A  
**SENSOR** ● **REMOTE CONTROL** ● **BEACON** ● **TRANSMITTER**  
USING

# Bluetooth Low Energy

IN IOS SWIFT



**Tony Gaitatzis**  
2017

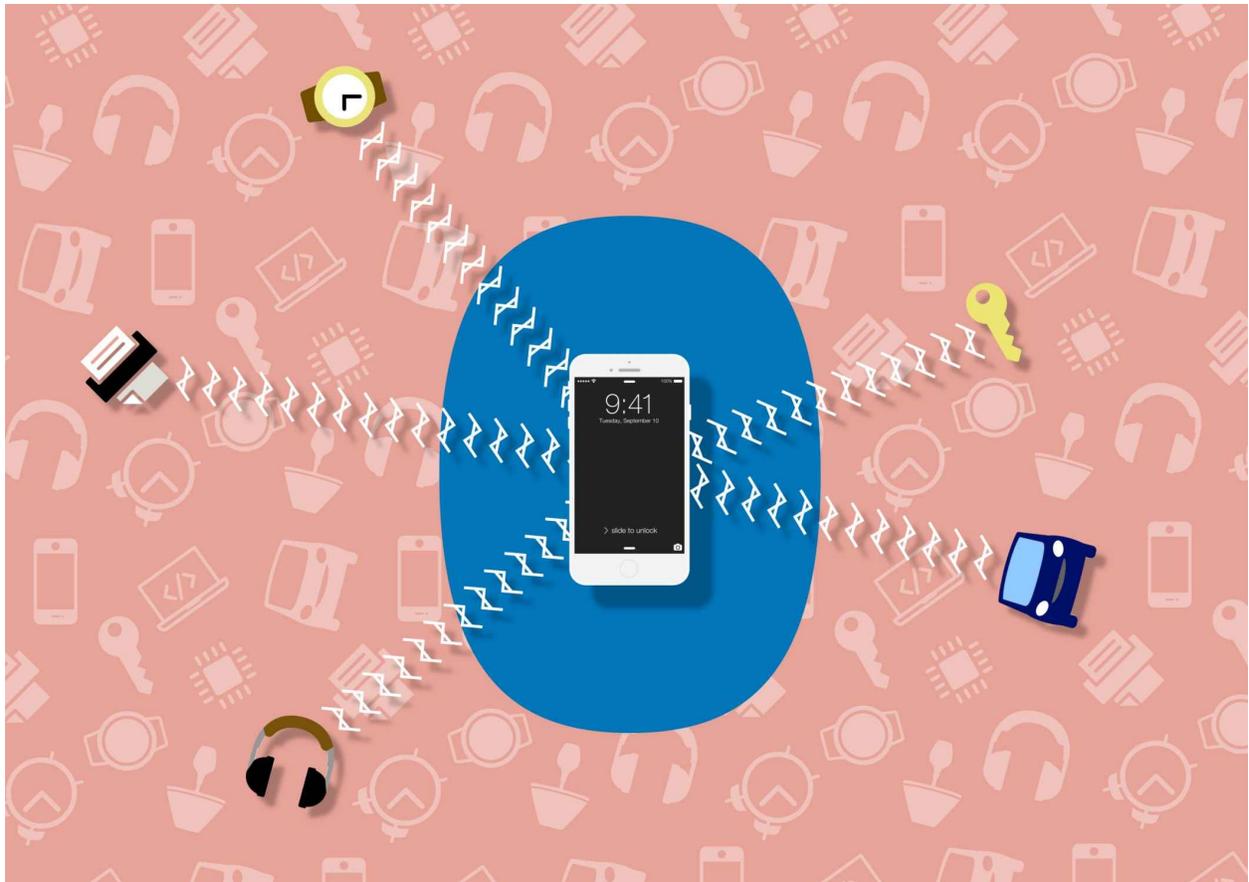
TURN YOUR IPHONE INTO A

**SENSOR - REMOTE CONTROL - IBEACON -  
TRANSMITTER**

USING

# **Bluetooth Low Energy**

IN IOS SWIFT



**Tony Gaitatzis**

2017

1st Edition

Tony Gaitatzis

BackupBrain Publishing, 2017

ISBN: 978-1-7751280-0-7

[backupbrain.co](http://backupbrain.co)



by Tony Gaitatzis  
Copyright © 2015 All Rights Reserved

All rights reserved. This book or any portion thereof may not be reproduced or used in any manner whatsoever without the express written permission of the publisher except for the use of brief quotations in a book review. For permission requests, write to the publisher, addressed “Bluetooth iOS Book Reprint Request,” at the address below.

[backupbrain@gmail.com](mailto:backupbrain@gmail.com)

This book contains code samples available under the MIT License, printed below:

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

ii

(This page intentionally left blank)



## **Dedication**

*To Linda for teaching me that sharing is caring,  
and Bruno for being much more thorough than I could ever be.*



(This page intentionally left blank)



# Preface

Thank you for buying this book. I'm excited to have written it and more excited that you are reading it.

I started with Bluetooth Low Energy in 2011 while making portable brain imaging technology. Later, while working on a friend's wearable electronics startup, I ended up working behind the scenes on the TV show America's Greatest Makers in the Spring of 2016.

Coming from a web programming background, I found the mechanics and nomenclature of BLE confusing and cryptic. After immersing myself in it for a period of time I acclimated to the differences and began to appreciate the power behind this low-power technology.

Unlike other wireless technologies, BLE can be powered from a coin cell battery for months at a time - perfect for a wearable or Internet of Things (IoT) project! Because of its low power and short data transmissions, it is great for transmitting bite size information, but not great for streaming data such as sound or video.

Good luck and enjoy!



## Conventions Used in This Book

Every developer has their own coding conventions. I personally believe that well-written code is self-explanatory. Moreover, consistent and organized coding conventions let developers step into each other's code much more easily, enabling them to reliably predict how the author has likely organized and implemented a feature, thereby making it easier to learn, collaborate, fix bugs and perform upgrades.

The coding conventions I used in this book is as follows:

Inline comments are as follows:

```
// inline comments
```

Multiline comments follow the Doxygen standard:

```
/**
```

```
This is a multiline comment
```

```
It features more than one line of comment
```

- Parameters:

- parameterOne: A description of what is expected for the first parameter \*/

Constants and variables are written in camel case:

```
let constantName = "Constant"
```

```
var normalVariable:[String]!
```



Function declarations are in Camel Case. In cases where there is not enough space for the whole function, parameters are written on another line:

```
func shortFunction() {  
}  
func longerFunction(value: String) {  
}  
func superLongFunctionName(  
value: String, parameterOne,  
parameterTwo: String) {  
...  
}
```



# Introduction

In this book you will learn the basics of how to program Central and Peripheral devices that communicate over Bluetooth Low Energy using iOS in Swift. These tutorials will culminate in three projects:

- A Beacon and Scanner
- A Echo Server and Client
- A Remote Controlled Device

Through the course of the book you will learn important concepts that relate to:

- How Bluetooth Low Energy works,
- How data is sent and received
- Common paradigms for handling data

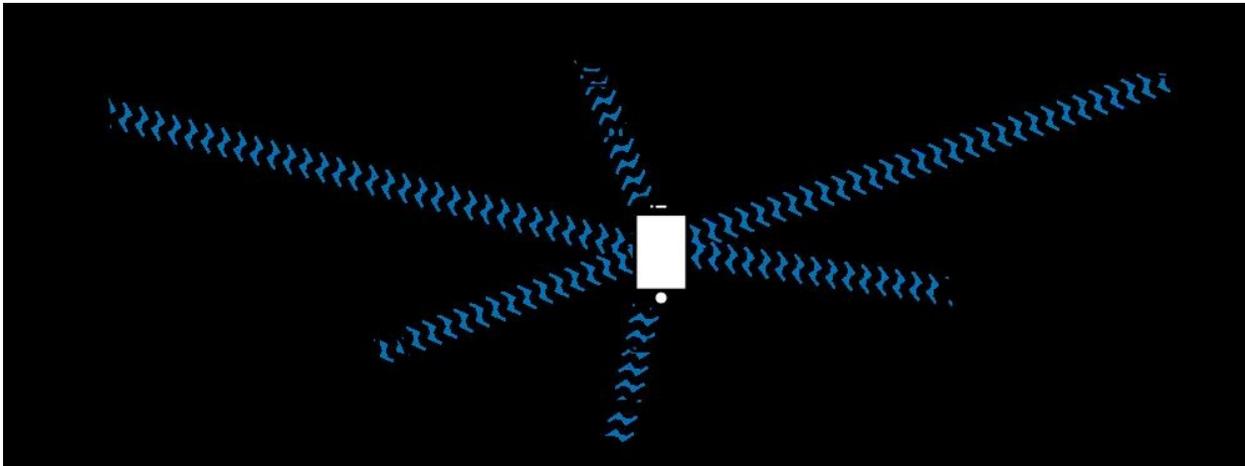
This book is an excellent read for anyone familiar with iOS programming, who wants to build an Internet of Things device or a tool that communicates with a Bluetooth device.

## Overview

Bluetooth Low Energy (BLE) is a digital radio protocol. Very simply, it works by transmitting radio signals from one computer to another.

Bluetooth supports a hub-and-spoke model of connectivity. One device acts as a hub, or “Central” in Bluetooth terminology. Other devices act as “Peripherals.”

A Central may hold several simultaneous connections with a number of peripherals, but a peripheral may only hold one connection at a time (Figure 1-1). Hence the names Central and Peripheral.



**Figure 1-1. Bluetooth network topology**

For example, your smartphone acts as a Central. It may connect to a Bluetooth speaker, lamp, smartwatch, and fitness tracker. Your fitness tracker and speaker, both Peripherals, can only be connected to one smartphone at a time.

The Central has two modes: scanning and connected. The Peripheral has two modes: advertising and connected. The Peripheral must be advertising for the Central to see it.

# **Advertising**

A Peripheral advertises by advertising its device name and other information on one radio frequency, then on another in a process known as frequency hopping. In doing so, it reduces radio interference created from reflected signals or other devices.

## Scanning

Similarly, the Central listens for a server's advertisement first on one radio frequency, then on another until it discovers an advertisement from a Peripheral. The process is not unlike that of trying to find a good show to watch on TV.

The time between radio frequency hops of the scanning Central happens at a different speed than the frequency hops of the advertising Peripheral. That way the scan and advertisement will eventually overlap so that the two can connect.

Each device has a unique media access control address (MAC address) that identifies it on the network. Peripherals advertise this MAC address along with other information about the Peripheral's settings.

## **Connecting**

A Central may connect to a Peripheral after the Central has seen the Peripheral's advertisement. The connection involves some kind of handshaking which is handled by the devices at the hardware or firmware level.

While connected, the Peripheral may not connect to any other device.

## **Disconnecting**

A Central may disconnect from a Peripheral at any time. The Peripheral is aware of the disconnection.

## **Communication**

A Central may send and request data to a Peripheral through something called a “Characteristic.” Characteristics are provided by the Peripheral for the Central to access. A Characteristic may have one or more properties, for example READ or WRITE. Each Characteristic belongs to a Service, which is like a container for Characteristics. This paradigm is called the Bluetooth Generic Attribute Profile (GATT).

The GATT paradigm is laid out as follows (Figure 1-2).

## Profile

### Service

Characteristic

Characteristic

Characteristic

### Service

Characteristic

Characteristic

Characteristic

## **Figure 1-2. Example GATT Structure**

To transmit or request data from a Characteristic, a Central must first connect to the Characteristic's Service.

For example, a heart rate monitor might have the following GATT profile, allowing a Central to read the beats per minute, name, and battery life of the server (Figure 1-3).

## Example Heartrate Monitor

### Device Info

Name

Manufacturer

Battery Life

### Heartrate Data

Beats Per Minute

Highest BPM

Lowest BPM

### **Figure 1-3. Example GATT structure for a heart monitor**

In order to retrieve the battery life of the Characteristic, the Central must be connected also to the Peripheral's "Device Info" Service.

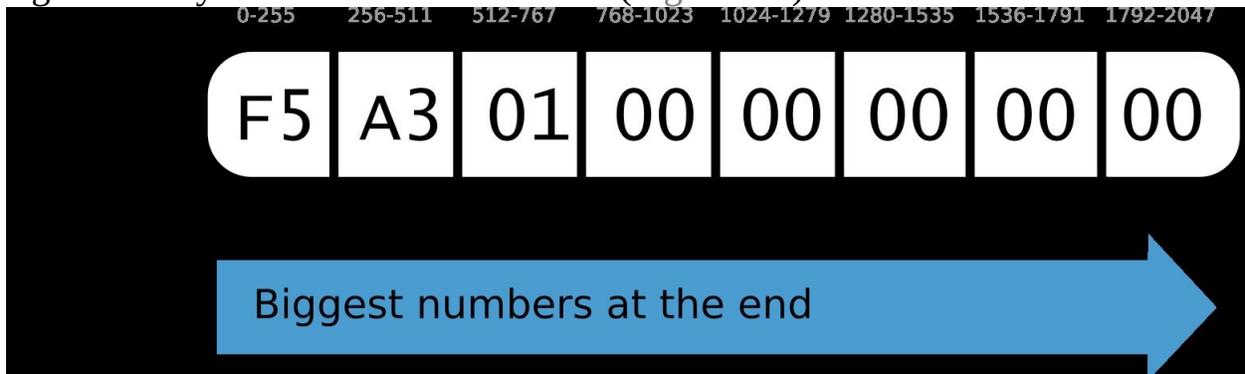
Because a Characteristic is provided by a Peripheral, the terminology refers to what can be done to the Characteristic. A "write" occurs when data is sent to the Characteristic and a "read" occurs when data is downloaded from the Characteristic.

To reiterate, a Characteristic is a field that can be written to or read from. A Service is a container that may hold one or more Characteristics. GATT is the layout of these Services and Characteristics. Characteristic can be written to or read from.

## Byte Order

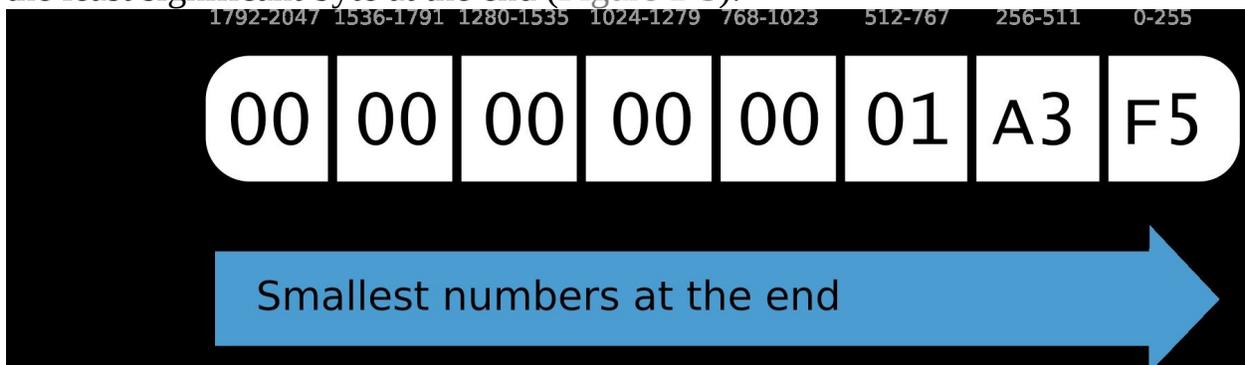
Bluetooth orders data in both Big-Endian and Little-Endian depending on the context.

During advertisement, data is transmitted in Big Endian, with the most significant bytes of a number at the end (Figure 1-4).



**Figure 1-4. Big Endian byte order**

Data transfers inside the GATT however are transmitted in Little Endian, with the least significant byte at the end (Figure 1-5).



**Figure 1-5. Little Endian byte order**

## **Permissions**

A Characteristic grants certain Permissions of the Central. These permissions include the ability to read and write data on the Characteristic, and to subscribe to Notifications.

## **Descriptors**

Descriptors describe the configuration of a Characteristic. The only one that has been specified so far is the “Notification” flag, which lets a Central subscribe to Notifications.

## UUIDs

A UUID, or Universally Unique Identifier is a very long identifier that is likely to be unique, no matter when the UUID was created or who created it.

BLE uses UUIDs to label Services and Characteristics so that Services and Characteristics can be identified accurately even when switching devices or when several Characteristics share the same name.

For example, if a Peripheral has two “Temperature” Characteristics - one for Celsius and the other in Fahrenheit, UUIDs allow for the right data to be communicated.

UUIDs are usually 128-bit strings and look like this:

ca06ea56-9f42-4fc3-8b75-e31212c97123

But since BLE has very limited data transmission, 16-bit UUIDs are also supported and can look like this:

0x1815

Each Characteristic and each Service is identified by its own UUID. Certain UUIDs are reserved for specific purposes.

For example, UUID 0x180F is reserved for Services that contain battery reporting Characteristics.

Similarly, Characteristics have reserved UUIDs in the Bluetooth Specification.

For example, UUID 0x2A19 is reserved for Characteristics that report battery levels.

A list of UUIDs reserved for specific Services can be found in **Appendix IV: Reserved GATT Services**.

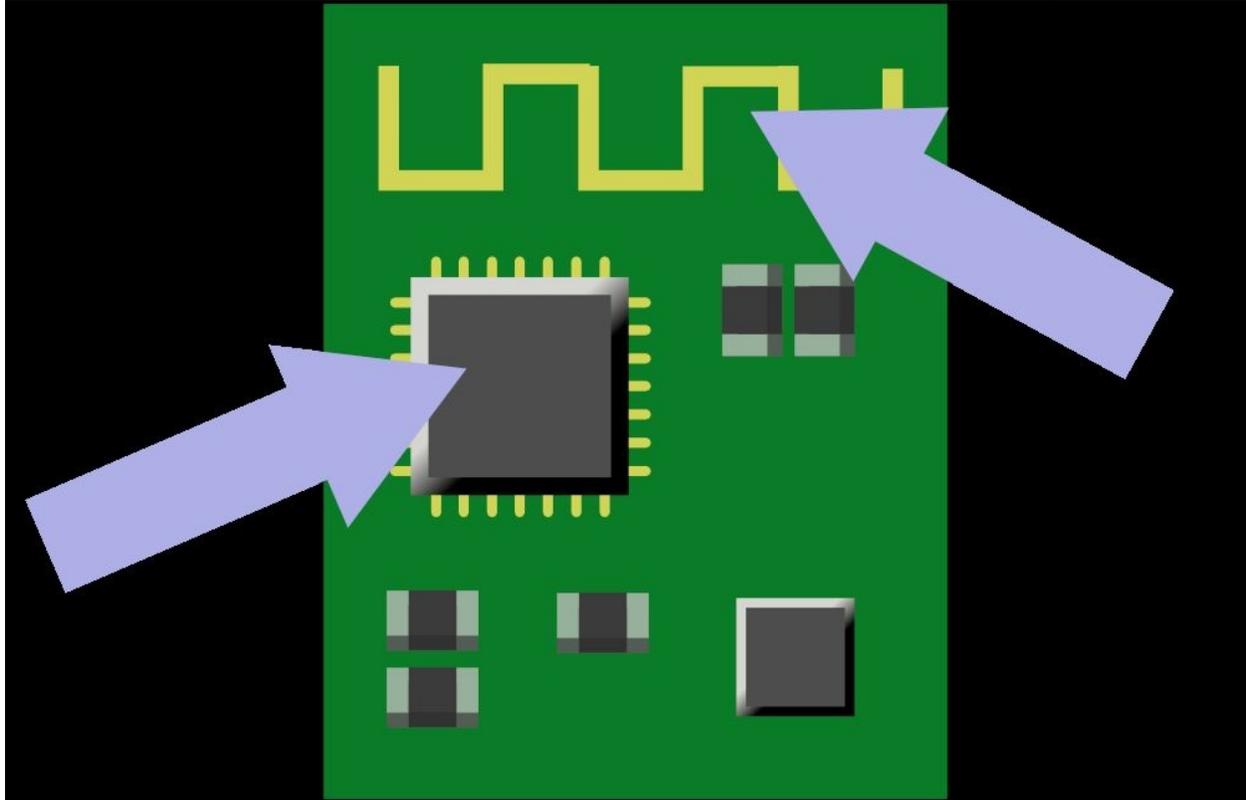
A list of UUIDs reserved for specific Characteristics can be in **Appendix V: Reserved GATT Characteristics**.

If you are unsure what UUIDs to use for a project, you are safe to choose an unassigned service (e.g. 0x180C) for a Service and generic Characteristic (0x2A56).

Although the possibility of two generated UUIDs being the same are extremely low, programmers are free to arbitrarily define UUIDs which may already exist. So long as the UUIDs defining the Services and Characteristics do not overlap in the a single GATT Profile, there is no issue in using UUIDs that exist in other contexts.

## Bluetooth Hardware

All Bluetooth devices feature at least a processor and an antenna (Figure 1-6).



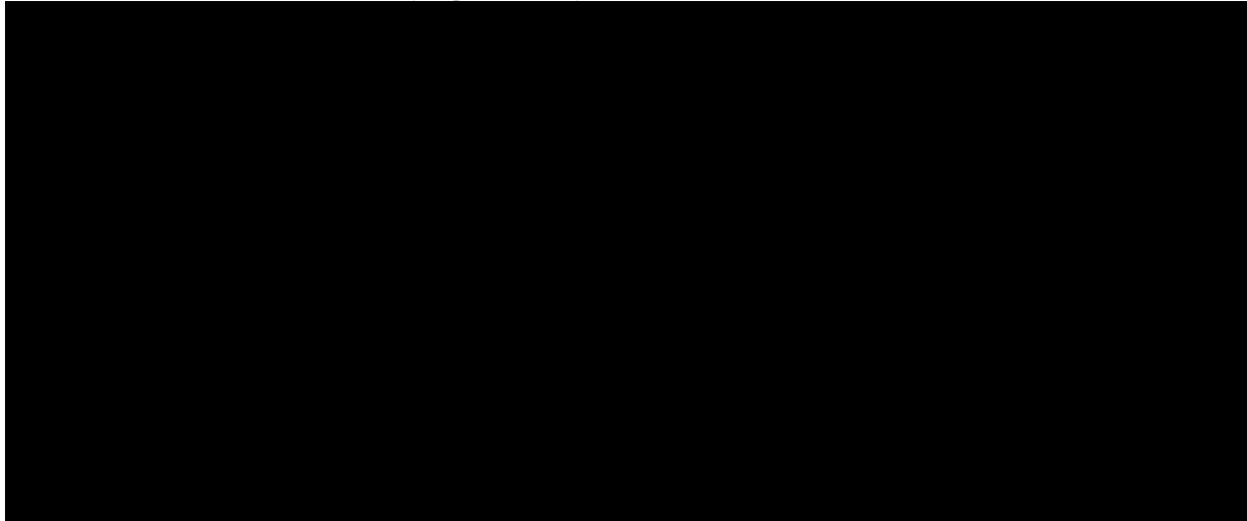
**Figure 1-6. Parts of a Bluetooth device**

The antenna transmits and receives radio signals. The processor responds to changes from the antenna and controls the antenna's tuning, the advertisement message, scanning, and data transmission of the BLE device.

## Power and Range

BLE has 20x2 Mhz channels, with a maximum 10 mW transmission power, 20 byte packet size, and 1 Mbit/s speed.

As with any radio signal, the quality of the signal drops dramatically with distance, as shown below (Figure 1-7).



**Figure 1-7. Distance versus Bluetooth Signal Strength**

This signal quality is correlated the Received Signal Strength Indicator (RSSI). If the RSSI is known when the Peripheral and Central are 1 meter apart ( $A$ ), as well as the RSSI at the current distance ( $R$ ) and the radio propagation constant ( $n$ ). The distance between the Central and the Peripheral in meters ( $d$ ) can be approximated with this equation:

$$d \approx 10^{A - R/10n}$$

The radio propagation constant depends on the environment, but it is typically somewhere between 2.7 in a poor environment and 4.3 in an ideal environment. Take for example a device with an RSSI of 75 at one meter, a current RSSI reading 35, with a propagation constant of 3.5:

$d$

$\approx$

10

$75 - 35/10 \times 3.5$

$d$

$\approx$

10

40/35

$d \approx 14$

Therefore the distance between the Peripheral and Central is approximately 14 meters.

# Introducing iOS

iOS is an incredibly easy platform to program Bluetooth Low Energy. Apple has done most of the work necessary to get Bluetooth Low Energy projects off the ground.

Apple makes it easy for anyone with an Apple computer to get into iOS programming. Xcode is a dream to work with, there are no developer registration costs, and the Swift programming language is easy to use.

That means developers can develop and test apps rapidly. iPhones and iPads, as with all modern mobile devices, are designed to support Bluetooth Low Energy.

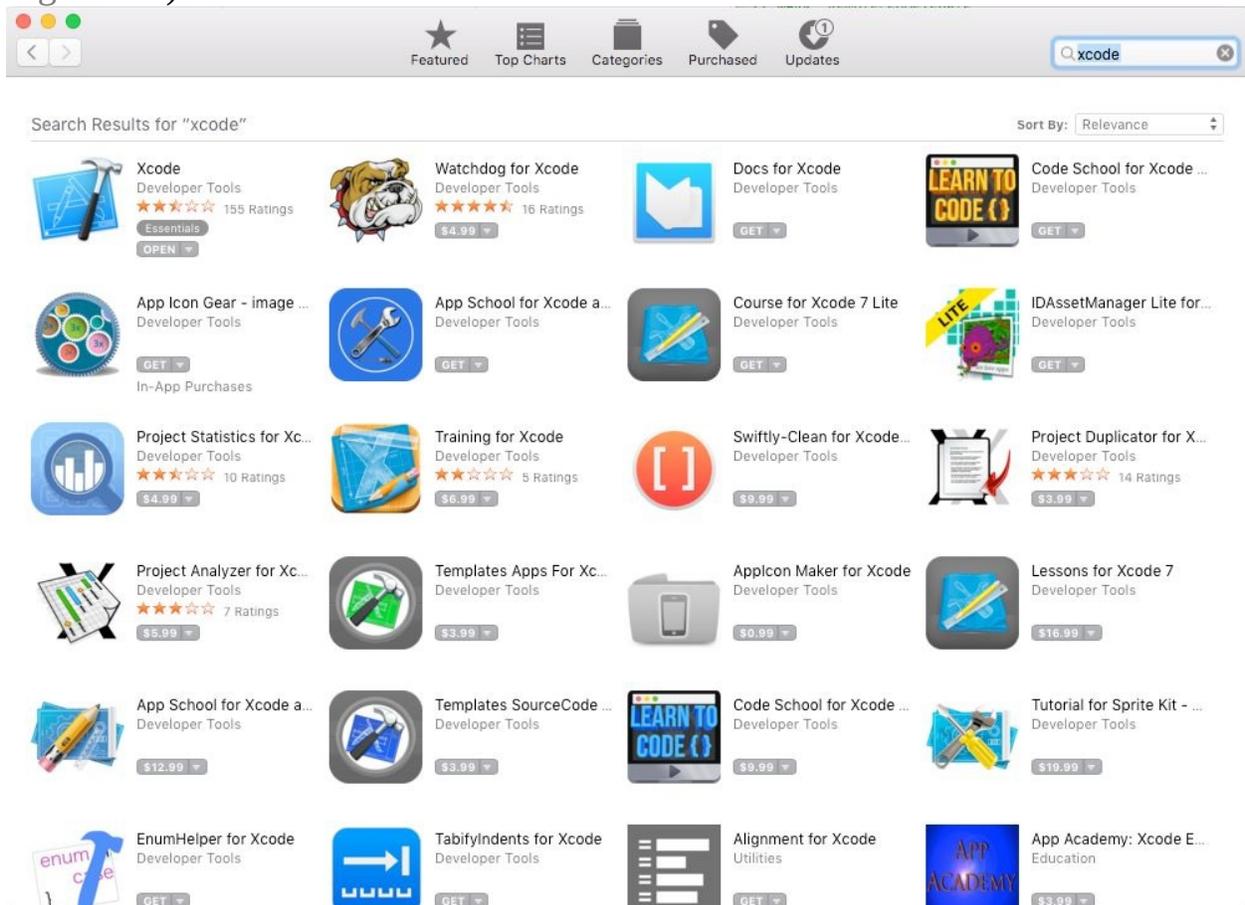
This book teaches how to make Bluetooth Low Energy (BLE) capable Apps using Swift for iOS. Although the examples in this book are relatively simple, the app potential of this technology is amazing.

# Xcode Setup

iPhones since version 5 and iPads since version 2 are designed to support Bluetooth Low Energy.

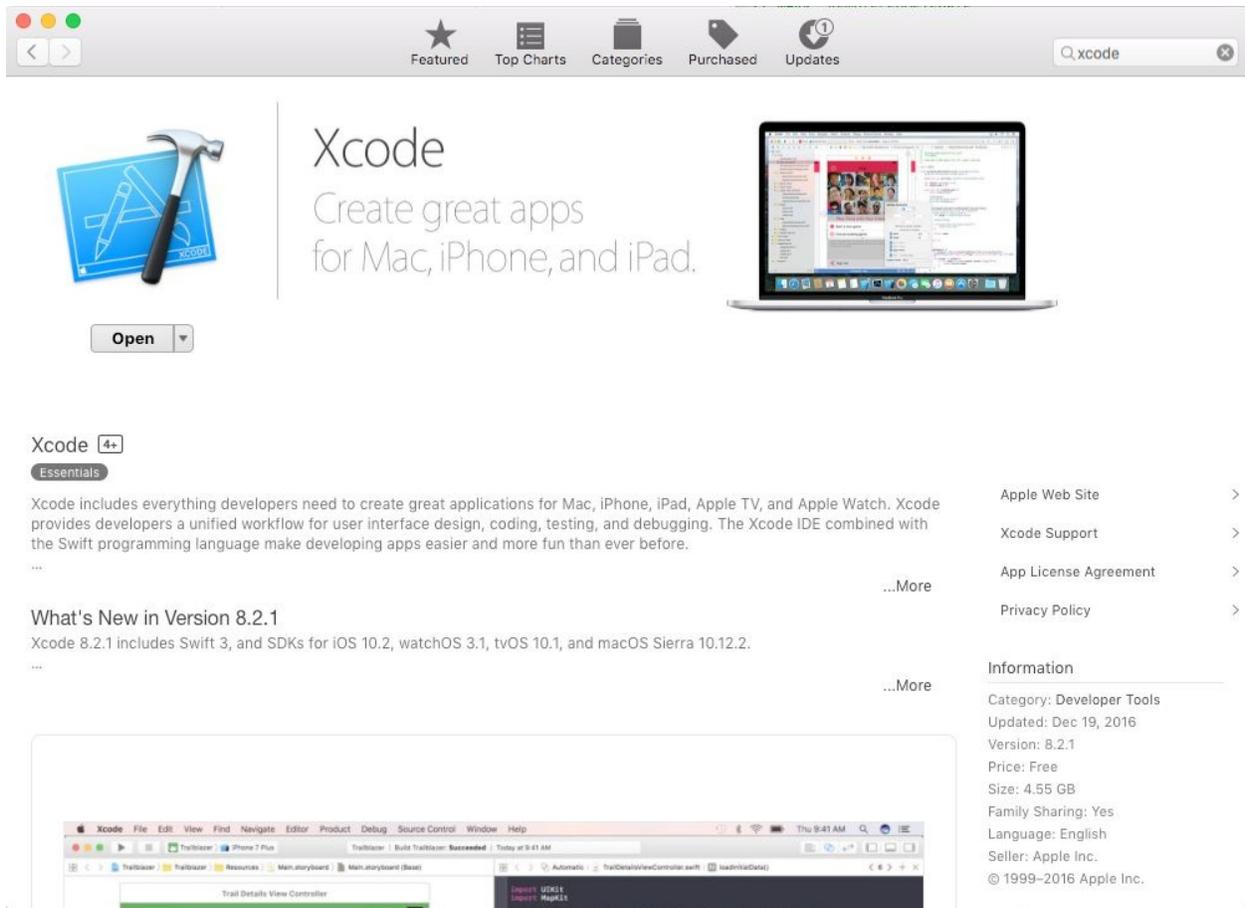
We will be using XCode to learn how to program Bluetooth Low Energy software with iOS. Although the examples in this book are relatively simple, the app potential of this technology is amazing. To program in iOS, you will need XCode on a Mac computer.

XCode can be downloaded for free from the App Store. Search for XCode in the App Store (Figure 2-1).



**Figure 2-1. XCode listed in the App Store**

Install XCode by selecting Xcode from the list and clicking the "Install" button (Figure 2-2).



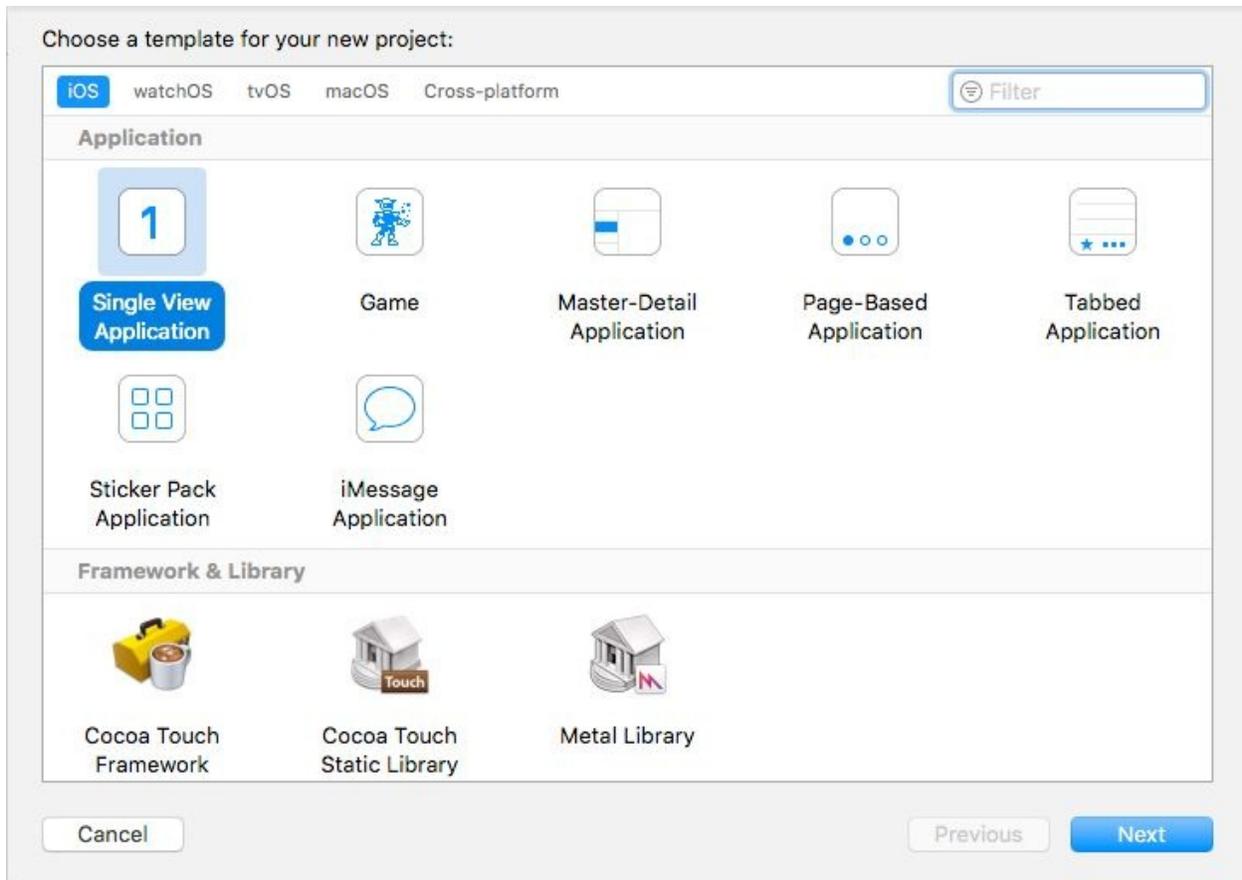
**Figure 2-2. XCode detail page**

Running XCode will open a screen like this. Select "Create a new Xcode project" to continue (Figure 2-3).



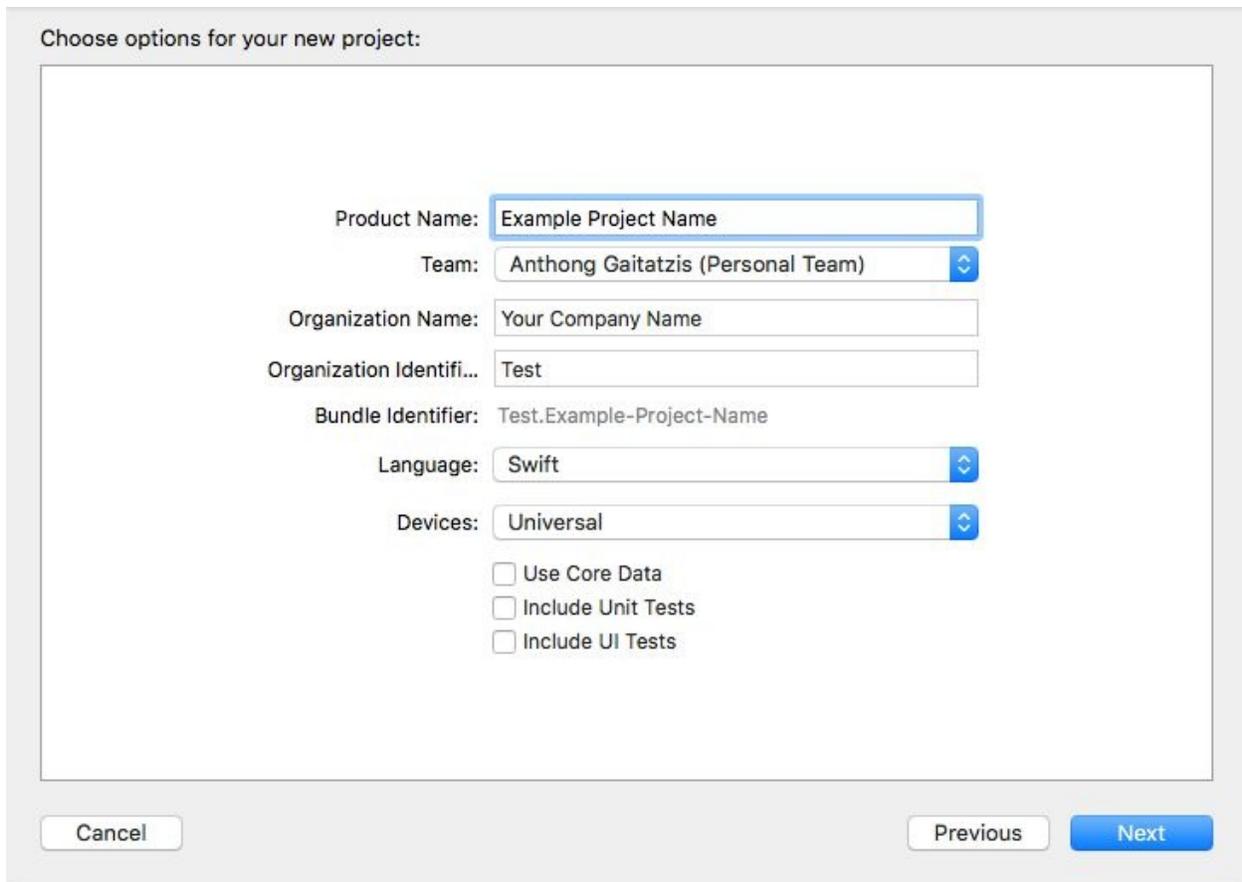
**Figure 2-3. XCode Launch Screen**

Each time a new project is created, the project type must be specified. In this book, "Single View Application" will be used for all projects (Figure 2-4).



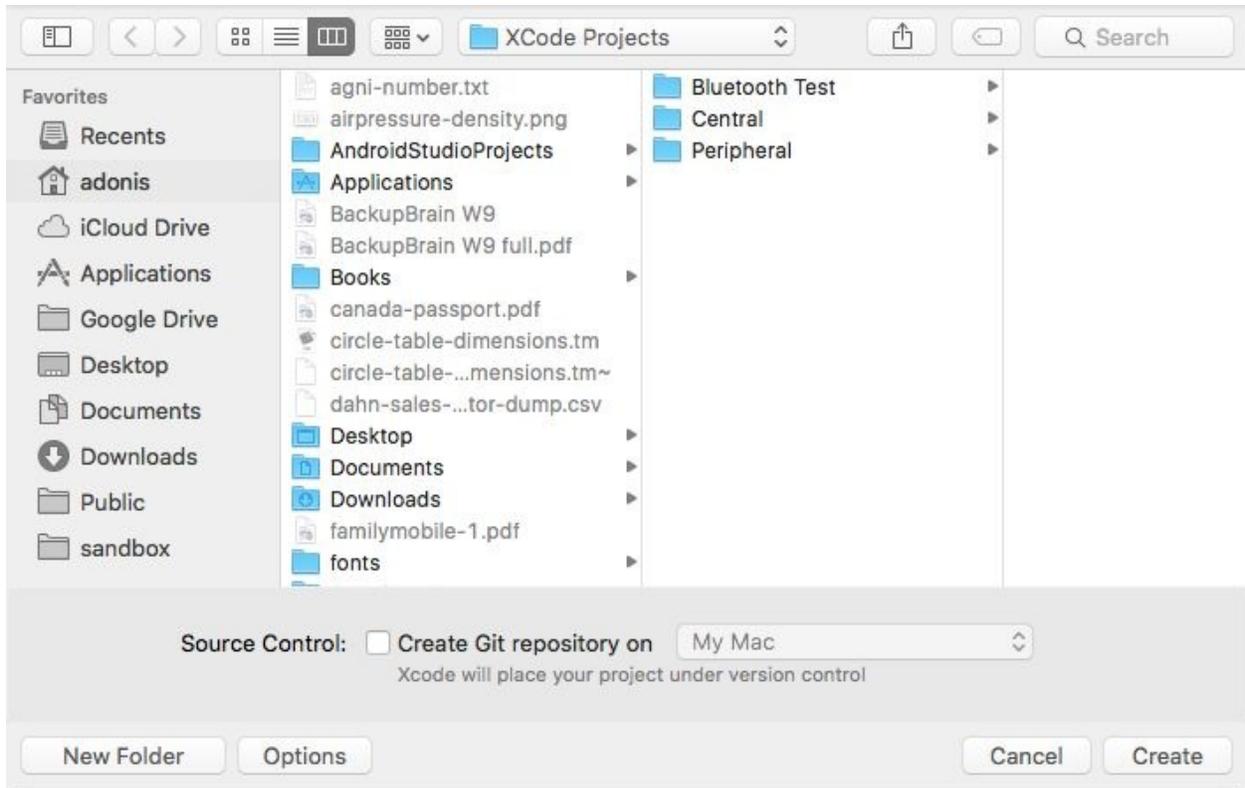
**Figure 2-4. New XCode Project Type**

From there, a Product Name, Team and Organization Name must be defined. These names are arbitrary. Names for each chapter project will be suggested (Figure 2-5).



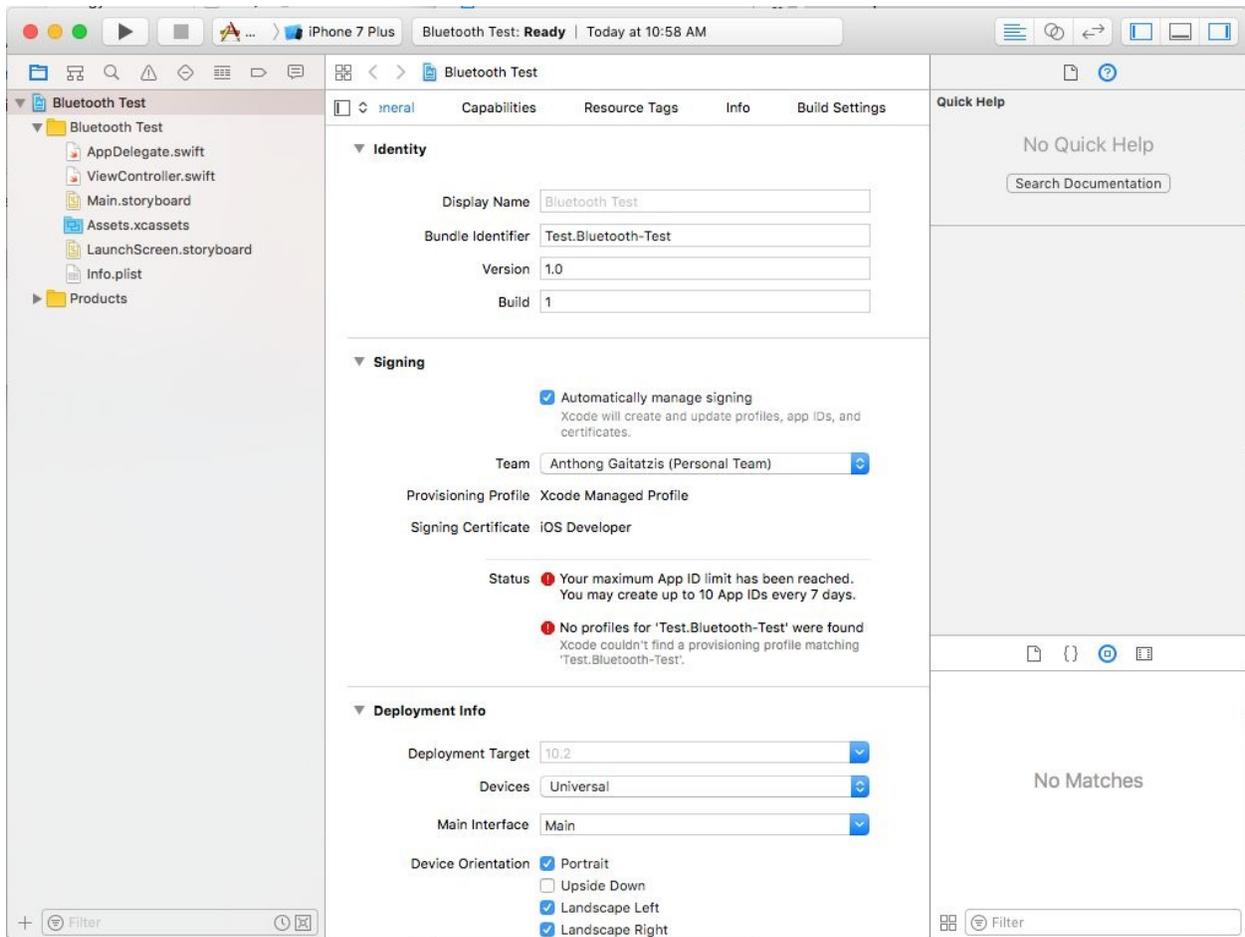
### Figure 2-5. New XCode Project Name

Click "Next" and XCode will present a "Save As" modal dialog. Select which folder to save the new project and click "Create" to save the project (Figure 2-6).



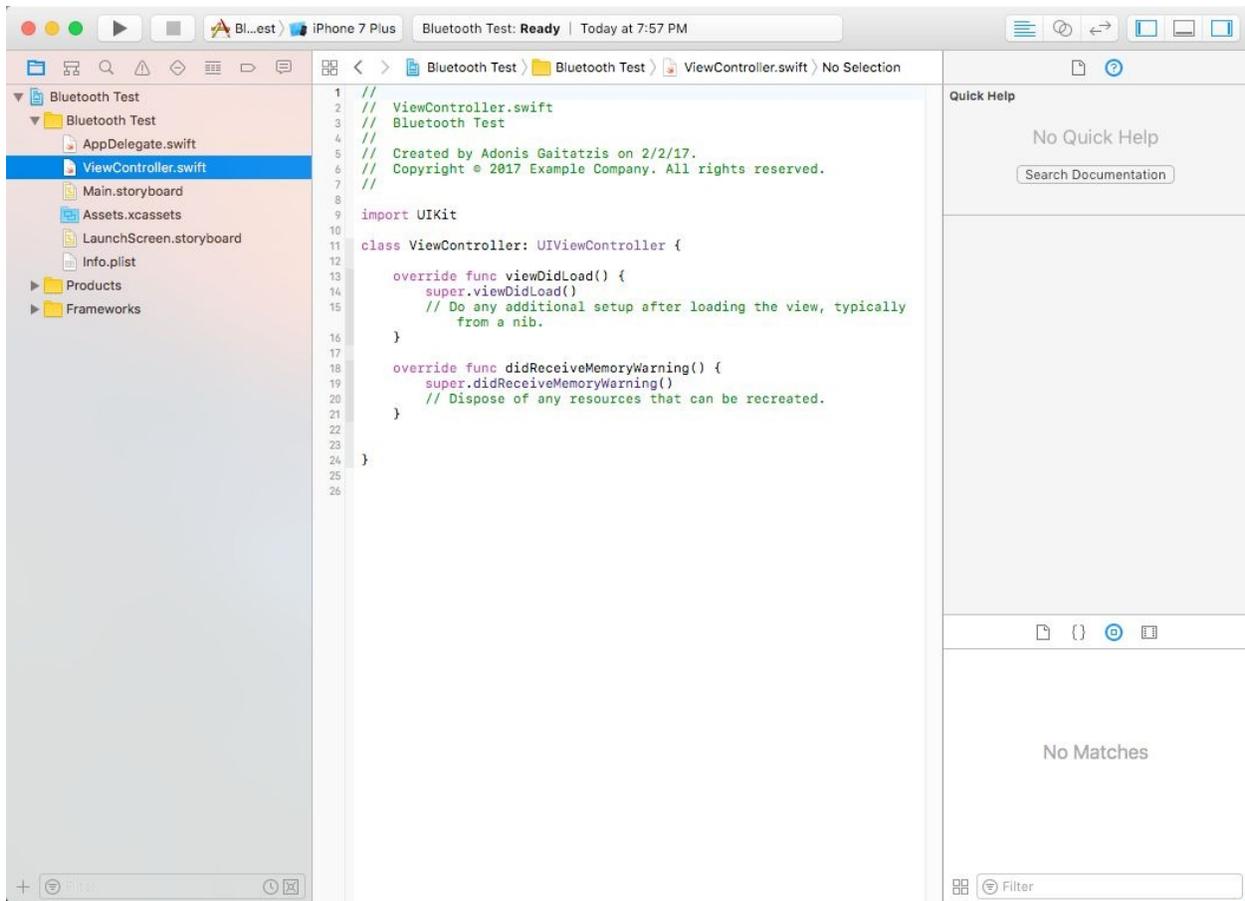
**Figure 2-6. New XCode Project Folder**

The next screen is the project settings, where the Project Name and Team can be changed. On the left is the project structure, where new classes and groups can be created (Figure 2-7).



**Figure 2-7. New XCode Project**

The center panel is where code and storyboards are edited (Figure 2-8).



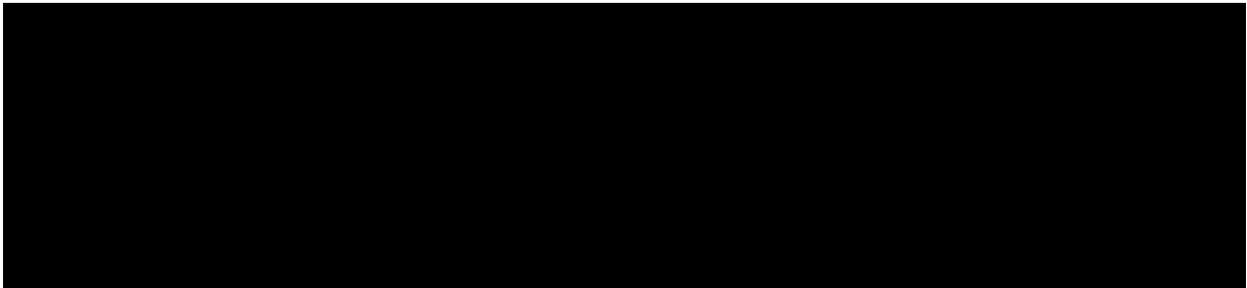
**Figure 2-8. XCode code editor**

## Bootstrapping

The first thing to do in any software project is to become familiar with the environment.

Because we are working with Bluetooth, it's important to learn how to initialize the Bluetooth radio and report what the program is doing.

Both the Central and Peripheral talk to the computer over USB when being programmed. That allows you to report errors and status messages to the computer when the programs are running (Figure 3-1).



**Figure 3-1. Programming configuration**

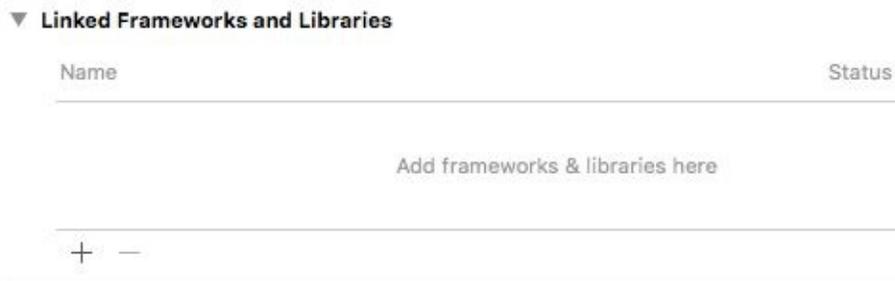
# Programming the Central

This chapter details how to create a Central App that turns the Bluetooth radio on. The Bluetooth radio requires the CoreBluetooth Framework and might be off by default.

## Adding Bluetooth Support

Since most Apps don't require Bluetooth, and the APIs take up valuable program space, the APIs are not included by default. To add support for Bluetooth, the CoreBluetooth Framework must be included.

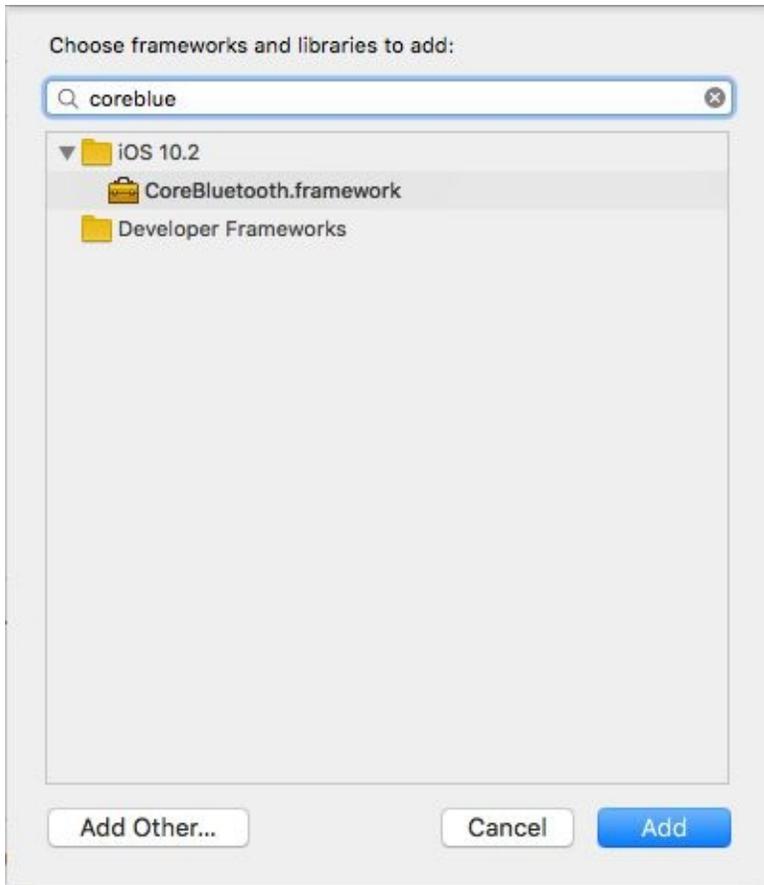
This is done by scrolling to the bottom of the Project Settings Screen, to the "Linked Frameworks and Libraries" section (Figure 3-2).



**Figure 3-2.**

### Linked Framework List

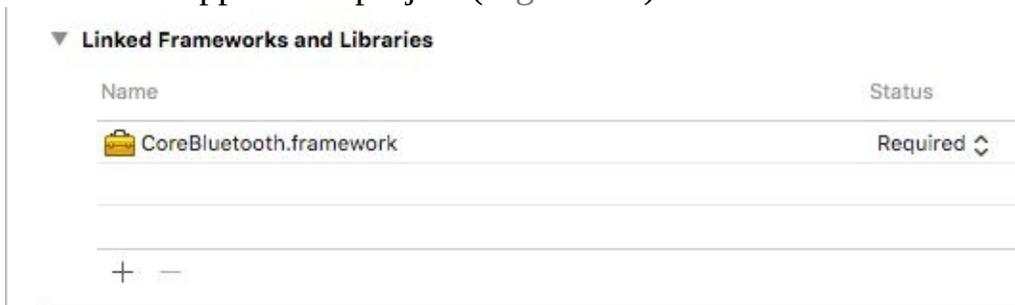
Click the "+" button to add a new Framework. A dialog will pop up. Search for "CoreBluetooth" in the search field (Figure 3-3):



**Figure 3-3. Linked**

### Framework List

Click on "CoreBluetooth.framework" and click the "Add" button to add Bluetooth support to a project (Figure 3-4).



**Figure 3-4.**

### Linked Framework List

### Enable Bluetooth

Before using any Bluetooth features, it is important to import the CoreBluetooth APIs at in the file header of any class that will use Bluetooth classes, and to turn on the Bluetooth radio

Importing the CoreBluetooth APIs is done like this:  
`import CoreBluetooth`

The user might turn the Bluetooth radio off any time. Therefore, every time the App loads, it needs to check if Bluetooth is still enabled or has been disabled, using this function.

This is done by making a ViewController a CBCentralManagerDelegate and instantiating a CBCentralManager.

The CBCentralManager allows the iOS device to act as a Bluetooth Central, and the CBCentralManager relays CBCentralManager state changes and events to the local object.

It takes a moment for Bluetooth to turn on. To prevent trying to access Bluetooth before it's ready, the App must listen for the CentralManagerDelegate to respond with a `centralManagerDidUpdateState` method, which is triggered by changes in the Bluetooth radio status.

```
class ViewController: UIViewController, CBCentralManagerDelegate { var  
centralManager:CBCentralManager
```

```
override func viewDidLoad() {  
super.viewDidLoad()  
centralManager = CBCentralManager(delegate: self, queue: nil)  
}
```

```
func centralManagerDidUpdateState(_ central: CBCentralManager) {  
print("Central Manager updated: checking state") switch (central.state) {  
case .poweredOff:
```

```
print ("BLE Hardware is powered off")  
bluetoothStatusLabel.text = "Bluetooth Radio Off" case .poweredOn:  
print ("BLE Hardware powered on and ready")  
bluetoothStatusLabel.text = "Bluetooth Radio On" case .resetting:  
print ("BLE Hardware is resetting...")  
bluetoothStatusLabel.text = "Bluetooth Radio Resetting..." case .unauthorized:  
print ("BLE State is unauthorized")  
bluetoothStatusLabel.text = "Bluetooth Radio Unauthorized" case .unsupported:
```

```
print ("Ble hardware is unsupported on this device") bluetoothStatusLabel.text =
"Bluetooth Radio Unsupported" case .unknown:
print ("Ble state is unavailable")
bluetoothStatusLabel.text = "Bluetooth State Unknown" }
}
}
```

The `centralManagerDidUpdateState` method will include an updated `CBCentralManager` object, including the new `CBCentralManagerState` which tells the state of the Bluetooth radio and its capabilities. The new state will be one of the following:

### Table 3-1 . `CBManagerState`

#### State Description

**poweredOff** Bluetooth is disabled

**poweredOn** Bluetooth is enabled

**resetting** Bluetooth radio is resetting

#### **unauthorized**

App is not authorized to use Bluetooth

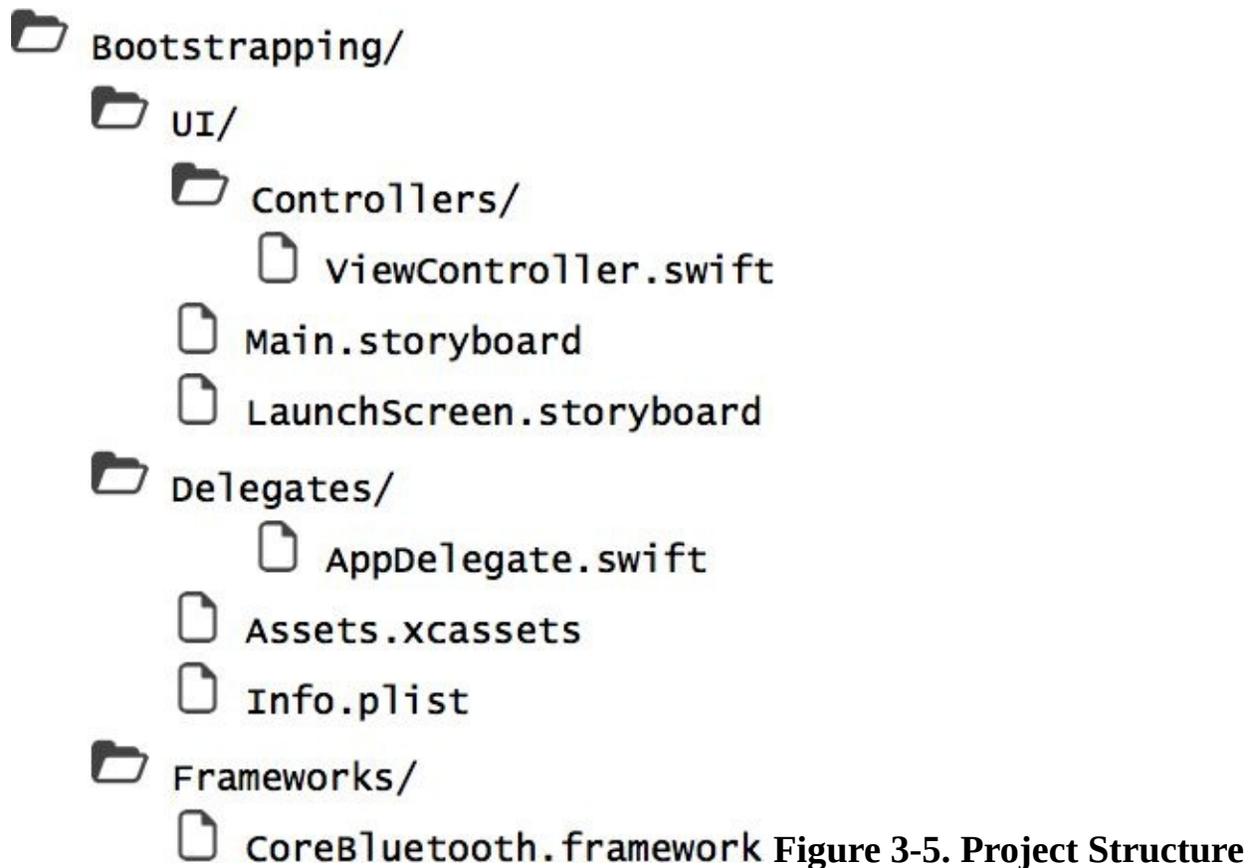
**unknown** There was a problem talking to the Bluetooth radio

**unsupported** Bluetooth is unavailable

### Putting It All Together

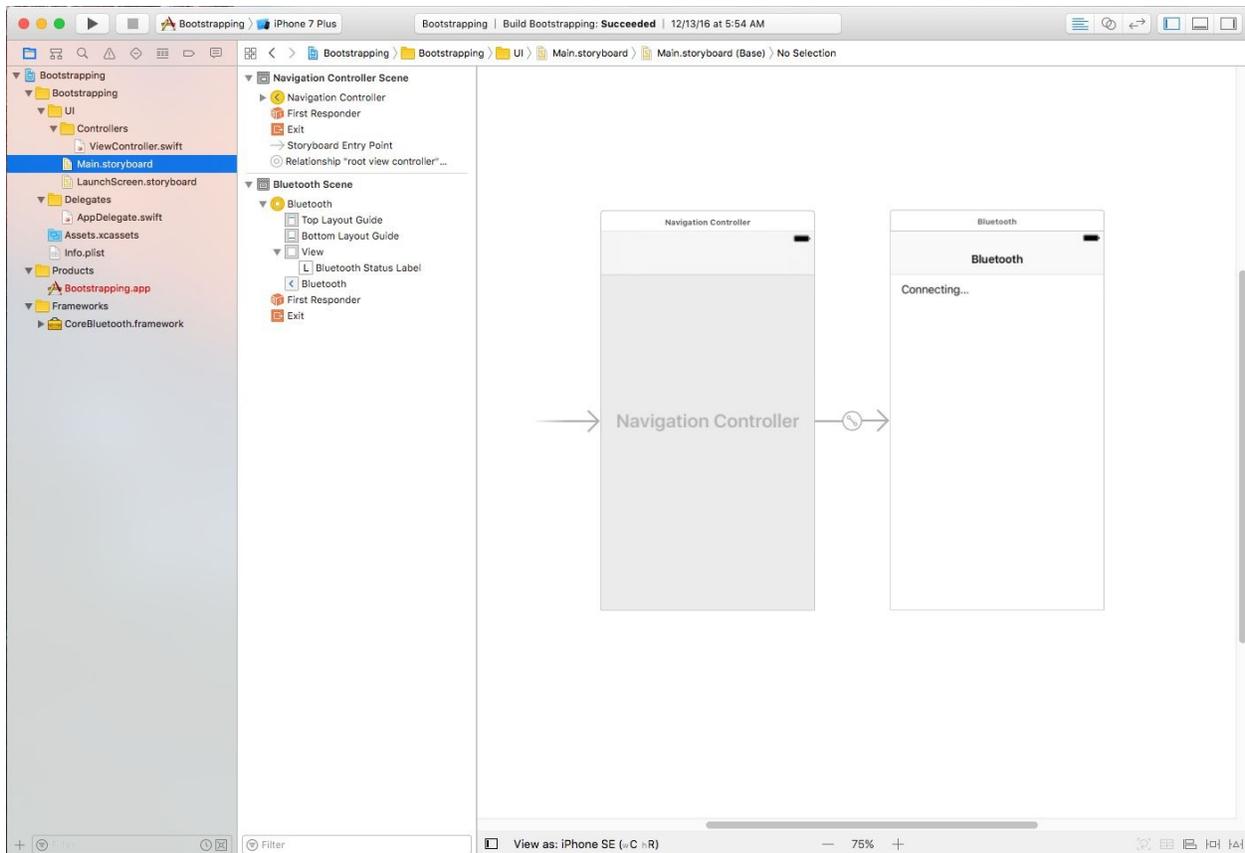
Create a new project called `Bootstrapping`. Create packages and classes so that the project structure resembles this:

Your code structure should now look like this (Figure 3-5).



## Storyboard

Create a UINavigationController and connect the Storyboard Entry Point to it. Add a UILabel to the UIViewController to be used to display the Bluetooth radio status (Figure 3-6):



**Figure 3-6. Project Storyboard Controllers**

The ViewController can create a CentralManager which is alerted when the device's Bluetooth radio is turned on or off.

Therefore, the ViewController is programmed like this (Example 3-1):

**Example 3-1. UI/Controllers/ViewController.swift**

```
import UIKit
import CoreBluetooth
/**
This view attempts to turn on the Bluetooth Radio
*/
class ViewController: UIViewController, CBCentralManagerDelegate {
// MARK: UI Elements
@IBOutlet weak var bluetoothStatusLabel: UILabel!
// MARK: Scan Properties
var centralManager: CBCentralManager!

/**
```

```
View loaded. Start Bluetooth radio. */
```

```
override func viewDidLoad() {  
    super.viewDidLoad()  
    print("Initializing central manager")  
    centralManager = CBCentralManager(delegate: self, queue: nil) }  
// MARK: CBCentralManagerDelegate Functions  
/**  
Bluetooth radio state changed
```

```
- Parameters:  
- central: the reference to the central  
*/
```

```
func centralManagerDidUpdateState(_ central: CBCentralManager) {  
    print("Central Manager updated: checking state")  
  
    switch (central.state) {  
    case .poweredOff:  
        print ("BLE Hardware is powered off")  
        bluetoothStatusLabel.text = "Bluetooth Radio Off" case .poweredOn:  
        print ("BLE Hardware powered on and ready")  
        bluetoothStatusLabel.text = "Bluetooth Radio On" case .resetting:  
        print ("BLE Hardware is resetting...")  
        bluetoothStatusLabel.text = "Bluetooth Radio Resetting..." case .unauthorized:  
        print ("BLE State is unauthorized")  
        bluetoothStatusLabel.text = "Bluetooth Radio Unauthorized" case .unsupported:  
        print ("Ble hardware is unsupported on this device")  
        bluetoothStatusLabel.text = "Bluetooth Radio Unsupported" case .unknown:  
        print ("Ble state is unavailable")  
        bluetoothStatusLabel.text = "Bluetooth State Unknown" }  
    }  
}
```

The resulting app will be able to turn the Bluetooth Radio on (Figure 3-7).

●●●○ Fido 3G

13:53

100%  ⚡

## Bluetooth

Bluetooth Radio Off

**Turn On Bluetooth to Allow  
"sketch" to Connect to  
Accessories**

Settings

OK

●●●○ Fido 3G

13:54

⌘ 100%  ⚡

## Bluetooth

---

Bluetooth Radio On

**Figure 3-7. Dialog to request user's permission to enable Bluetooth and Main App Screen**

# Programming the Peripheral

Peripheral Mode in iOS is supported since iOS 10. It is simple to use, but requires first that the CoreBluetooth Framework is included and imported into code, and that the Bluetooth radio is turned on

## Import CoreBluetooth

Link the CoreBluetooth Framework from the project Settings (Figure 3-8).



**Figure 3-8. CoreBluetooth Framework linked into project**

Import the CoreBluetooth library in the code header to access the Bluetooth APIs:

```
import CoreBluetooth
```

Enable Bluetooth

To turn on Bluetooth, instantiate a new CBPeripheralManager:

```
// empty dispatch queue
```

```
let dispatchQueue:DispatchQueue! = nil
```

```
let peripheralManager = \
```

```
CBPeripheralManager(delegate: self, queue: dispatchQueue)
```

The CBPeripheralManagerDelegate will execute the peripheralManagerDidUpdateState callback, which will alert the App when the Bluetooth radio state has changed:

```
class ViewController: UIViewController, CBPeripheralManagerDelegate { ...
```

```
func peripheralManagerDidUpdateState(  
  
_ peripheral: CBPeripheralManager)
```

```
{ switch (state) {  
case CBManagerState.poweredOn:
```

```
print("Bluetooth on")
```

```

case CBManagerState.poweredOff:
print("Bluetooth off")
case CBManagerState.resetting:
print("Bluetooth is resetting")
case CBManagerState.unauthorized:
print("App not authorized ot use Bluetooth") case CBManagerState.unknown:
print("Bluetooth off")
case CBManagerState.poweredOff:
print("Unknown problem when talking trying to start \ Bluetooth radio")
case CBManagerState.unsupported:
print("Bluetooth not supported")
}
}
...
}

```

It does this by passing a CBManagerState representing the new state of the Bluetooth radio, with the following possible states:

### **Table 3-2 . CBManagerState**

#### **State Description**

**poweredOff** Bluetooth is disabled

**poweredOn** Bluetooth is enabled

**resetting** Bluetooth radio is resetting

**unauthorized**

App is not authorized to use Bluetooth

**unknown** There was a problem talking to the Bluetooth radio

**unsupported** Bluetooth is unavailable

### **Putting It All Together**

Create a new project called Bootstrapping with the following project structure (Figure 3-9).



**Figure 3-9. Project**

### **Structure**

This project will be a single UIView app that creates a custom Bluetooth Peripheral.

### **Models**

The BlePeripheral object will create a custom Bluetooth Peripheral and its track events and state changes (Example 3-2).

### **Example 3-2. Models/BlePeripheral.swift**

```
import UIKit
import CoreBluetooth
class BlePeripheral : NSObject, CBPeripheralManagerDelegate { // MARK:
Peripheral properties
// Advertized name
let advertisingName = "MyDevice"
```

```

// MARK: Peripheral State

// Peripheral Manager
var peripheralManager:CBPeripheralManager! // Connected Central
var central:CBCentral!
// delegate
var delegate:BlePeripheralDelegate!

/**
Initialize BlePeripheral with a corresponding Peripheral

- Parameters:
- delegate: The BlePeripheralDelegate
- peripheral: The discovered Peripheral

*/
init(delegate: BlePeripheralDelegate?) { super.init()
// empty dispatch queue
let dispatchQueue:DispatchQueue! = nil

self.delegate = delegate
peripheralManager = \
CBPeripheralManager(delegate: self, queue: dispatchQueue) }
// MARK: CBPeripheralManagerDelegate

/**
Peripheral will become active */
func peripheralManager(

_ peripheral: CBPeripheralManager, willRestoreState dict: [String : Any])

{
print("restoring peripheral state")
}

/**
Bluetooth Radio state changed
*/

func peripheralManagerDidUpdateState( _ peripheral: CBPeripheralManager)

```

```

{
peripheralManager = peripheral
delegate?.blePeripheral?(stateChanged: peripheral.state)
} }

```

## Delegates

The BlePeripheralDelegate will relay important events from the BlePeripheral to the ViewController (Example 3-3).

### Example 3-3. Delegates/BlePeripheralDelegate.swift

```

import UIKit
import CoreBluetooth
@objc protocol BlePeripheralDelegate : class { /**
Bluetooth Radio state changed

```

```

- Parameters:
- state: new CBManagerState */

```

```

@objc optional func blePeripheral(stateChanged state: CBManagerState) }

```

## Controllers

The main UIView will instantiate a BlePeripheral object and print a message to the debugger when Bluetooth radio has turned on (Example 3-4).

### Example 3-4. UI/Controllers/ViewController.swift

```

import UIKit
import CoreBluetooth
class ViewController: UIViewController, BlePeripheralDelegate {
// MARK: BlePeripheral
// BlePeripheral
var blePeripheral:BlePeripheral!

/**
UIView loaded
*/

override func viewDidLoad() { super.viewDidLoad()
}

```

```

/**
View appeared. Start the Peripheral
*/

override func viewDidAppear(_ animated: Bool) { blePeripheral =
BlePeripheral(delegate: self)
}
// MARK: BlePeripheralDelegate

/**
Bluetooth radio state changed

- Parameters:
- state: the CBManagerState
*/

func blePeripheral(stateChanged state: CBManagerState) { switch (state) {
case CBManagerState.poweredOn:

print("Bluetooth on")
case CBManagerState.poweredOff:
print("Bluetooth off")
default:
print("Bluetooth not ready yet...")
}
}
}

```

The resulting app will be able to turn the Bluetooth Radio on (Figure 3-10).

●●●○ Fido 3G

13:45

100%  ⚡

**Figure 3-10. Main app screen**

## Example code

The code for this chapter is available online  
at: <https://github.com/BluetoothLowEnergyIniOSSwift/Chapter03>

## Scanning and Advertising

The first step to any Bluetooth Low Energy interaction is for the Peripheral to make the Central aware of its existence, through a process called Advertising. During the Advertising process, a peripheral Advertises while a Central Scans.

Bluetooth devices discover each other when they are tuned to the same radio frequency, also known as a Channel. There are three channels dedicated to device discovery in Bluetooth Low Energy (Table 4-1):

### **Table 4-1. Bluetooth Low Energy Discovery Radio Channels**

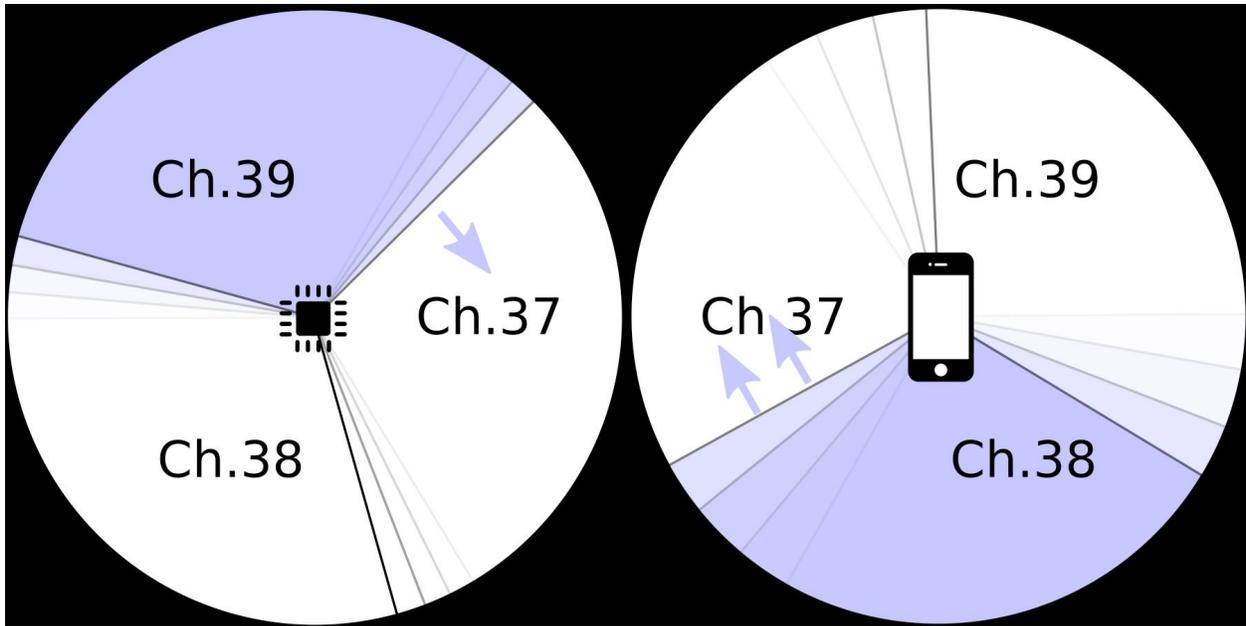
#### **Channel Radio Frequency**

**37** 2402 Mhz

**39** 2426 Mhz

**39** 2480 Mhz

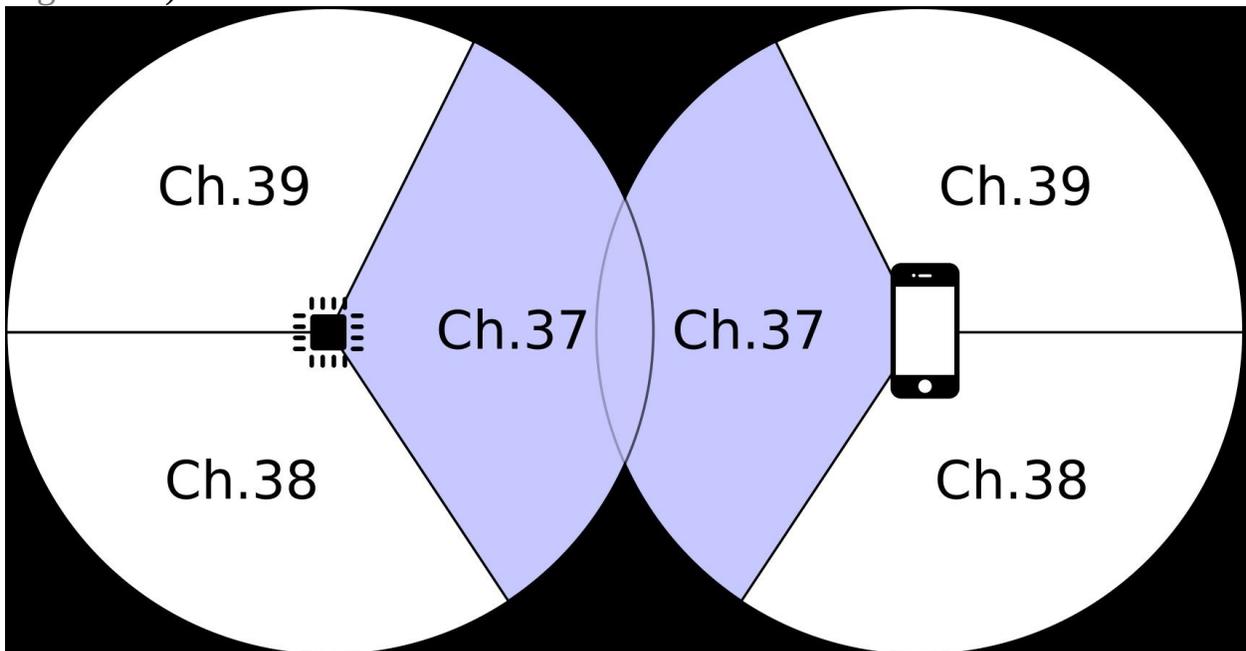
The peripheral will advertise its name and other data over one channel and then another. This is called frequency hopping (Figure 4-1).



**Figure 4-1. Advertise and scan processes**

Similarly, the Central listens for advertisements first on one channel and then another. The Central hops frequencies faster than the Peripheral, so that the two are guaranteed to be on the same channel eventually.

Peripherals may advertise from 100ms to 100 seconds depending on their configuration, changing channels every 0.625ms (Figure 4-2).



**Figure 4-2. Scan finds Advertiser**

Scanning settings vary wildly, for example scanning every 10ms for 100ms, or scanning for 1 second for 10 seconds.

## Programming the Central

The previous chapter showed how to access the Bluetooth hardware, specifically the CBCentralManager. This chapter will show how to scan for Bluetooth devices. This is done by scanning for Peripherals for a short period of time. During that time any time a Peripheral is discovered, the system will trigger a callback function. From there discovered Peripheral can be inspected.

The CBCentralManager lets you scan for Peripherals.

```
// scan for Peripherals
centralManager.scanForPeripherals(withServices: nil, options: nil)
```

If you happen to know one or more Service UUIDs hosted on a Peripheral your App is searching for, these UUIDs can be passed into the withServices parameter like this:

```
serviceUuids = [ "1815", "180C" ]
centralManager.scanForPeripherals(withServices: serviceUuids, options: nil)
```

Only Peripherals hosting matching Service UUIDs will be returned.

To stop an in-progress scan, execute the stopScan function:

```
centralManager.stopScan()
```

It is typical to scan for a period of time before stopping. 3-5 seconds is a reasonable amount of time to assume that most devices will be discovered during the scanning process.

This can be done with a Timer:

```
func startScan() {
scanCountdown = 5 // 5 seconds
scanTimer = Timer.scheduledTimer(
timeInterval: 1.0,
target: self,
selector: #selector(updateScanCounter), userInfo: nil, repeats: true
)
}
```

```
if let centralManager = centralManager {
centralManager.scanForPeripherals(withServices: nil, options: nil)
```

```

}
}

func updateScanCounter() {
//you code, this is an example if scanCountdown > 0 {

scanCountdown -= 1
} else {
centralManager?.stopScan() }
}

```

As each new Peripheral is discovered, the centralManager didDiscover event is triggered, which can reveal a Peripheral's advertised name and identifier. centralManager didDiscover can be implemented like this to get the Peripheral identifier:

```

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

{
let peripheralIdentifier = peripheral.identifier
}

```

For security reasons, iOS does not reveal the MAC address of a Peripheral. Instead, it creates a 32-bit UUID identifier.

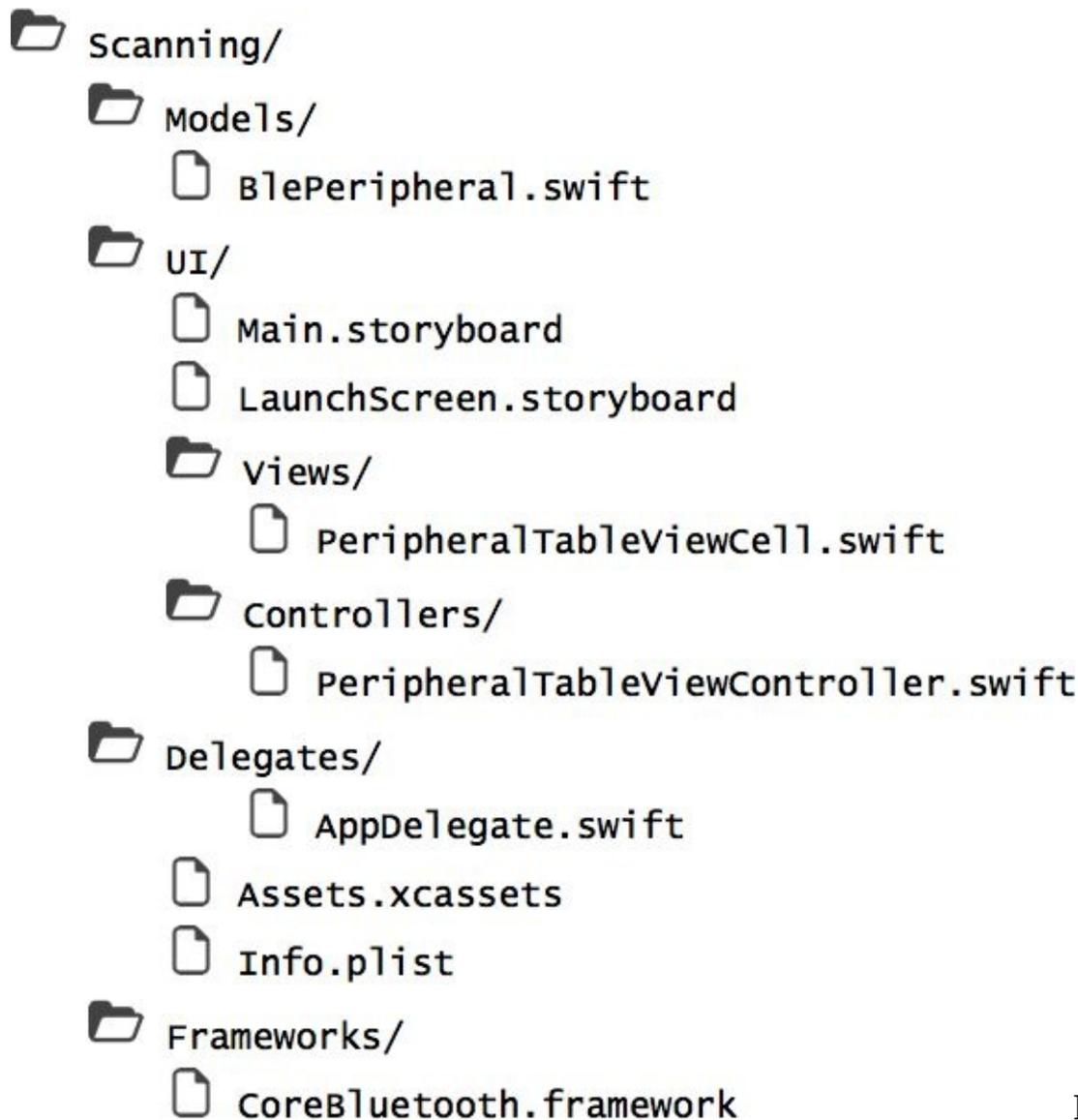
The Advertised name of a Peripheral is typically buried in the GAP Advertisement data, which can be retrieved like this:

```
var advertisedName = advertisementData["kCBAAdvDataLocalName"] as! String
```

## Putting It All Together

Create a new project called Scanner, and copy everything from the previous example.

Create a file structure that looks like this (Figure 4-3).



Figure

### 4-3. Project structure

This example will feature a UITableView to list discovered Peripherals

#### Models

Create a BlePeripheral class that holds information about a CBPeripheral.

#### Example 4-1. Models/BlePeripheral.swift

```
import UIKit
import CoreBluetooth
class BlePeripheral: NSObject {
// MARK: Peripheral properties
```

```
// connected Peripheral var peripheral:CBPeripheral! // advertised name
var advertisedName:String! // RSSI
```

```
var rssi:NSNumber!
```

```
/**
```

```
Initialize BlePeripheral with a corresponding Peripheral
```

```
- Parameters:
```

```
- delegate: The BlePeripheralDelegate
```

```
- peripheral: The discovered Peripheral
```

```
*/
```

```
init(peripheral: CBPeripheral) { super.init()
```

```
self.peripheral = peripheral
```

```
}
```

```
/** Get a broadcast name from an advertisementData packet. This may be  
different than the actual broadcast name */
```

```
static func getAlternateBroadcastFromAdvertisementData( advertisementData:  
[String : Any]) -> String?
```

```
{
```

```
// grab thekCBAdvDataLocalName from the advertisementData // to see if  
there's an alternate broadcast name
```

```
if advertisementData["kCBAdvDataLocalName"] != nil { return  
(advertisementData["kCBAdvDataLocalName"] as! String) }
```

```
return nil
```

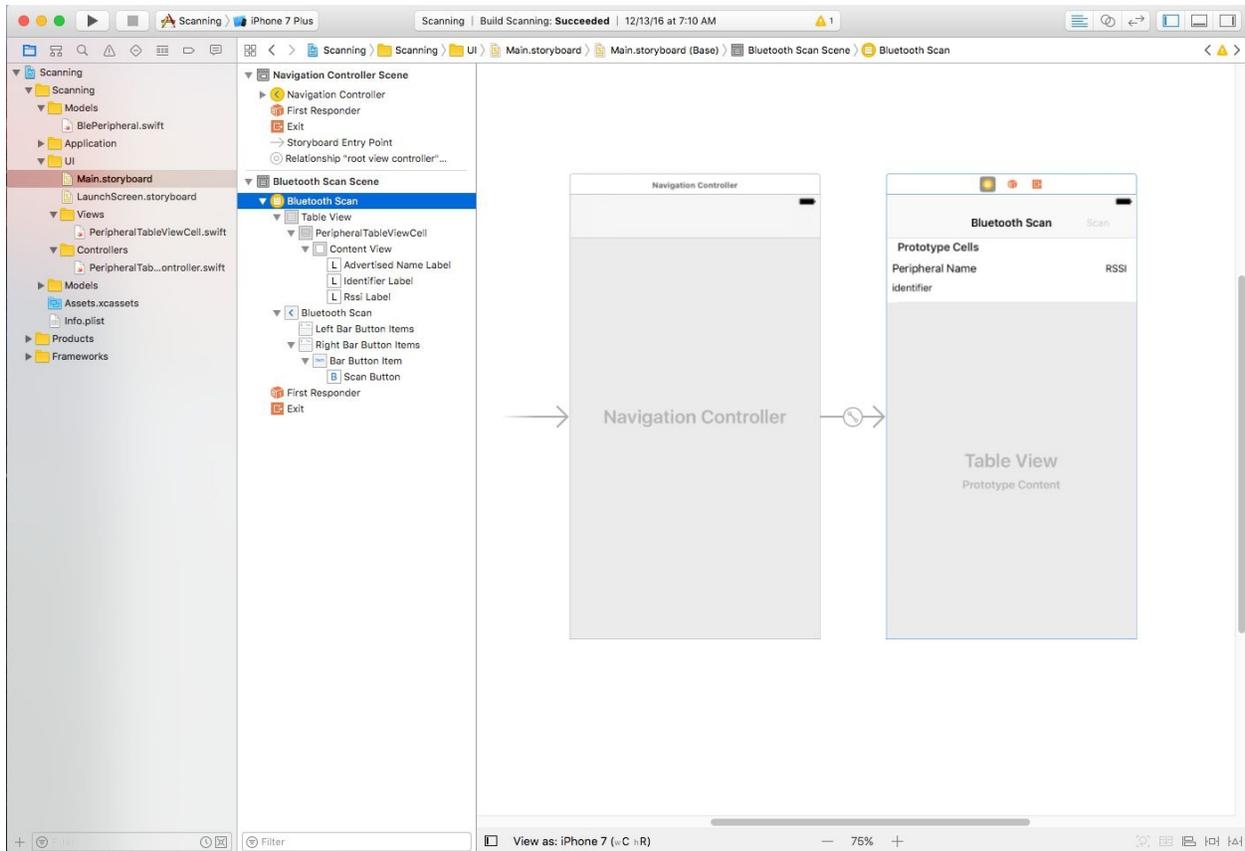
```
}
```

```
}
```

## Storyboard

Create a UITableView and connect it to the UINavigationController in place of the default UIView. Give it the class name "PeripheralTableView."

Add a UITableViewCell to it, with the class name "PeripheralTableViewCell" and the Reuse Identifier of "PeripheralTableViewCell." In the PeripheralTableViewCell, create and link three UILabels to be used for describing the connected Peripheral properties (Figure 4-4):



**Figure 4-4. Project Storyboard Views**

Link the three UILabels from the PeripheralTableViewCell to the corresponding swift file and create a render function:

**Example 4-2. UI/Views/PeripheralTableViewCell.swift**

```
import UIKit
import CoreBluetooth
class PeripheralTableViewCell: UITableViewCell {

// MARK: UI Elements
@IBOutlet weak var advertisedNameLabel: UILabel!
@IBOutlet weak var identifierLabel: UILabel!
@IBOutlet weak var rssiLabel: UILabel!

/**
Render Cell with Peripheral properties
*/

func renderPeripheral(_ blePeripheral: BlePeripheral) {
advertisedNameLabel.text = blePeripheral.advertisedName identifierLabel.text =
```

```
\
```

```
blePeripheral.peripheral.identifier.uuidString rssiLabel.text =  
blePeripheral.rssi.stringValue }  
}
```

## Controllers

The View Controller must be able to initialize a scan when the user clicks the Scan button. It will scan for Peripherals for 5 seconds, an arbitrarily reasonable scanning time. The UITableView is updated with each new Peripheral as discovered.

### Example 4-3. UI/Controllers/PeripheralTableViewController.swift

```
import UIKit  
import CoreBluetooth  
class PeripheralTableViewController: UITableViewController, \  
CBCentralManagerDelegate {  
  
    // MARK: UI Elements  
    @IBOutlet weak var scanButton: UIButton!  
    // Default unknown advertisement name  
    let unknownAdvertisedName = "(UNMARKED)"  
    // PeripheralTableViewCell reuse identifier  
    let peripheralCellReusedentifier = "PeripheralTableViewCell"  
  
    // MARK: Scan Properties // total scan time  
    let scanTimeout_s = 5; // seconds // current countdown  
    var scanCountdown = 0  
    // scan timer  
    var scanTimer: Timer!  
    // Central Bluetooth Manager  
    var centralManager: CBCentralManager! // discovered peripherals  
    var blePeripherals = [BlePeripheral]()  
  
    /**  
    View loaded. Start Bluetooth radio.  
    */  
  
    override func viewDidLoad() {
```

```

super.viewDidLoad()
print("Initializing central manager")
centralManager = CBCentralManager(delegate: self, queue: nil)

}

/**
User touched the "Scan/Stop" button
*/

@IBAction func onScanButtonTouched(_ sender: UIButton) { print("scan
button clicked")
// if scanning
if scanCountdown > 0 {

stopBleScan()
} else {
startBleScan()
}
}

/**
Scan for Bluetooth peripherals

*/
func startBleScan() {
scanButton.setTitle("Stop", for: UIControlState.normal)
blePeripherals.removeAll()
tableView.reloadData()
print ("discovering devices")
scanCountdown = scanTimeout_s
scanTimer = Timer.scheduledTimer(

timeInterval: 1.0,
target: self,
selector: #selector(updateScanCounter), userInfo: nil,
repeats: true)

if let centralManager = centralManager {
centralManager.scanForPeripherals(

```

```

withServices: nil,
options: nil)

}
}

/**
Stop scanning for Bluetooth Peripherals
*/

func stopBleScan() {
if let centralManager = centralManager {
centralManager.stopScan()

}
scanTimer.invalidate()
scanCountdown = 0
scanButton.setTitle("Start", for: UIControlState.normal)

}

/**
Update the scan countdown timer */
func updateScanCounter() {

//you code, this is an example
if scanCountdown > 0 {
print("\(scanCountdown) seconds until Ble Scan ends")
scanCountdown -= 1
} else {
stopBleScan()
}
}

// MARK: CBCentralManagerDelegate Functions
/**
New Peripheral discovered

```

- Parameters
- central: the CentralManager for this UIView

- peripheral: a discovered Peripheral
- advertisementData: the Bluetooth GAP data discovered
- rssi: the radio signal strength indicator for this Peripheral

```

*/
func centralManager(
  _ central: CBCentralManager,
  didDiscover peripheral: CBPeripheral,
  advertisementData: [String : Any],
  rssi RSSI: NSNumber)
{
  print("Discovered \(peripheral.identifier.uuidString) " + "\(peripheral.name)")

  // check if this peripheral has already been discovered var peripheralFound =
  false
  for blePeripheral in blePeripherals {

    if blePeripheral.peripheral.identifier == \ peripheral.identifier
    {
      peripheralFound = true break
    }
  }

  // don't duplicate discovered devices
  if !peripheralFound {
    print(advertisementData)
    // Broadcast name in advertisement data
    // may be different than the actual broadcast name
    // It's ideal to use the advertisement data version
    // as it's supported on programmable bluetooth devices
    var advertisedName = unknownAdvertisedName
    if let alternateName = \
    BlePeripheral.getAlternateBroadcastFromAdvertisementData(
    advertisementData: advertisementData)
    {
      if alternateName != "" {
        advertisedName = alternateName
      } else {
        if let peripheralName = peripheral.name{ advertisedName = peripheralName }

```

```
}  
}
```

```
let blePeripheral = BlePeripheral(peripheral: peripheral) blePeripheral.rssi =  
RSSI
```

```
blePeripheral.advertisedName = advertisedName
```

```
blePeripherals.append(blePeripheral)
```

```
tableView.reloadData()
```

```
}
```

```
}
```

```
/**
```

```
Bluetooth radio state changed
```

```
- Parameters:
```

```
- central: the reference to the central
```

```
*/
```

```
func centralManagerDidUpdateState(_ central: CBCentralManager) {  
    print("Central Manager updated: checking state") switch (central.state) {  
    case .poweredOn:
```

```
        print("BLE Hardware powered on and ready")
```

```
        scanButton.isEnabled = true
```

```
    default:
```

```
        print("Bluetooth unavailable")
```

```
    }
```

```
    }
```

```
// MARK: - Table view data source
```

```
/**
```

```
return number of sections. Only 1 is needed
```

```
*/
```

```
override func numberOfSections(in tableView: UITableView) -> Int { return 1
```

```
}
```

```
/**
```

```
Return number of Peripheral cells
```

```
*/
```

```

override func tableView(
    _ tableView: UITableView,
    numberOfRowsInSection section: Int) -> Int

{
    return blePeripherals.count
}

/**
Return rendered Peripheral cell

*/
override func tableView(
    _ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) -> UITableViewCell

{
    print("setting up table cell")
    let cell = tableView.dequeueReusableCell(

        withIdentifier: peripheralCellReusedentifier, for: indexPath) as!
        PeripheralTableViewCell
    // fetch the appropriate peripheral for the data source layout
    let peripheral = blePeripherals[indexPath.row]
    cell.renderPeripheral(peripheral)

    return cell }
}

```

Compile and run the app. When it runs, you will see a screen with a scan button. When the scan button is clicked, it locates your BLE device (Figure 4-5).

# Bluetooth Scan

Scan

---

---

---

---

---

---

---

●●●●○ Fido 3G

19:25

✳ 76%  ⚡

## Peripherals

Stop

---

(UNMARKED)

-96

9C1BCA3A-EA75-43C9-9C5F-744FA0AFEE7D

---

MyDevice

-56

ED4B9003-D4A9-4453-A731-4245167870B1

---

---

---

---

---

---

---

---

---

---

**Figure 4-5. App screen prior to a Bluetooth scan and after discovering Bluetooth Peripherals**

# Programming the Peripheral

The previous Chapter showed how to turn on the Bluetooth radio and detect if Peripheral is supported by the iOS hardware

This chapter will show how to advertise a Bluetooth Low Energy Peripheral. Advertising is simple. Create a Dictionary of advertising parameters and pass the Dictionary into the `CBPeripheralManager.startAdvertising` method:

```
let advertisementData:[String: Any] = [  
  
    CBAvertisementDataLocalNameKey: advertisingName ]  
peripheralManager.startAdvertising(advertisementData)
```

The Dictionary may contain up to two keys, `CBAvertisementDataLocalNameKey` and `CBAvertisementDataServiceUUIDsKey`:

## Table 4-2 . advertisementData Dictionary

### State Data Type Description

**CBAvertisementDataLocalNameKey** String

The advertised name of a Bluetooth Peripheral

**CBAvertisementDataServiceUUIDsKey**<sub>[CBUUID]</sub> UUIDs of listed Services

As this chapter is focused on Advertising, only the first parameter will be discussed.

When the Peripheral begins to fails to Advertise, the `peripheralManagerDidStartAdvertising` callback of the `CBPeripheralManagerDelegate` will be triggered, containing an updated `CBPeripheralManager` and possible Error:

```
func peripheralManagerDidStartAdvertising( _ peripheral:  
    CBPeripheralManager, error: Error?)  
  
{  
    if error != nil {  
        print ("Error advertising peripheral") print(error.debugDescription)    }  
}
```

```
}  
// store a copy of the updated Peripheral peripheralManager = peripheral  
}
```

To stop Advertising, use the `CBPeripheralManager.stopAdvertising` method:  
`peripheralManager.stopAdvertising()`

## Putting It All Together

Create a new app called `ExampleBlePeripheral`, and copy everything from the previous example.

This example will add a `UISwitch` that shows when a Peripheral has begun Advertising.

Custom scanner callbacks will be created, which respond to events when scanning has stopped.

### Models

Add an `advertisingName` to the `BlePeripheral`, functionality to start and stop Advertising, and a callback handler to handle the `peripheralManagerDidStartAdvertising` callback:

#### Example 4-4. Models/`BlePeripheral.swift`

```
class BlePeripheral : NSObject, CBPeripheralManagerDelegate {  
// MARK: Peripheral properties  
// Advertized name  
let advertisingName = "MyDevice" ...  
/**  
Stop advertising, shut down the Peripheral */  
func stop() {  
peripheralManager.stopAdvertising() }  
  
/**  
Start Bluetooth Advertising.  
*/  
  
func startAdvertising() {  
let advertisementData:[String: Any] = [  
  
CBAdvertisementDataLocalNameKey: advertisingName ]
```

```

peripheralManager.startAdvertising(advertisementData)
}
// MARK: CBPeripheralManagerDelegate

/**
Peripheral started advertising
*/

func peripheralManagerDidStartAdvertising( _ peripheral:
CBPeripheralManager, error: Error?)

{
if error != nil {
print ("Error advertising peripheral") print(error.debugDescription)
}
self.peripheralManager = peripheral
delegate?.blePeripheral?(startedAdvertising: error)
}
...
}

```

## Delegates

Add a new method to the BlePeripheralDelegate to relay the Advertising started event:

### Example 4-5. Delegates/BlePeripheralDelegate.swift

```

... /**
BlePeripheral started advertising

```

- Parameters:
  - error: the error message, if any
- ```

*/

```

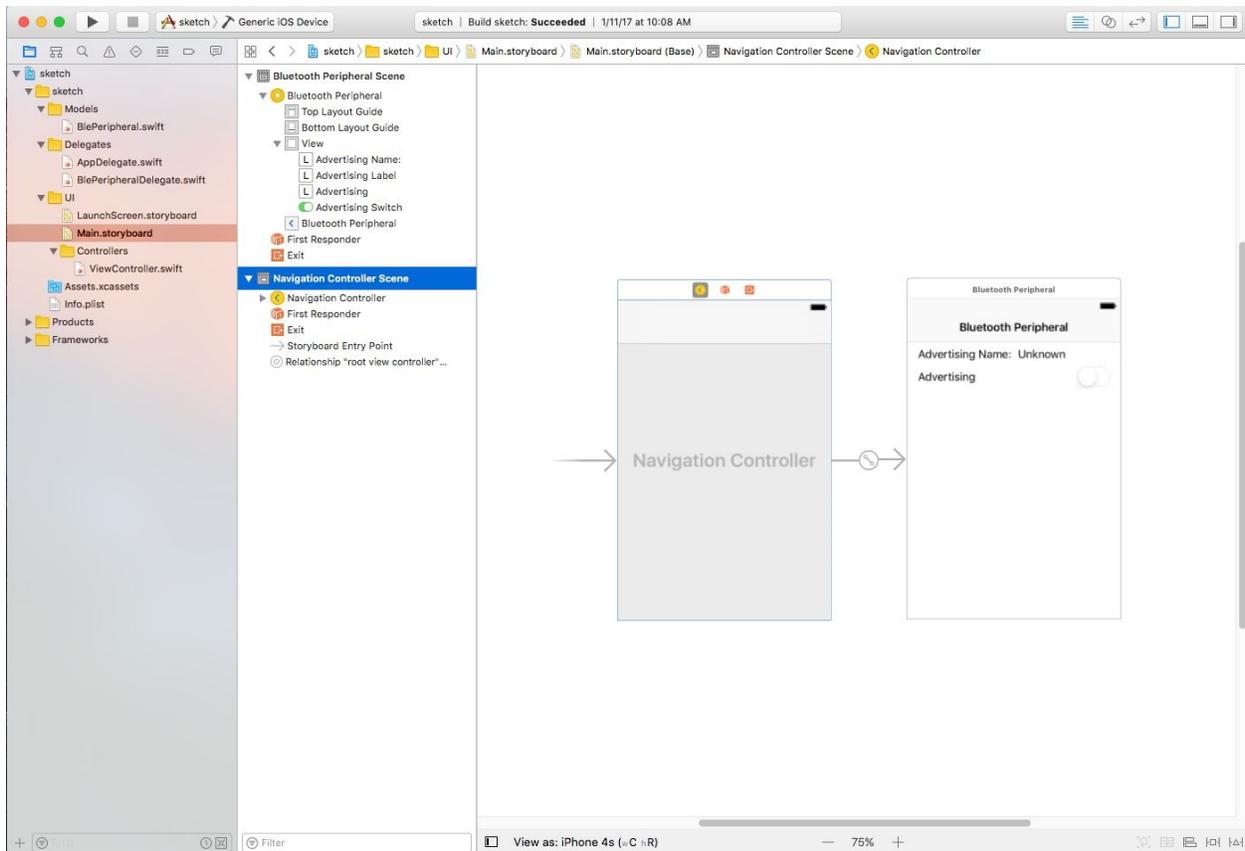
```

@objc optional func blePeripheral(startedAdvertising error: Error?) ...

```

## Storyboard

Add a UILabel to show the Peripheral's Advertised name and a UISwitch to show the Advertising state of the Peripheral (Figure 4-6):



**Figure 4-6. Project Storyboard Controllers**

Add a UILabel to display the Advertising name and a UISwitch to show the Advertising state of the Peripheral. Add functionality in the viewDidLoad, viewWillAppear, and viewWillDisappear to change the text on the UILabel and the state of the UISwitch to reflect the state of the Peripheral. And add a callback handler for the new BlePeripheralDelegate method:

**Example 4-6. UI/Controllers/ViewController.swift**

```

...
// MARK: UI Elements
@IBOutlet weak var advertisingLabel: UILabel! @IBOutlet weak var
advertisingSwitch: UISwitch!

...
/**
View appeared. Start the Peripheral
*/

```

```

override func viewDidAppear(_ animated: Bool) { blePeripheral =
BlePeripheral(delegate: self) advertisingLabel.text =
blePeripheral.advertisingName

}

/**
View will appear. Stop transmitting random data */

override func viewWillDisappear(_ animated: Bool) { blePeripheral.stop()
}

/**
View disappeared. Stop advertising
*/

override func viewDidDisappear(_ animated: Bool) {
advertisingSwitch.setOn(false, animated: true)
}
...
/**
BlePeripheral started advertising

- Parameters:
- error: the error message, if any
*/

func blePerihperal(startedAdvertising error: Error?) {
if error != nil {
print("Problem starting advertising: " + error.debugDescription)

} else {
print("adertising started")
advertisingSwitch.setOn(true, animated: true)

}
}
...
Compile and run the app. When it runs, a Bluetooth Peripheral will be
advertising (

```

Figure 4-7).

●●●● Fido 3G

13:43

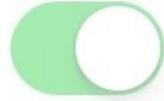
⌘ 88%  ⚡

## Bluetooth Peripheral

---

Advertising Name: MyDevice

Advertising



**Figure 4-7. App screen showing advertising Peripheral**

## Example code

The code for this chapter is available online  
at: <https://github.com/BluetoothLowEnergyIniOSSwift/Chapter04>

## Connecting

Each Bluetooth Device has a unique Media Access Control (MAC) address, a 48-bit identifier value written like this

00:A0:C9:14:C8:3A

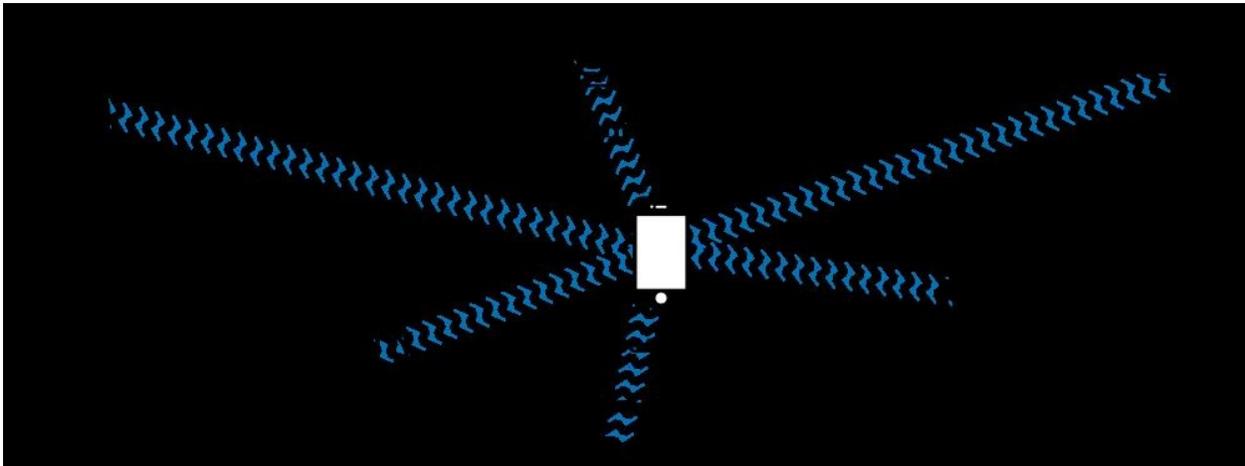
Devices advertise data on the network with the intended recipient's MAC address attached so that recipient devices can filter data packets that are intended for them.

For security reasons, iOS does not reveal the MAC address of a Peripheral.

Instead, it creates a 32-bit UUID identifier, like this:

2fe058bd-5edb-4b3f-b7bd-fc8e93e2dbc4

Once a Central has discovered a Peripheral, the central can attempt to connect. This must be done before data can be passed between the Central and Peripheral. A Central may hold several simultaneous connections with a number of peripherals, but a Peripheral may only hold one connection at a time. Hence the names Central and Peripheral (Figure 5-1).

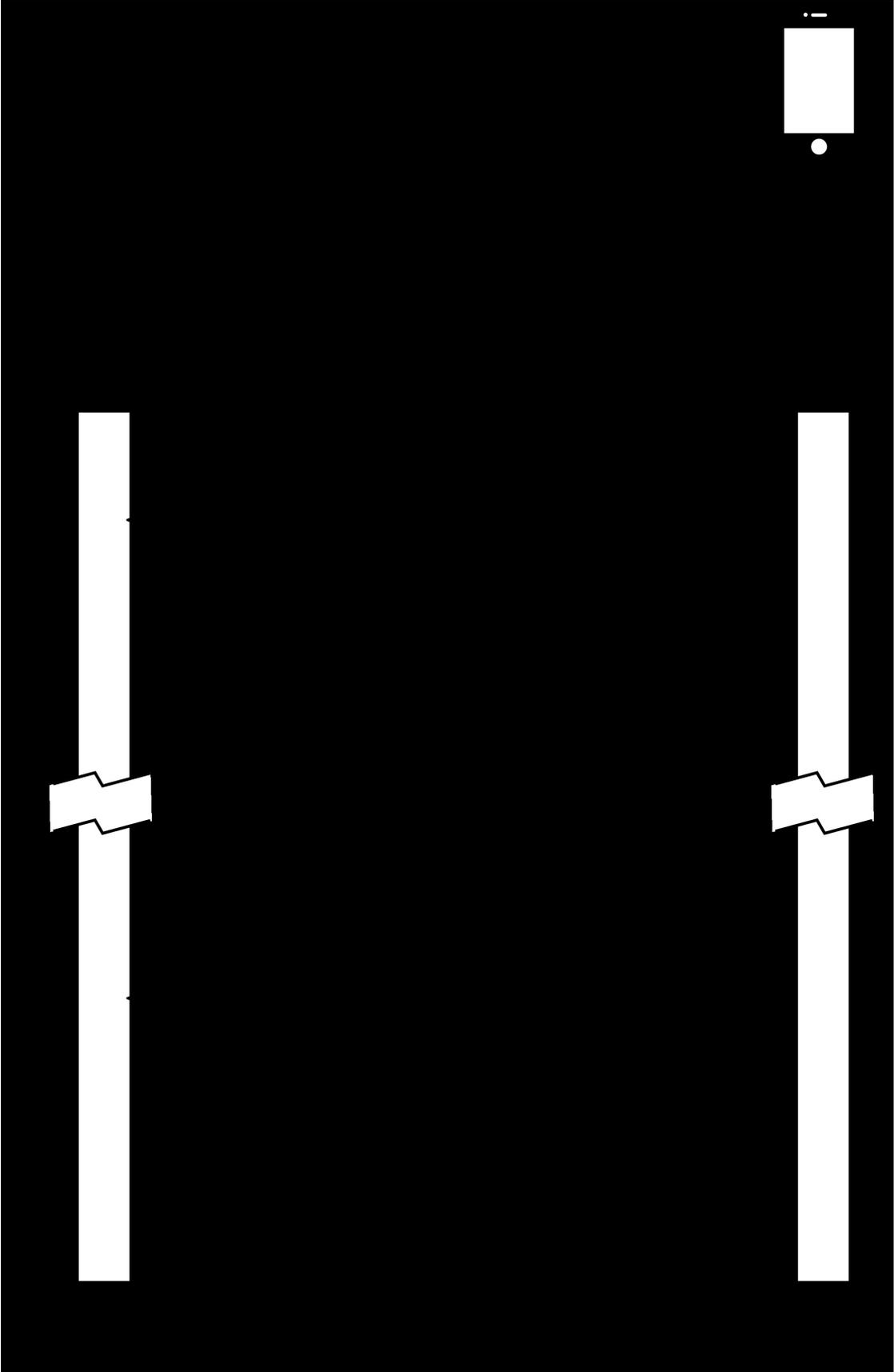


**Figure 5-1. Bluetooth network topology**

Bluetooth supports data 37 data channels ranging from 2404 MHz to 2478 MHz. Once the connection is established, the Central and Peripheral negotiate which of these channels to begin communicating over.

Because the Peripheral can only hold one connection at a time, it must disconnect from the Central before a new connection can be made.

The connection and disconnection process works like this (Figure 5-2).



**Figure 5-2. Connection and disconnection process**

## Programming the Central

The previous chapter's App showed how to discover nearby Peripherals. Once a Peripheral is discovered, iOS can initiate a connection like this:

```
centralManager.connect(peripheral)
```

If the connection is successful, the `didConnect` callback will be triggered.

```
func centralManager(
    _ central: CBCentralManager, didConnect peripheral: CBPeripheral)
{
}
}
```

The `didFailToConnect` will be triggered otherwise.

```
func centralManager(
    _ central: CBCentralManager,
    didFailToConnect peripheral: CBPeripheral, error: Error?)
{
}
}
```

A disconnection can be initiated like this:

```
centralManager.cancelPeripheralConnection(peripheral)
```

When successful, the `didDisconnectPeripheral` event will be triggered:

```
func centralManager(
    _ central: CBCentralManager,
    didDisconnectPeripheral peripheral: CBPeripheral, error: Error?)
{
}
}
```

Disconnecting is important. The Peripheral can only be connected to one device at a time. Sometimes, closing an Activity without disconnecting the Peripheral can leave the Peripheral in a connected state - unable to advertise or connect to a Central again in the future.

## Putting It All Together

Create a new project called Connecting and copy everything from the previous example. This example will show how to create an app that looks for a “MyDevice” BLE advertisement, and connect to it.

BlePeripheral represents a remote Peripheral. PeripheralViewController will connect to the Peripheral and list the Peripheral properties.

Create a project structure that resembles this (Figure 5-3).



Figure 5-3.

**Added project structure**

## Models

Modify the BlePeripheral to support CBPeripheralDelegate callbacks

### Example 5-1. Models/BlePeripheral.swift

```
import UIKit
import CoreBluetooth
class BlePeripheral: NSObject, CBPeripheralDelegate { // MARK: Peripheral
properties
```

```
// delegate
```

```
var delegate:BlePeripheralDelegate? // connected Peripheral
```

```
var peripheral:CBPeripheral! // advertised name
```

```
var advertisedName:String!
```

```
// RSSI
```

```
var rssi:NSNumber!
```

```
/**
```

Initialize BlePeripheral with a corresponding Peripheral

- Parameters:

- delegate: The BlePeripheralDelegate

- peripheral: The discovered Peripheral

```
*/
```

```
init(delegate: BlePeripheralDelegate?, peripheral: CBPeripheral) { super.init()
```

```
self.peripheral = peripheral
```

```
self.peripheral.delegate = self
```

```
self.delegate = delegate
```

```
}
```

```
/**
```

Notify the BlePeripheral that the peripheral has been connected

- Parameters:

- peripheral: The discovered Peripheral \*/

```
func connected(peripheral: CBPeripheral) { self.peripheral = peripheral
```

```
self.peripheral.delegate = self // check for services and the RSSI
```

```
self.peripheral.readRSSI()
```

```
}
```

```

/**
Get a broadcast name from an advertisementData packet. This may be different
than the actual broadcast name */

static func getAlternateBroadcastFromAdvertisementData( advertisementData:
[String : Any]) -> String?
{
// grab thekCBAdvDataLocalName from the advertisementData // to see if
there's an alternate broadcast name
if advertisementData["kCBAdvDataLocalName"] != nil { return
(advertisementData["kCBAdvDataLocalName"] as! String) }
return nil
}

/**
Determine if this peripheral is connectable
from it's advertisementData packet.
*/

static func isConnectable(advertisementData: [String: Any]) -> Bool { let
isConnectable = \
advertisementData["kCBAdvDataIsConnectable"] as! Bool return isConnectable
}
// MARK: CBPeripheralDelegate

/**
RSSI read from peripheral. */

func peripheral(
_ peripheral: CBPeripheral, didReadRSSI RSSI: NSNumber, error: Error?)

{
print("RSSI: \(RSSI.stringValue)")
rssi = RSSI
delegate?.blePeripheral?(readRssi: rssi, blePeripheral: self)
} }

```

## Delegates

Create a BlePeripheralDelegate class that lets the BlePeripheral trigger events as

a result of changes in its state.

### **Example 5-2. Models/BlePeripheralDelegate.swift**

```
import UIKit
import CoreBluetooth
@objc protocol BlePeripheralDelegate: class { /**
RSSI was read for a Peripheral

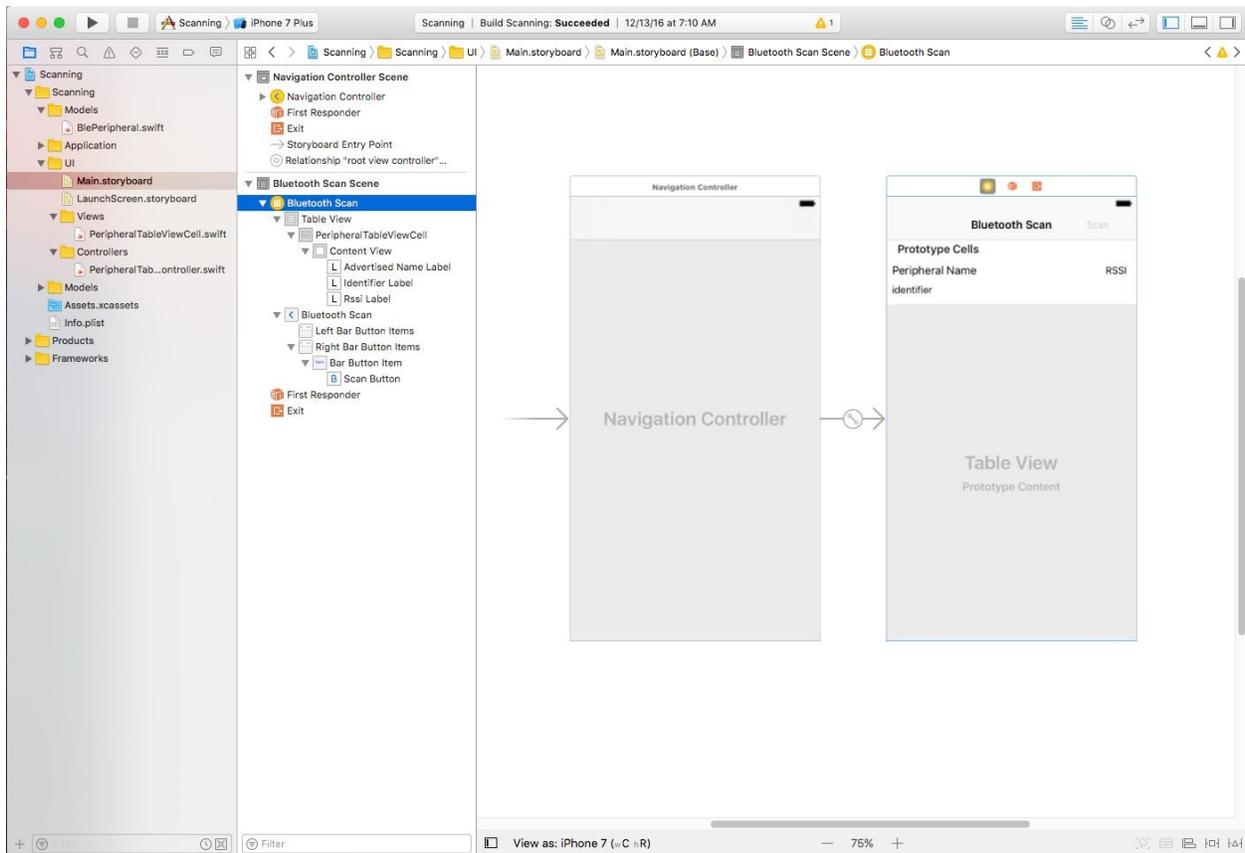
- Parameters:
- rssi: the RSSI
- blePeripheral: the BlePeripheral

*/
@objc optional func blePeripheral( readRssi rssi: NSNumber, blePeripheral:
BlePeripheral)

}
```

### **Storyboard**

Create a new UIView and give it the class name "PeripheralViewController."  
Add a segue between the two UIViews. Create and link three UILabels to be used for describing the connected Peripheral properties (Figure 5-4):



**Figure 5-4. Project Storyboard Controllers**

The previous app was able to detect and list nearby Peripherals. This app will allow connecting to a Peripheral when a user selects that Peripheral from the UITableView.

Add a segue and UITableViewDelegate functionality in the PeripheralTableViewController.

**Example 5-3. UI/Controllers/PeripheralTableViewController.swift**

```

... override func tableView(
  _ tableView: UITableView,
  didSelectRowAt indexPath: IndexPath)

{
  stopBleScan()
  let selectedRow = indexPath.row print("Row: \(selectedRow)")
  print(blePeripherals[selectedRow])
}
// MARK: - Navigation

```

```

override func prepare(for segue: UIStoryboardSegue, sender: Any?) { let
peripheralViewController = \

segue.destination as! PeripheralViewController
if let selectedIndexPath = tableView.indexPathForSelectedRow {
let selectedRow = selectedIndexPath.row
if selectedRow < blePeripherals.count {
// prepare next UIView
peripheralViewController.centralManager = centralManager
peripheralViewController.blePeripheral = \
blePeripherals[selectedRow]
}
tableView.deselectRow(at: selectedIndexPath, animated: true) }
}
}

```

The PeripheralViewController will connect to a Peripheral and list the advertised name and identifier in UILabels

When a connection is confirmed, the user interface is updated.

#### **Example 5-4. UI/Controllers/PeripheralViewController.swift**

```

import UIKit
import CoreBluetooth
class PeripheralViewController: UIViewController, \
CBCentralManagerDelegate, BlePeripheralDelegate {

// MARK: UI Elements
@IBOutlet weak var advertisedNameLabel: UILabel! @IBOutlet weak var
identifierLabel: UILabel! @IBOutlet weak var rssiLabel: UILabel!

// MARK: Connected Peripheral Properties

// Central Manager
var centralManager:CBCentralManager! // connected Peripheral
var blePeripheral:BlePeripheral!

/**
UIView loaded
*/

override func viewDidLoad() {

```

```

super.viewDidLoad()
print("Will connect to " + \

"\(blePeripheral.peripheral.identifier.uuidString)") // Assign delegates
blePeripheral.delegate = self
centralManager.delegate = self
centralManager.connect(blePeripheral.peripheral)

}
/**
RSSI discovered. Update UI

*/
func blePeripheral(
readRssi rssi: NSNumber, blePeripheral: BlePeripheral)

{
rssiLabel.text = rssi.stringValue
}

// MARK: CBCentralManagerDelegate code

/**
Peripheral connected. Update UI */

func centralManager(
_ central: CBCentralManager, didConnect peripheral: CBPeripheral)

{
print("Connected Peripheral: \(peripheral.name)") advertisedNameLabel.text =
blePeripheral.advertisedName identifierLabel.text =\

blePeripheral.peripheral.identifier.uuidString blePeripheral.connected(peripheral:
peripheral) }

/**
Connection to Peripheral failed.
*/

func centralManager(

```

```

_ central: CBCentralManager,
didFailToConnect peripheral: CBPeripheral, error: Error?)

{
print("failed to connect") print(error.debugDescription) /**

Peripheral disconnected. Leave UIView

*/
func centralManager(
_ central: CBCentralManager,
didDisconnectPeripheral peripheral: CBPeripheral, error: Error?)

{
print("Disconnected Peripheral: \(peripheral.name)") dismiss(animated: true,
completion: nil)

}

/**
Bluetooth radio state changed.
*/

func centralManagerDidUpdateState(_ central: CBCentralManager) {
print("Central Manager updated: checking state") switch (central.state) {
case .poweredOn:

print("bluetooth on")
default:
print("bluetooth unavailable")
}
}

// MARK: Navigation
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
print("leaving view - disconnecting from peripheral")
if let peripheral = blePeripheral.peripheral {
centralManager.cancelPeripheralConnection(peripheral)
}
}
}

```

The resulting App will be one that can scan and connect to an advertising Peripheral (Figure 5-5).

●●●●○ Fido 3G

19:25

⌘ 76%  ⚡

[← Peripherals](#) **Peripheral**

**MyDevice** -77

ED4B9003-D4A9-4453-A731-4245167870B1

**Figure 5-5. App screen after connecting to a Peripheral**

## **Peripheral Programming**

In iOS, no notifications are sent when a Peripheral is connected to. There Peripheral code remains the same as the previous chapter.

## Example code

The code for this chapter is available online  
at: <https://github.com/BluetoothLowEnergyIniOSSwift/Chapter05>

## Services and Characteristics

Before data can be transmitted back and forth between a Central and Peripheral, the Peripheral must host a GATT Profile. That is, the Peripheral must have Services and Characteristics.

### Identifying Services and Characteristics

Each Service and Characteristic is identified by a Universally Unique Identifier (UUID). The UUID follows the pattern 0000XXXX-0000-1000-8000-00805f9b34fb, so that a 32-bit UUID 00002a56-0000-1000-8000-00805f9b34fb can be represented as 0x2a56.

Some UUIDs are reserved for specific use. For instance any Characteristic with the 16-bit UUID 0x2a35 (or the 32-bit UUID 00002a35-0000-1000-8000-00805f9b34fb) is implied to be a blood pressure reading.

For a list of reserved Service UUIDs, see ***Appendix IV: Reserved GATT Services***.

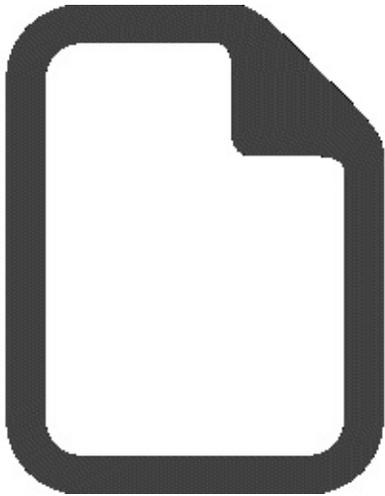
For a list of reserved Characteristic UUIDs, see ***Appendix V: Reserved GATT Characteristics***.

### Generic Attribute Profile

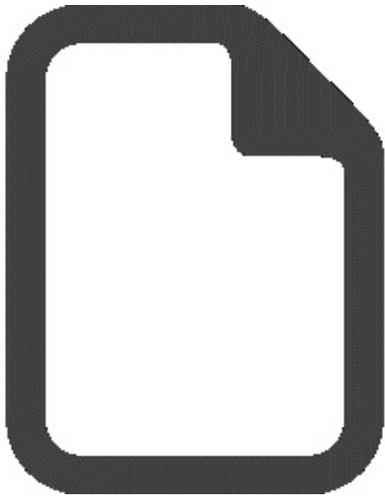
Services and Characteristics describe a tree of data access points on the peripheral. The tree of Services and Characteristics is known as the Generic Attribute (GATT) Profile. It may be useful to think of the GATT as being similar to a folder and file tree (Figure 6-1).



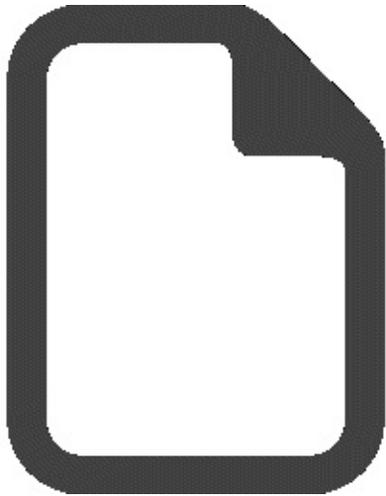
Service/



Characteristic



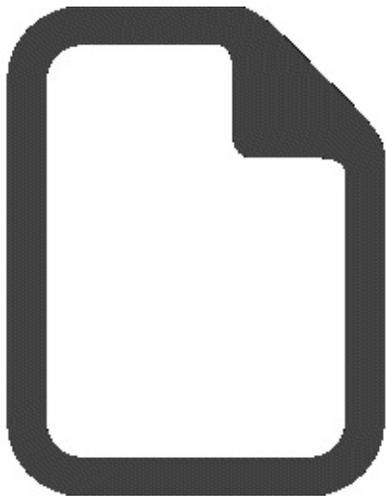
Characteristic



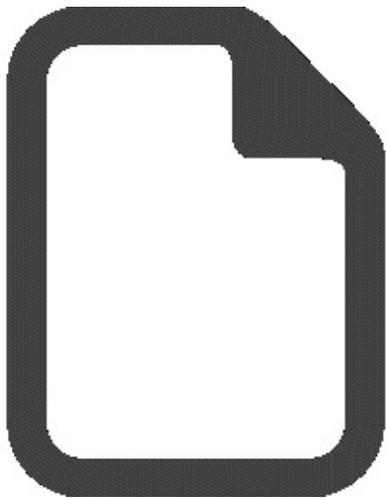
Service/



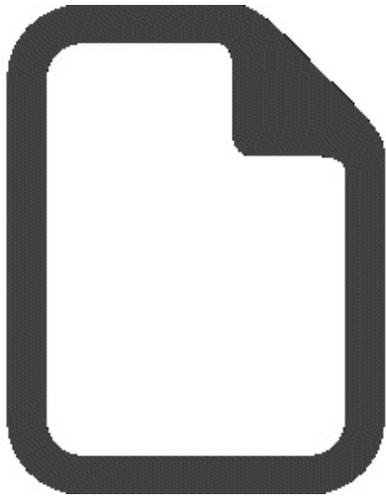
Characteristic



Characteristic



Characteristic



Characteristic

## Profile

### Service

Characteristic

Characteristic

Characteristic

### Service

Characteristic

Characteristic

Characteristic

## Figure 6-1. GATT Profile filesystem metaphor

Characteristics act as channels that can be communicated on, and Services act as containers for Characteristics. A top level Service is called a Primary service, and a Service that is within another Service is called a Secondary Service.

### Permissions

Characteristics can be configured with the following attributes, which define what the Characteristic is capable of doing (Table 6-1):

#### Table 6-1. Characteristic Permissions

##### Descriptor Description

**Read** Central can read this Characteristic, Peripheral can set the value.

Central can write to this Characteristic, Peripheral will be notified when **Write** the Characteristic value changes and Central will be notified when the write operation has occurred.

Central can write to this Characteristic. Peripheral will be notified when **Write without Response** the Characteristic value changes but the Central will not be notified that the write operation has occurred.

**Notify** Central will be notified when Peripheral changes the value.

Because the GATT Profile is hosted on the Peripheral, the terms used to describe a Characteristic's permissions are relative to how the Peripheral accesses that Characteristic. Therefore, when a Central uploads data to the Peripheral, the Peripheral can "read" from the Characteristic. The Peripheral "writes" new data to the Characteristic, and can "notify" the Central that the data is altered.

### Data Length and Speed

It is worth noting that Bluetooth Low Energy has a maximum data packet size of 20 bytes, with a 1 Mbit/s speed.

# Programming the Central

The Central can be programmed to read the GATT Profile of the Peripheral after connection, like this:

```
peripheral.discoverServices(nil)
```

If only a subset of the Services hosted by the Peripheral are needed, those Service UUIDs can be passed into the discoverServices function like this:

```
let serviceUuids = [ "1800", "1815" ] peripheral.discoverServices(serviceUuids)
```

When the Services are discovered, a callback will be executed by the CBPeripheralManagerDelegate, containing an updated CBPeripheral object. This updated object contains an array of Services:

```
func peripheral(
    _ peripheral: CBPeripheral,
    didDiscoverServices error: Error?)

{
    if error != nil {
        print("Discover service Error: \(error)")
    }
}
```

In order for the class to access these methods, it must implement CBPeripheralManagerDelegate.

There are Primary Services and Secondary services. Secondary Services are contained within other Services; Primary Services are not. The type of Service can be discovered by inspecting the CBService.isPrimary flag.

```
boolean isPrimary = service.isPrimary
```

To discover the Characteristics hosted by these services, simply loop through the discovered Services and handle the resulting peripheral didDiscoverCharacteristicsFor callback:

```
func peripheral(
    _ peripheral: CBPeripheral, didDiscoverServices error: Error?)

{
```

```

if error != nil {
print("Discover service Error: \(error)")
} else {
for service in peripheral.services!{
self.peripheral.discoverCharacteristics(nil, for: service) }
}
}

func peripheral(
_peripheral: CBPeripheral,
didDiscoverCharacteristicsFor service: CBService, error: Error?)

{
let serviceIdentifier = service.uuid.uuidString if let characteristics =
service.characteristics {

for characteristic in characteristics { // do something with Characteristic
}
}
}
}

```

Each Characteristic has certain permission properties that allow the Central to read, write, or receive notifications from it (Table 6-2).

### **Table 6-2. CBCharacteristicProperties**

#### **Value Permission Description**

**read Read** Central can read data altered by the Peripheral

**write Write** Central can send data, Peripheral reads

**writeWithoutResponse Write** Central can send data. No response from Peripheral

**notify Notify** Central is notified as a result of a change

In iOS, these properties are expressed as a binary integer which can be extracted like this:

```
let properties = characteristic.properties.rawValue
```

```
let isWritable = (properties & \
```

```
CBCharacteristicProperties.write.rawValue) != 0;
```

```
let isWritableNoResponse = (properties & \
```

```
CBCharacteristicProperties.writeWithoutResponse.rawValue) != 0; let
```

```
isReadable = (properties & \
CBCharacteristicProperties.read.rawValue) != 0;
let isNotifiable = (properties & \
CBCharacteristicProperties.notify.rawValue) != 0;
```

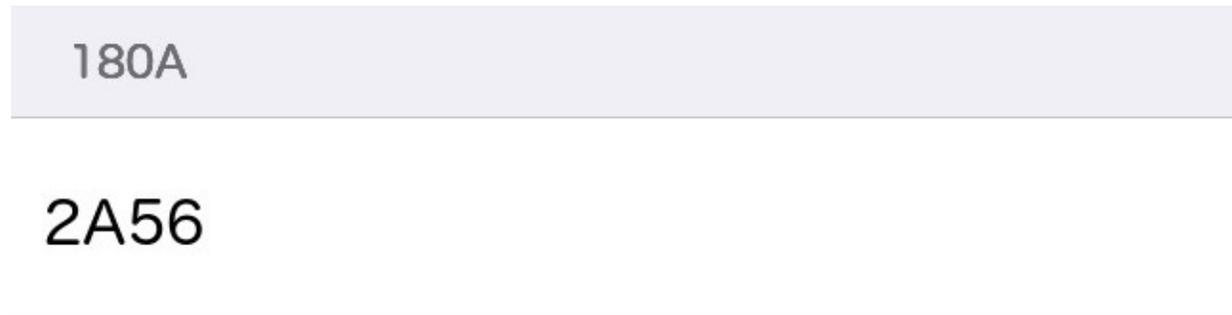
## A Note on Caching

Because Bluetooth was designed to be a low-power protocol, measures are taken to limit redundancy and power consumption through radio and CPU usage. As a result, a Peripheral's GATT Profile is cached on iOS. This is not a problem for normal use, but when developing, it can be confusing to change Characteristic permissions and not see the updates reflected on iOS.

To get around this, the iOS device must be restarted each time a Peripheral with the same Identifier has changed its GATT Profile

## Putting It All Together

This app will work like the one from the previous chapter, except that once the it connects to the Peripheral, it will also list the GATT Profile for that Peripheral. The GATT Profile will be displayed in an UITableView (Figure 6-2).



180A

2A56

### **Figure 6-2. GATT Profile downloaded from Peripheral**

Create a new project called Services and copy everything from the previous example. (Figure 6-3).



Figure 6-3.

## Project Structure

### Objects

Modify BlePeripheral.swift to discover Services and Characteristics

### Example 6-1. Models/BlePeripheral.swift

...

/\*\*

Services were discovered on the connected Peripheral \*/

```

func peripheral(
  _ peripheral: CBPeripheral,
  didDiscoverServices error: Error?)

{
  print("services discovered")
  // clear GATT profile - start with fresh services listing gattProfile.removeAll()
  if error != nil {

    print("Discover service Error: \(error)")
  } else {
    print("Discovered Service")
    for service in peripheral.services! {
      self.peripheral.discoverCharacteristics(nil, for: service)
    }
    print(peripheral.services!)
  }
}

```

```

/**
Characteristics were discovered
for a Service on the connected Peripheral
*/

```

```

func peripheral(
  _ peripheral: CBPeripheral,
  didDiscoverCharacteristicsFor service: CBService, error: Error?)

{
  print("characteristics discovered")
  // grab the service
  let serviceIdentifier = service.uuid.uuidString print("service: \(
  serviceIdentifier)")

  gattProfile.append(service)
  if let characteristics = service.characteristics { print("characteristics found: \(
  characteristics.count)")
  for characteristic in characteristics {
    print("-> \(characteristic.uuid.uuidString)")
  }
}

```

```

}
delegate?.blePerihperal?(
discoveredCharacteristics: characteristics,
forService: service,
blePeripheral: self)
}
}
...

```

## Delegates

Add a function to the BlePeripheralDelegate to alert when Characteristics have been discovered:

### Example 6-2. Delegates/BlePeripheralDelegate.swift

```

... /**
Characteristics were discovered for a Service

```

- Parameters:
- characteristics: the Characteristic list
- forService: the Service these Characteristics are under
- blePeripheral: the BlePeripheral

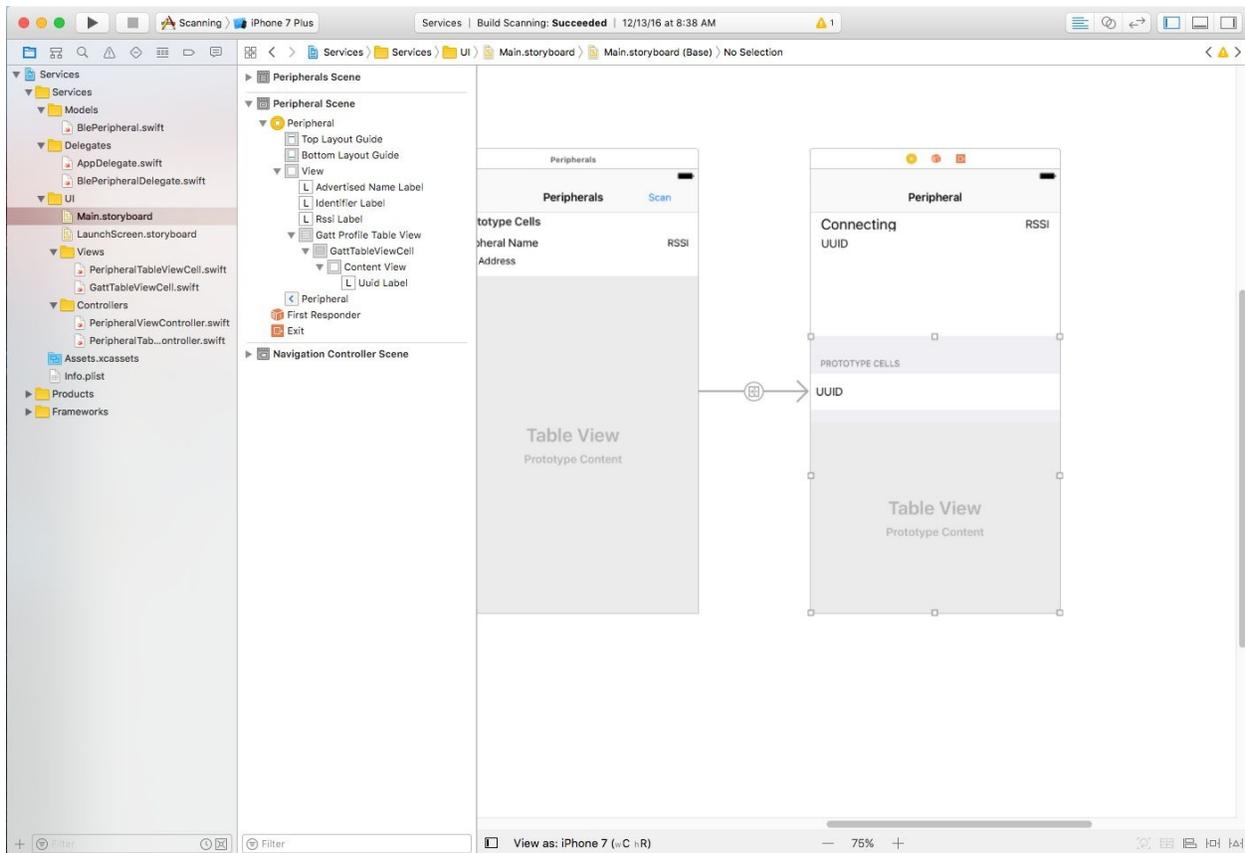
```

*/
@objc optional func blePerihperal(
discoveredCharacteristics characteristics: [CBCharacteristic], forService:
CBService,
blePeripheral: BlePeripheral)

```

## Storyboard

Add a UITableView and UITableViewCell to the PeripheralViewController. Make the UITableView a "grouped" TableView and make the UITableViewCell of class "GattTableViewCell" Give it the Reuse Identifier "GattTableViewCell." In the new GattTableViewCell, create and link a UILabel to be used to hold the Characteristic UUID (Figure 6-4):



**Figure 6-4. Project Storyboard Views**

The GATT Profile will be represented as a Grouped UITableView, with Services as the table header and Characteristics as the GattTableViewCell table cell. Each GattTableViewCell will display the UUID of a Characteristic.

**Example 6-3. UI/Views/GattTableViewCell.swift**

```
import UIKit
import CoreBluetooth
class GattTableViewCell: UITableViewCell { @IBOutlet weak var uuidLabel: UILabel!

func renderCharacteristic(characteristic: CBCharacteristic) { uuidLabel.text = characteristic.uuid.uuidString print(characteristic.uuid.uuidString)

}
}
```

**Controllers**

Add functionality in the PeripheralViewController to render the GATT Profile

table and to handle the blePeripheral discoveredCharacteristics callback from the BlePeripheralDelegate class:

#### **Example 6-4. UI/Controllers/PeripheralViewController.swift**

```
class PeripheralViewController: UIViewController, UITableViewDataSource, \
UITableViewDelegate, CBCentralManagerDelegate, BlePeripheralDelegate {
```

```
// MARK: UI Elements
```

```
@IBOutlet weak var advertisedNameLabel: UILabel! @IBOutlet weak var
identifierLabel: UILabel! @IBOutlet weak var rssiLabel: UILabel!
```

```
@IBOutlet weak var gattProfileTableView: UITableView! @IBOutlet weak var
gattTableView: UITableView!
```

```
// Gatt Table Cell Reuse Identifier
```

```
let gattCellReuseIdentifier = "GattTableViewCell" // MARK:
BlePeripheralDelegate
```

```
/**
```

```
Characteristics were discovered. Update the UI
```

```
*/
```

```
func blePeripheral(
discoveredCharacteristics characteristics: [CBCharacteristic], forService:
CBService,
blePeripheral: BlePeripheral)
```

```
{
gattTableView.reloadData()
}
```

```
/**
```

```
RSSI discovered. Update UI */
```

```
func blePeripheral(
readRssi rssi: NSNumber, blePeripheral: BlePeripheral)
```

```
{
rssiLabel.text = rssi.stringValue
}
```

```
// MARK: UITableViewDataSource
```

```
/**
```

```
Return number of rows in Service section */
```

```
func tableView(  
_ tableView: UITableView,  
numberOfRowsInSection section: Int) -> Int  
  
{  
print("returning num rows in section")  
if section < blePeripheral.gattProfile.count {  
  
if let characteristics = \  
blePeripheral.gattProfile[section].characteristics  
  
{ return characteristics.count  
}  
}  
return 0  
}
```

```
/**
```

```
Return a rendered cell for a Characteristic
```

```
*/
```

```
func tableView(  
_ tableView: UITableView,  
cellForRowAt indexPath: IndexPath) -> UITableViewCell
```

```
{  
print("returning table cell")  
let cell = tableView.dequeueReusableCell(  
  
withIdentifier: gattCellReuseIdentifier, for: indexPath) as! GattTableViewCell  
let section = indexPath.section  
let row = indexPath.row  
  
if section < blePeripheral.gattProfile.count {  
if let characteristics = \  

```

```

blePeripheral.gattProfile[section].characteristics {
if row < characteristics.count {
cell.renderCharacteristic(
characteristic: characteristics[row]) }

}
}
return cell

}
/**
Return the number of Service sections

*/
func numberOfSections(in tableView: UITableView) -> Int { print("returning
number of sections")
print(blePeripheral)
print(blePeripheral.gattProfile)
return blePeripheral.gattProfile.count

}

/**
Return the title for a Service section
*/

func tableView(
_ tableView: UITableView,
titleForHeaderInSection section: Int) -> String?

{
print("returning title at section \(section)")
if section < blePeripheral.gattProfile.count {

return blePeripheral.gattProfile[section].uuid.uuidString }
return nil

}

/**

```

User selected a Characteristic table cell. Update UI and open the next UIView  
\*/

```
func tableView(
    _ tableView: UITableView,
    didSelectRowAt indexPath: IndexPath)

{
    let selectedRow = indexPath.row
    print("Selected Row: \(selectedRow)")
    tableView.deselectRow(at: indexPath, animated: true)

}
// MARK: Navigation
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {

print("leaving view - disconnecting from peripheral") if let peripheral =
blePeripheral.peripheral {
centralManager.cancelPeripheralConnection(peripheral) }
}
}
```

The resulting app will be able to connect to a Peripheral and list the Services and Characteristics (Figure 6-5).

●●●○○ Fido 3G

20:37

⌘ 100%  ⚡

[← Peripherals](#) **Peripheral**

MyDevice

-67

501AFEEE-AA49-4AF1-8924-E56689A4I

180A

2A56

**Figure 6-5. App screen showing GATT profile from a connected Peripheral**

# Programming the Peripheral

The Peripheral can be programmed to host a GATT Profile - the tree structure of Services and Characteristics that a connected Central will use to communicate with the Peripheral.

Services are created and added to the `CBPeripheralManager`, like this:

```
// Service UUID
let serviceUuid = CBUUID(string: "0000180c-0000-1000-8000-00805f9b34fb")
let service = CBMutableService(type: serviceUuid, primary: true)
peripheralManager.add(service)
```

When a Service is added to the Peripheral, the `peripheralManager didAdd` callback will be triggered by the `CBPeripheralManagerDelegate`.

```
func peripheralManager(
    _ peripheral: CBPeripheralManager, didAdd service: CBService, error: Error?)
{
}
```

Characteristics must have defined properties. These properties allow a connected Central to read data from, write data to, and/or subscribe to notifications from a Characteristic. Some common properties are enumerated in the `CBCharacteristicProperties` class:

## Table 6-3. Common `CBCharacteristicProperties`

### Value Permission Description

**read** **Read** The characteristic's value can be read.

**write**

### **Write**

The characteristic's value can be written, with a response from the peripheral to indicate that the write was successful.

**writeWithoutResponse**

### **Write**

The characteristic's value can be written, without a response from the peripheral.

**notify Notify** Notifications of the characteristic's value are permitted. Create the Characteristics properties by instantiating and merging CBCharacteristicProperties:

```
// create read Characteristic properties
var characteristicProperties = CBCharacteristicProperties.read // append write
propertie
characteristicProperties.formUnion(CBCharacteristicProperties.write) // append
notify support
characteristicProperties.formUnion(CBCharacteristicProperties.notify)
```

A Characteristic must also define its attribute permissions. An example of an attribute is a flag that is set when a connected Characteristic wants to subscribe to a Characteristic.

Examples of common Attribute Permissions are enumerated in the CBAAttributePermissions class.

### **Table 6-3. Common CBAAttributePermissions**

#### **Value Description**

**readable** The Characteristic's Attributes can be read by a connected Central.

**writable** The Characteristic's Attributes can be altered by a connected Central. Create Attribute permissions.

```
// set the Characteristic's Attribute permissions
var characterisiticPermissions = CBAAttributePermissions.writable // append
permissions
characterisiticPermissions.formUnion(CBAAttributePermissions.readable)
```

Create a new CBMutableCharacteristic with the defined properties. Optionally an initial value can be set.

```
let characteristicUuid = \
```

```
CBUUID(string: "00002a56-0000-1000-8000-00805f9b34fb") var value:Data!
```

```
// instantiate a Characteristic
```

```
let characteristic = CBMutableCharacteristic(
```

```
type: characteristicUuid,
```

```
properties: characteristicProperties, value: value,
```

```
permissions: characterisiticPermissions)
```

Add one or more Characteristics to a Service by creating a [CBMutableCharacteristic] array.

```
// set the service Characteristic array service.characteristics = [ characteristic ]
```

### **A Note on GATT Profile Best Practices**

All Peripherals should contain Device information and a Battery Service, resulting in a minimal GATT profile for any Peripheral that resembles this (Figure 6-6).

## Minimal GATT Profile

Device Info (0x180a)

Device Name (0x2a00)

Model Number (0x2a24)

Serial Number (0x2a25)

Battery level (0x180f)

Battery Level (0x2a19)

Figure 6-6. Minimal GATT Profile for Peripherals

This provides Central software, surveying tools, and future developers to better understand what each Peripheral is, how to interact with it, and what the battery capabilities are.

For pedagogical reasons, many of the examples will not include this portion of the GATT Profile.

## Putting It All Together

Create a new project called GattProfile and copy everything from the previous example.

### Models

Modify BlePeripheral.swift to build a minimal Gatt Services profile. Build the GATT Profile structure, and handle the callback when Services are added.

#### Example 6-5. Models/BlePeripheral.swift

```
... // MARK: GATT Profile

// Service UUID
let serviceUuid = CBUUID(string: "0000180c-0000-1000-8000-00805f9b34fb")
// Characteristic UUIDs
let characteristicUuid = CBUUID(

string: "00002a56-0000-1000-8000-00805f9b34fb")
// Read Characteristic
var characteristic:CBMutableCharacteristic!

...
/**
Build Gatt Profile.
This must be done after Bluetooth Radio has turned on */

func buildGattProfile() {
let service = CBMutableService(type: serviceUuid, primary: true) var
characteristicProperties = CBCharacteristicProperties.read
characteristicProperties.formUnion(

CBCharacteristicProperties.notify)
var characterisitcPermissions = CBAttributePermissions.writable
characterisitcPermissions.formUnion(CBAttributePermissions.readable)
```

```

characteristic = CBMutableCharacteristic( type: characteristicUuid,
properties: characteristicProperties, value: nil,
permissions: characteristicPermissions)

service.characteristics = [ characteristic ] peripheralManager.add(service)

}
...
/**
Peripheral added a new Service
*/
func peripheralManager(
 _ peripheral: CBPeripheralManager, didAdd service: CBService,
error: Error?)
{
print("added service to peripheral") if error != nil {
print(error.debugDescription) }
}

/**
Bluetooth Radio state changed */

func peripheralManagerDidUpdateState( _ peripheral: CBPeripheralManager)

{
peripheralManager = peripheral
switch peripheral.state {
case CBManagerState.poweredOn:

buildGattProfile()
startAdvertising()
default: break
}
delegate?.blePeripheral?(stateChanged: peripheral.state) }
...

```

The resulting app will be able to host a minimal GATT Profile.

## Example code

The code for this chapter is available online  
at: <https://github.com/BluetoothLowEnergyIniOSSwift/Chapter06>

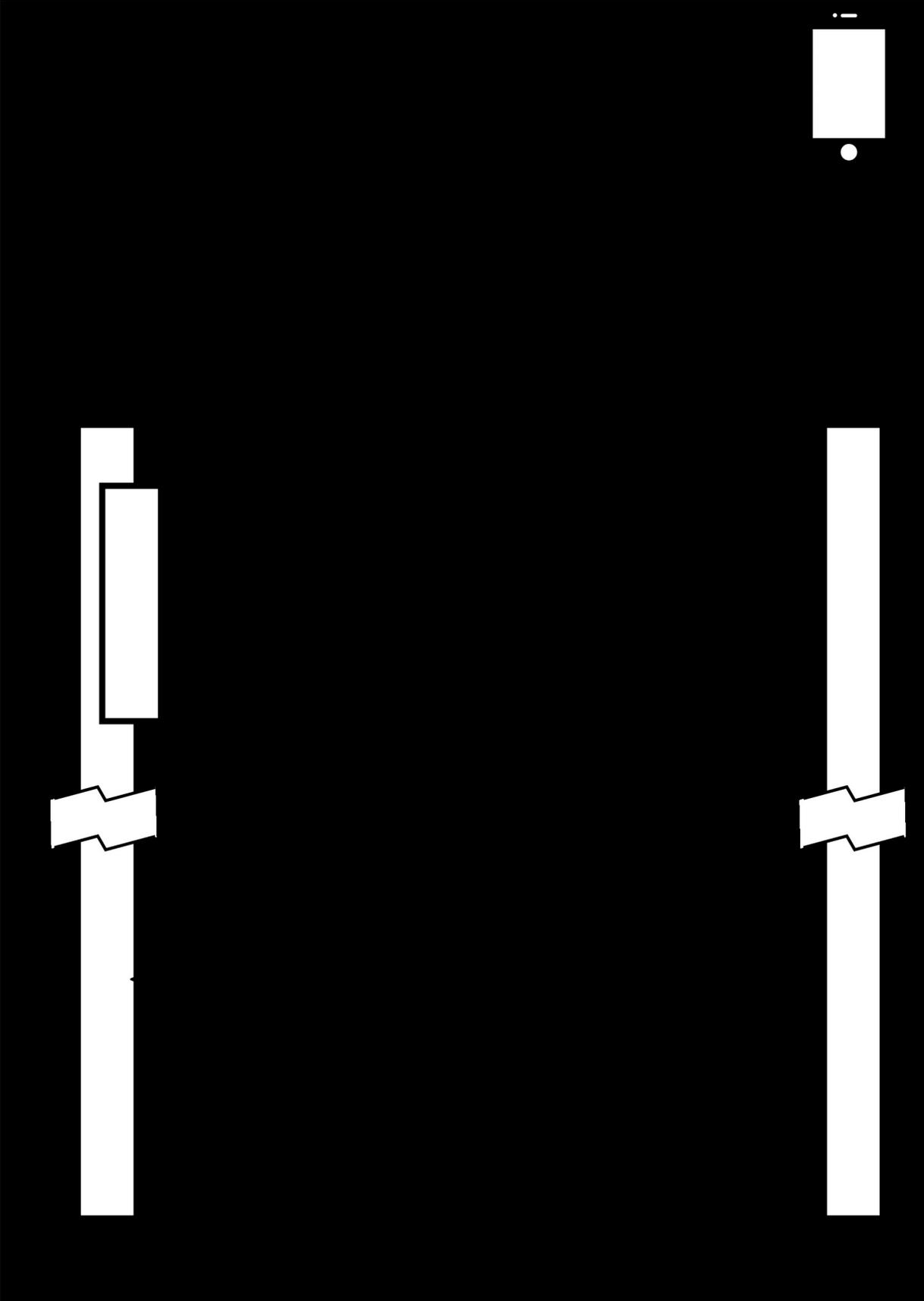
## Reading Data from a Peripheral

The real value of Bluetooth Low Energy is the ability to transmit data wirelessly.

Bluetooth Peripherals are passive, so they don't push data to a connected Central. Instead, Centrals make a request to read data from a Characteristic. This can only happen if the Characteristic enables the Read Attribute.

This is called "reading a value from a Characteristic."

Therefore, if a Peripheral changes the value of a Characteristic, then later a Central downloads data from the Peripheral, the process looks like this (Figure 7-1):



**Figure 7-1. The process of a Central reading data from a Peripheral**  
A Central can read a Characteristic repeatedly, regardless if Characteristic's value has changed.

## Programming the Central

Before reading data from a connected Peripheral, it may be useful to know if a Characteristic provides read permission. Read permission can be read by getting the Characteristic property bit map and isolating the read property from it, like this:

```
let isReadable = (characteristic.properties.rawValue & \
CBCharacteristicProperties.read.rawValue) != 0
```

Once the Central has a Bluetooth GATT connection and has access to a Characteristic with which to communicate with a connected Peripheral, the Central can request to read data from that Characteristic like this:

```
peripheral.readValue(for: characteristic)
This will initiate a read request from the Central to the Peripheral.
```

When the Central finishes reading data from the Peripheral's Characteristic, the peripheral `didUpdateValueFor` method is triggered in the `CBPeripheralManagerDelegate`.

In this callback, the Characteristic's value can read as a `Data` object using the `characteristic.value` property.

```
func peripheral(
    _ peripheral: CBPeripheral,
    didUpdateValueFor characteristic: CBCharacteristic, error: Error?)
{
    let value = characteristic.value
}
```

From here the data can be converted into any format, including a `String` or an `Integer`.

```
// convert to byte array
let byteArray = [UInt8](value)
// convert to String
let stringValue = String(data: value, encoding: .ascii)
```

```
// convert to signed integer
let intValue = value.withUnsafeBytes { (ptr: UnsafePointer<Int>) -> Int in
return ptr.pointee
}
```

```
// convert to float
let floatValue = value.withUnsafeBytes { (ptr: UnsafePointer<Float>) -> Float in
return ptr.pointee
}
```

## **Putting It All Together**

Create a new project with the following structure (Figure 7-2).



**Figure 7-2.**

## **Added project files**

### **Models**

Modify the `BlePeripheral` class to include a method that checks if a `Characteristic` is readable, one that initiates a read request from a `Characteristic`, and one that responds to the resulting callback.

## Example 7-1. Models/BlePeripheral.swift

```
...
/**
Read from a Characteristic */
func readValue(from characteristic: CBCharacteristic) {
self.peripheral.readValue(for: characteristic) }
/**
Check if Characteristic is readable
- Parameters:
- characteristic: The Characteristic to test
- returns: True if characteristic is readable */
static func isCharacteristic(
isReadable characteristic: CBCharacteristic) -< Bool {
if (characteristic.properties.rawValue & \

CBCharacteristicProperties.read.rawValue) != 0 {
return true }
return false }

// MARK: CBPeripheralDelegate

/**
Value downloaded from Characteristic on connected Peripheral */

func peripheral(
_ peripheral: CBPeripheral,
didUpdateValueFor characteristic: CBCharacteristic, error: Error?)

{
print("characteristic updated") if let value = characteristic.value {
print(value.debugDescription) print(value.description)

if let stringValue = String(data: value, encoding: .ascii) { print(stringValue)
// received response from Peripheral
delegate?.blePeripheral?(

characteristicRead: stringValue,
characteristic: characteristic,
blePeripheral: self)
```

```
}  
}  
}  
}
```

## Delegates

Add a method to the BlePeripheralDelegate that alerts subscribers of a read operation on a Characteristic.

### Example 7-2. Delegates/BlePeripheralDelegate.swift

```
... /**
```

```
Characteristic was read
```

- Parameters:
- stringValue: the value read from the Characteristic
- characteristic: the Characteristic that was read
- blePeripheral: the BlePeripheral

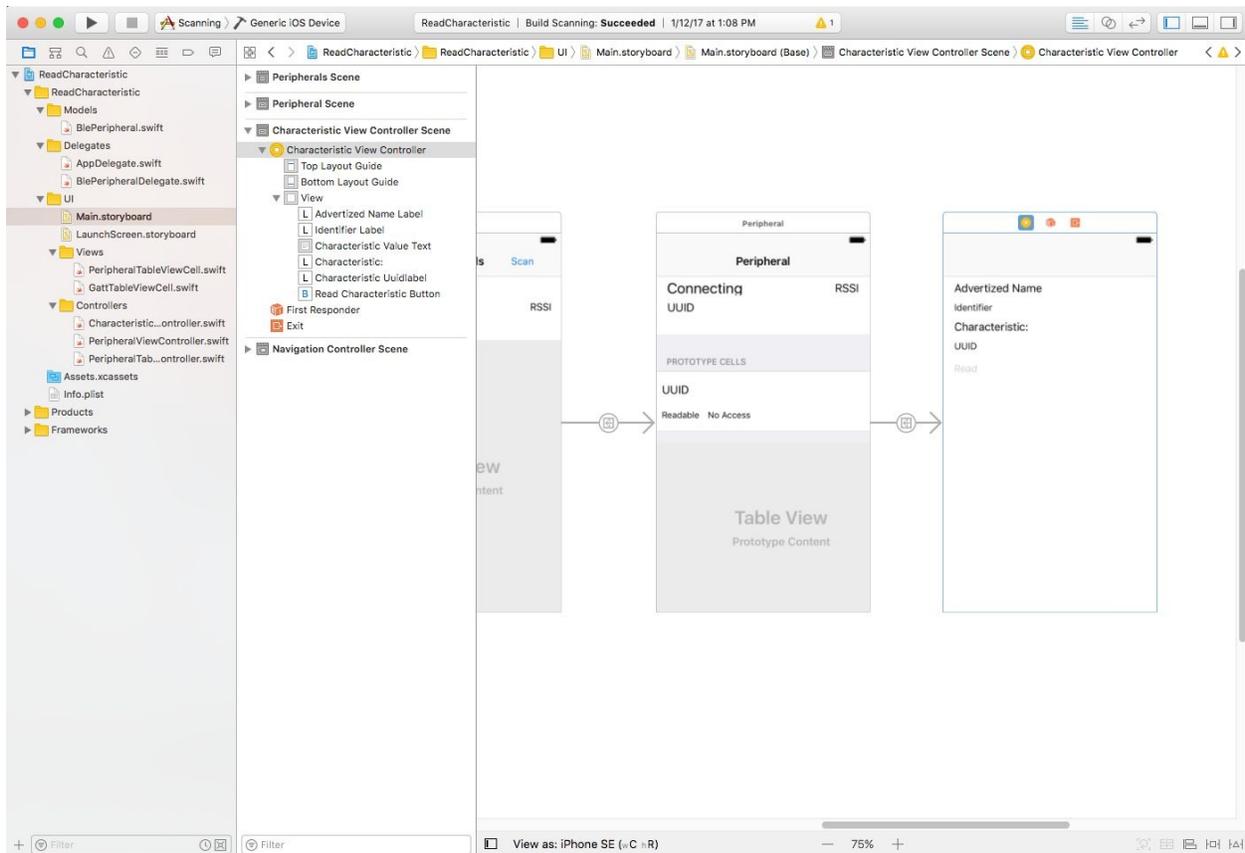
```
*/
```

```
@objc optional func blePeripheral(  
characteristicRead stringValue: String, characteristic: CBCharacteristic,  
blePeripheral: BlePeripheral)
```

```
...
```

## Storyboard

Create a new UIView, with class name "CharacteristicViewController" and a new segue to it from the PeripheralViewController. Create and link three UILabel in the GattTableViewCell to show the Characteristic UUID and read/no access properties. Create and link three UILabels to show the Peripheral and Characteristic identifiers, plus a UIButton and UITextView to allow the user to trigger a Characteristic read and display the result on screen (Figure 7-3):



**Figure 7-3. Project Storyboard Views**

Alter the `renderCharacteristic` method in the `GattTableViewCell` class to display the appropriate `UILabel`Views when the `Characteristic` is readable or not

**Example 7-3. UI/Views/GattTableViewCell.swift**

```
import UIKit
import CoreBluetooth
class GattTableViewCell: UITableViewCell {

// MARK: UI Elements
@IBOutlet weak var uuidLabel: UILabel! @IBOutlet weak var readableLabel:
UILabel! @IBOutlet weak var noAccessLabel: UILabel!

/**
Render the cell with Characteristic properties
*/

func renderCharacteristic(characteristic: CBCharacteristic) { uuidLabel.text =
characteristic.uuid.uuidString
let isReadable = \
```

```

BlePeripheral.isCharacteristic(isReadable: characteristic)
readableLabel.isHidden = !isReadable
if isReadable {

noAccessLabel.isHidden = true
} else {
noAccessLabel.isHidden = false
}
}
}
}

```

## Controllers

Create a new view controller, `CharacteristicViewController` that will interact with a selected Characteristic. It will display the properties of the Characteristic and allow the user to trigger a read event on the `BlePeripheral`. The Characteristic's value will be displayed in a `UITextView`.

### Example 7-4. UI/Controllers/CharacteristicViewController.swift

```

import UIKit
import CoreBluetooth
class CharacteristicViewController: UIViewController, \
CBCentralManagerDelegate, BlePeripheralDelegate {

// MARK: UI elements
@IBOutlet weak var advertizedNameLabel: UILabel! @IBOutlet weak var
identifierLabel: UILabel!
@IBOutlet weak var characteristicUuidlabel: UILabel! @IBOutlet weak var
readCharacteristicButton: UIButton! @IBOutlet weak var
characteristicValueText: UITextView!

// MARK: Connected devices

// Central Bluetooth Radio
var centralManager:CBCentralManager! // Bluetooth Peripheral
var blePeripheral:BlePeripheral!
// Connected Characteristic
var connectedService:CBService!
// Connected Characteristic
var connectedCharacteristic:CBCharacteristic!

```

```

/**
UIView loaded */
override func viewDidLoad() {

super.viewDidLoad()
print("Will connect to device " + \

"\(blePeripheral.peripheral.identifier.uuidString)") print("Will connect to
characteristic " + \
"\(connectedCharacteristic.uuid.uuidString)") centralManager.delegate = self
blePeripheral.delegate = self
loadUI()
}

/**
Load UI elements
*/

func loadUI() {
advertizedNameLabel.text = blePeripheral.advertisedName identifierLabel.text = \

blePeripheral.peripheral.identifier.uuidString characteristicUuidlabel.text = \
connectedCharacteristic.uuid.uuidString
readCharacteristicButton.isEnabled = true
// characteristic is not readable
if !BlePeripheral.isCharacteristic(
isReadable: connectedCharacteristic) {
readCharacteristicButton.isHidden = true
characteristicValueText.isHidden = true
}
}

/**
User touched Read button. Request to read the Characteristic */

@IBAction func onReadCharacteristicButtonTouched(_ sender: UIButton) {
print("pressed button")
readCharacteristicButton.isEnabled = false
blePeripheral.readValue(from: connectedCharacteristic) // MARK:

```

## BlePeripheralDelegate

```
/**
Characteristic was read. Update UI */

func blePeripheral(
characteristicRead stringValue: String, characteristic: CBCharacteristic,
blePeripheral: BlePeripheral)

{
print(stringValue)
readCharacteristicButton.isEnabled = true
characteristicValueText.insertText(stringValue + "\n") let stringLength =
characteristicValueText.text.characters.count
characteristicValueText.scrollToVisible(NSMakeRange(
stringValueLength-1, 0))
}
// MARK: CBCentralManagerDelegate
/**
Peripheral disconnected

- Parameters:
- central: the reference to the central
- peripheral: the connected Peripheral

*/
func centralManager(
_ central: CBCentralManager,
didDisconnectPeripheral peripheral: CBPeripheral, error: Error?)

{
// disconnected. Leave print("disconnected")

if let navController = navigationController {
navController.popToRootViewController(animated: true) dismiss(animated: true,
completion: nil)

}
}
```

```

/**
Bluetooth radio state changed

- Parameters:
- central: the reference to the central
*/

func centralManagerDidUpdateState(_ central: CBCentralManager) {
print("Central Manager updated: checking state") switch (central.state) {
case .poweredOn:

print("bluetooth on")
default:
print("bluetooth unavailable")
}
}

// MARK: - Navigation

/**
Animate the segue
*/

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

centralManager.cancelPeripheralConnection(
connectedBlePeripheral.peripheral)
}
}

```

Modify the PeripheralViewController to launch the CharacteristicViewCnotroller when a table cell is clicked. Add the corresponding UITableViewDelegate methods and segue.

**Example 7-5. [swift/example.com.exampleble/ConnectActivity.swift](https://github.com/example.com/exampleble/ConnectActivity.swift)**

```

... /**
User selected a Characteristic table cell. Update UI and open the next UIView
*/

func tableView(
    _ tableView: UITableView,
    didSelectRowAt indexPath: IndexPath)

{
    let selectedRow = indexPath.row print("Selected Row: \(selectedRow)")
}
// MARK: Navigation

/**
Handle the Segue. Prepare the next UIView with necessary information */

override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    print("leaving view - disconnecting from peripheral")

    if let indexPath = gattTableView.indexPathForSelectedRow { let selectedSection
    = indexPath.section
    let selectedRow = indexPath.row
    let characteristicViewController = \

    segue.destination as! CharacteristicViewController if selectedSection <
    blePeripheral.gattProfile.count { let service =
    blePeripheral.gattProfile[selectedSection] if let characteristics = \

    blePeripheral.gattProfile[selectedSection].\ characteristics

    {
    if selectedRow < characteristics.count { // populate next UIView with necessary
    information characteristicViewController.centralManager = \

    centralManager
    characteristicViewController.blePeripheral = \
    blePeripheral
    characteristicViewController.connectedService = \
    service

```

```

characteristicViewController.\
connectedCharacteristic = \
characteristics[selectedRow]
}
}
}
gattTableView.deselectRow(at: indexPath, animated: true) } else {
if let peripheral = blePeripheral.peripheral {
centralManager.cancelPeripheralConnection(peripheral) }
}
}
}

```

When run, the App will be able to scan for and connect to a Peripheral. Once connected, it can list the GATT Profile - the Services and Characteristics hosted on the Peripheral. A Characteristic can be selected and values can be read from that Characteristic (Figure 7-4).

●●●○○ Fido 3G

20:12

⌘ 99%  ⚡

[← Peripherals](#) **Peripheral**

MyDevice

-75

501AFEEE-AA49-4AF1-8924-E56689A4I

180A

2A56

Readable

●●●○○ Fido 3G

20:13

⌘ 99%  ⚡

[< Peripheral](#)

MyDevice

501AFEEE-AA49-4AF1-8924-E56689A4F0A8

Characteristic:

2A56

[Read](#)

CURWWY6

**Figure 7-4. App screens showing GATT Profile for connected Peripheral and values read from a Characteristic on a connected Peripheral**

## Programming the Peripheral

This chapter will show how to create a Characteristic with read access. A read-only Characteristic is created and added to a Primary Service like this:

```
// create Characteristic UUID
let readCharacteristicUuid = CBUUID(
string: "00002a56-0000-1000-8000-00805f9b34fb")

// Make Characteristic readable
let characteristicProperties = CBCharacteristicProperties.read // Make Attributes
readable
let characterisitcPermissions = CBAttributePermissions.readable // create the
readable Characteristic
let readCharacteristic = CBMutableCharacteristic(

type: readCharacteristicUuid, properties: characteristicProperties, value: nil,
permissions: characterisitcPermissions)

// set the Characteristic as the only one in the Service
service.characteristics = [ readCharacteristic ]
```

When the Central requests to read data from the Peripheral's Characteristic, the peripheralManager didReceiveRead method is triggered the CBPeripheralManagerDelegate object.

```
func peripheralManager(
_peripheral: CBPeripheralManager, didReceiveRead request: CBATTRequest)
{
}
```

The Central cannot read the Characteristic value until the request's value is set:

```
func peripheralManager(
_peripheral: CBPeripheralManager, didReceiveRead request: CBATTRequest)
{
let characteristic = request.characteristic
if let value = characteristic.value {
```

```
// Respond to the Central with the Characteristic value let range =
Range(uncheckedBounds: (
lower: request.offset,
upper: value.count - request.offset)) request.value = value.subdata(in: range) }
}
```

The Central needs to know if the request was successful. This is done by responding to the with a `CBATTErrors` status.

`CBATTErrors` status messages include the following:

### **Table 7-1. Common `CBATTErrors` values**

#### **Value Description**

**success** Operation was successful

**readNotPermitted** Read request not permitted.

**requestNotSupported** Request was not supported.

**invalidOffset** Requested data offset is invalid.

For example, a successful read request warrants a `CBATTErrors.success` response:

```
... peripheral.respond(to: request, withResult: CBATTErrors.success)
...
```

A full implementation looks like this:

```
func peripheralManager(
_ peripheral: CBPeripheralManager, didReceiveRead request: CBATTRequest)
{
let characteristic = request.characteristic
if let value = characteristic.value {

// if the requested offset is
// larger than the Characteristic value, send an error if request.offset >
value.count {

peripheralManager.respond(
to: request,
withResult: CBATTErrors.invalidOffset)

return
}
```

```
// Respond to the Central with the Characteristic value let range =  
Range(uncheckedBounds: (  
  
  lower: request.offset,  
  upper: value.count - request.offset))  
request.value = value.subdata(in: range)  
peripheral.respond(to: request, withResult: CBALError.success) }  
}
```

## Putting It All Together

Copy the previous chapter's project into a new project and modify the files below.

### Models

Modify the `BlePeripheral` class to include build a readable `Characteristic`, sets a random `String` to the `Characteristic` every 5 seconds, and responds when the `Central` initiates a read request:

#### Example 7-6. Models/`BlePeripheral.swift`

```
... // MARK: GATT Profile

// Service UUID
let serviceUuid = CBUUID(string: "0000180c-0000-1000-8000-00805f9b34fb")
// Characteristic UUIDs
let readCharacteristicUuid = CBUUID(

string: "00002a56-0000-1000-8000-00805f9b34fb")
// Read Characteristic
var readCharacteristic:CBMutableCharacteristic!

...

/**
Build Gatt Profile.
This must be done after Bluetooth Radio has turned on
*/

func buildGattProfile() {
let service = CBMutableService(type: serviceUuid, primary: true) let
characteristicProperties = CBCharacteristicProperties.read let
characteristicPermissions = CBAttributePermissions.readable readCharacteristic
= CBUUID(

type: readCharacteristicUuid,
properties: characteristicProperties,
value: nil, permissions: characteristicPermissions)
```

```

service.characteristics = [ readCharacteristic ]
peripheralManager.add(service)
}

/**
Set a Characteristic to some text value */

func setCharacteristicValue(
_ characteristic: CBMutableCharacteristic, value: Data

) { characteristic.value = value
if central != nil {

peripheralManager.updateValue( value,
for: readCharacteristic, onSubscribedCentrals: [central])

}
}

...
/**
Connected Central requested to read from a Characteristic */

func peripheralManager(
_ peripheral: CBPeripheralManager,
didReceiveRead request: CBATTRequest)

{
let characteristic = request.characteristic
if let value = characteristic.value {

//let stringValue = String(data: value, encoding: .utf8)! if request.offset >
value.count {
peripheralManager.respond(

to: request, withResult: CBATTError.invalidOffset) return
}
let range = Range(uncheckedBounds: (

lower: request.offset,

```

```

upper: value.count - request.offset))
request.value = value.subdata(in: range)
peripheral.respond(to: request, withResult: CBATTError.success) }
delegate?.blePeripheral?(characteristicRead: request.characteristic) }
...

```

## Delegates

Add a method to the BlePeripheralDelegate that sends a notification when a Characteristic has been read:

### Example 7-7. Delegates/BlePeripheralDelegate.swift

```

... /**
Characteristic was read

```

- Parameters:
- characteristic: the Characteristic that was read \*/

```

@objc optional func blePeripheral(
characteristicRead fromCharacteristic: CBCharacteristic) ...

```

## Controllers

Add a method to the BlePeripheralDelegate that sends a notification when a Characteristic has been read:

### Example 7-7. UI/Controllers/ViewController.swift

```

... @IBOutlet weak var characteristicValueTextField: UITextField!
...

```

```

// MARK: Update BlePeripheral Properties
/**

```

Generate a random String

- Parameters
- length: the length of the resulting string
- returns: random alphanumeric string
- \*/

```

func randomString(length: Int) -> String {
let letters : NSString = "abcdefghijklmnopqrstuvwxyz" + \

```

```

"ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789" let len =
UInt32(letters.length)

```

```

var randomString = ""
for _ in 0 ..< length {

let rand = arc4random_uniform(len)
var nextChar = letters.character(at: Int(rand)) randomString += \

NSString(characters: &nextChar, length: 1) as String }
return randomString

```

```

}

```

```

/**

```

```

Set Read Characteristic to some random text value */

```

```

func setRandomCharacteristicValue() {

```

```

let stringValue = randomString(
length: Int(arc4random_uniform(

```

```

UInt32(blePeripheral.readCharacteristicLength - 1)) )
)

```

```

let value:Data = stringValue.data(using: .utf8)!
blePeripheral.setCharacteristicValue(

```

```

blePeripheral.readCharacteristic,
value: value
)

```

```

characteristicValueTextField.text = stringValue }

```

```

...

```

```

/** BlePeripheral started advertising

```

```

- Parameters:

```

```

- error: the error message, if any

```

```

*/

```

```

func blePeripheral(startedAdvertising error: Error?) {

```

```

if error != nil {

```

```

print("Problem starting advertising: " + error.debugDescription)

```

```

} else {

```

```

print("advertising started")

```

```
advertisingSwitch.setOn(true, animated: true)
setRandomCharacteristicValue()
randomTextTimer = Timer.scheduledTimer(

timeInterval: 5,
target: self,
selector: #selector(setRandomCharacteristicValue), userInfo: nil,
repeats: true

)
}
}
```

...

When run, the App will be able to host a simple GATT profile with a single Characteristic that sets the Characteristic to a random String every 5 seconds (Figure 7-5).

••••• Fido 3G

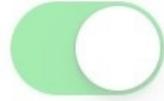
14:05

98%  

## Bluetooth Peripheral

Advertising Name: MyDevice

Advertising



Characteristic Value:

dlnIRRYrn

**Figure 7-5. App screen showing random Characteristic value on Advertising Peripheral**

## Example code

The code for this chapter is available online  
at: <https://github.com/BluetoothLowEnergyIniOSSwift/Chapter07>

## **Writing Data to a Peripheral**

Data is sent from the Central to a Peripheral when the Central writes a value in a Characteristic hosted on the Peripheral, presuming that Characteristic has write permissions.

The process looks like this (Figure 8-1):



**Figure 8-1. The process of a Central writing data to a Peripheral**

# Programming the Central

Before attempting to write data a Characteristic, it is useful to know if the Characteristic has “write” permissions. Write permissions can be read by accessing a Characteristic’s properties and isolating the Write properties:

```
// Characteristic supports write permissions
let properties = characteristic.properties.rawValue;
let isWritable = (properties & \
    CBCharacteristicProperties.write.rawValue) != 0 ||
(properties & \
    CBCharacteristicProperties.writeWithoutResponse.rawValue) != 0
```

```
// Characteristic supports write (with response) permissions let
isWritableWithResponse = (properties & \
    CBCharacteristicProperties.write.rawValue) != 0
// Characteristic supports write without response permissions let
isWritableWithoutResponse = (properties & \
```

```
    CBCharacteristicProperties.writeWithoutResponse.rawValue) != 0 The
Characteristic is written to like this:
peripheral.writeValue(value, for: characteristic)
```

Regardless of the initial data type, the value written to the Characteristic must be sent as a Data object. Here is how to convert some common data types into a Data object:

```
// convert String to Data
let stringValue = "Hello"
let arrayValue = Array(stringValue.utf8)
let stringDataValue = Data(Array(byteValue[0..
```

```
// convert Int to Data
var intValue = 8;
let intDataValue = Data(buffer: UnsafeBufferPointer(
start: &intValue,
```

```

count: 1))

// convert Float to Data
var floatValue = 10.2;
let floatDataValue = Data(buffer: UnsafeBufferPointer(

start: &floatValue,
count: 1))

```

If the Characteristic supports write (with response), the peripheral `didWriteValueFor` event gets triggered in the `CBPeripheralDelegate` following a write operation:

```

func peripheral(
    _ peripheral: CBPeripheral,
    didWriteValueFor characteristic: CBCharacteristic, error: Error?)

{
}

```

## Putting It All Together

Copy the project from the previous chapter into a new project, titled "WriteCharacteristic."

### Models

Modify the `BlePeripheral` class to include methods to test the write permissions of a Characteristic, to write a value to a Characteristic, and to handle the resulting `CBPeripheralDelegate` callback:

#### Example 8-1. Models/BlePeripheral.swift

```
... /**
```

Write a text value to the `BlePeripheral`

- Parameters:
- `value`: the value to write to the connected Characteristic \*/

```

func writeValue(value: String, to characteristic: CBCharacteristic) { let
byteValue = Array(value.utf8)
// cap the outbound value length to be
// less than the characteristic length

```

```

var length = byteValue.count
if length > characteristicLength {

length = characteristicLength
}
let transmissibleValue = Data(Array(byteValue[0..<length]))
print(transmissibleValue)
var writeType = CBCharacteristicWriteType.withResponse if
BlePeripheral.isCharacteristic(

isWritableWithoutResponse: characteristic) { writeType =
CBCharacteristicWriteType.withoutResponse }
peripheral.writeValue(
transmissibleValue,
for: characteristic,
type: writeType)
print("write request sent")
}

/**
Check if Characteristic is writable
- Parameters:
- characteristic: The Characteristic to test
- returns: True if characteristic is writable */
static func isCharacteristic(
isWritable characteristic: CBCharacteristic) -> Bool

{
if (characteristic.properties.rawValue & \
CBCharacteristicProperties.write.rawValue) != 0 ||
(characteristic.properties.rawValue & \
CBCharacteristicProperties.writeWithoutResponse.rawValue) != 0 { return true

}
return false }
/**
Check if Characteristic is writable with response
- Parameters:
- characteristic: The Characteristic to test

```

- returns: True if characteristic is writeable with response

```
*/
```

```
static func isCharacteristic(  
isWritableWithResponse characteristic: CBCharacteristic) -> Bool { if  
(characteristic.properties.rawValue & \
```

```
CBCharacteristicProperties.write.rawValue) != 0 { return true
```

```
}
```

```
return false
```

```
}
```

```
/**
```

Check if Characteristic is writeable without response

- Parameters:

- characteristic: The Characteristic to test

- returns: True if characteristic is writeable without response \*/

```
static func isCharacteristic(  
isWritableWithoutResponse characteristic: CBCharacteristic) -> Bool
```

```
{
```

```
if (characteristic.properties.rawValue & \
```

```
CBCharacteristicProperties.writeWithoutResponse.rawValue) != 0 { return true
```

```
}
```

```
return false
```

```
}
```

```
// MARK: CBPeripheralDelegate
```

```
/**
```

Value was written to the Characteristic

```
*/
```

```
func peripheral(  
_ peripheral: CBPeripheral,
```

```
didWriteValueFor characteristic: CBCharacteristic, error: Error?)
```

```
{
```

```
print("data written")
```

```
delegate?.blePeripheral?(
```

```
valueWritten: characteristic, blePeripheral: self)
}
...
```

## Delegates

Add a function to the BlePeripheralDelegate to alert subscribers that a write operation happened.

### Example 8-2. Delegates/BlePeripheralDelegate.swift

```
... /**
```

```
Value written to Characteristic
```

```
- Parameters:
```

```
- characteristic: the Characteristic that was written to
```

```
- blePeripheral: the BlePeripheral
```

```
*/
```

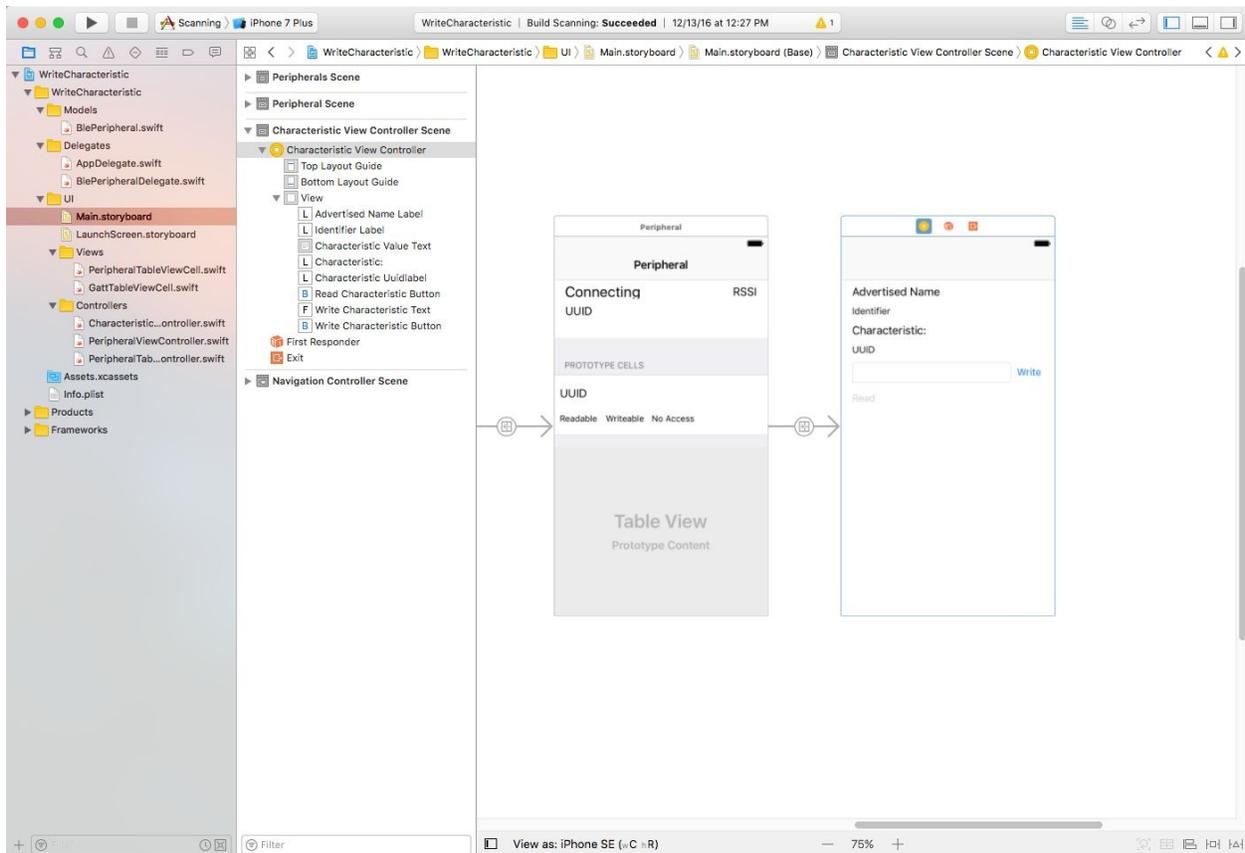
```
@objc optional func blePeripheral(
```

```
valueWritten characteristic: CBCharacteristic, blePeripheral: BlePeripheral)
```

```
...
```

## Storyboard

Create and link a UILabel in the GattTableViewCell to show the write property and a UITextField and UIButton to allow a user to type and submit text to the Characteristic (Figure 8-2):



**Figure 8-2. Project Storyboard Views**

Modify the `GattTableViewCell` to display the appropriate `UILabelView` when a `Characteristic` is writeable.

**Example 8-3. UI/Views/GattTableViewCell.swift**

```

... @IBOutlet weak var writeableLabel: UILabel!
...
func renderCharacteristic(characteristic: CBCharacteristic) { uuidLabel.text =
characteristic.uuid.uuidString let isReadable = BlePeripheral.isCharacteristic(
isReadable: characteristic)
let isWritable = BlePeripheral.isCharacteristic( isWritable: characteristic)
readableLabel.isHidden = !isReadable
writeableLabel.isHidden = !isWritable
if isReadable || isWritable {
noAccessLabel.isHidden = true
} else {
noAccessLabel.isHidden = false
}
}

```

```
}  
...
```

## Controllers

Add functionality to the `CharacteristicViewController` to display new UI elements to handle write operations, as well as an implementation of the new `BlePeripheralDelegate` method listed above.

### Example 8-4. `UI/Controllers/CharacteristicViewController.swift`

```
... @IBOutlet weak var writeCharacteristicButton: UIButton! @IBOutlet weak  
var writeCharacteristicText: UITextField!
```

```
...  
/**
```

```
Load UI elements  
*/
```

```
func loadUI() {  
    advertisedNameLabel.text = blePeripheral.advertisedName identifierLabel.text =  
    \
```

```
blePeripheral.peripheral.identifier.uuidString characteristicUuidlabel.text = \  
connectedCharacteristic.uuid.uuidString readCharacteristicButton.isEnabled =  
true
```

```
// characteristic is not readable  
if !BlePeripheral.isCharacteristic(  
isReadable: connectedCharacteristic) {  
    readCharacteristicButton.isHidden = true  
    characteristicValueText.isHidden = true }  
// characteristic is not writeable
```

```
if !BlePeripheral.isCharacteristic(  
isWritable: connectedCharacteristic) {  
    writeCharacteristicText.isHidden = true  
    writeCharacteristicButton.isHidden = true }  
}
```

```
...  
/**
```

```
User touched Read button. Request to write to the Characteristic */
```

```

@IBAction func onWriteCharacteristicButtonTouched(_ sender: UIButton) {
    print("write button pressed")
    writeCharacteristicButton.isEnabled = false
    if let stringValue = writeCharacteristicText.text {
        print(stringValue)
        blePeripheral.writeValue(
            value: stringValue,
            to: connectedCharacteristic) writeCharacteristicText.text = "" }
    }
    ...
/**
Characteristic was written to. Update UI */
func blePeripheral(
    valueWritten characteristic: CBCharacteristic, blePeripheral: BlePeripheral)
    {
    print("value written to characteristic!") writeCharacteristicButton.isEnabled =
    true }
    ...

```

Compile and run. The updated app will be able to scan for and connect to a Peripheral. Once connected, it lists services and characteristics, connect to Characteristics and send and data (Figure 8-3).

●●●○○ Fido 3G

20:37

⌘ 100%  ⚡

[← Peripherals](#) **Peripheral**

MyDevice

-67

501AFEEE-AA49-4AF1-8924-E56689A4I

180A

2A56

Writeable



**Figure 8-3. App screens showing GATT Profile of connected Peripheral with Write access to a Characteristic, and an interface to send text to a Characteristic**

# Programming the Peripheral

This chapter will show how to create a Characteristic with read access.

A writeable Characteristic is has the `CBCharacteristicProperties.write` property:

```
// Characteristic is writable
```

```
var characteristicProperties = CBCharacteristicProperties.write
```

It is possible to append other properties to additionally make the Characteristic readable:

```
// Read support
```

```
characteristicProperties.formUnion(CBCharacteristicProperties.read)
```

```
// add write support
```

```
characteristicProperties.formUnion(CBCharacteristicProperties.notify)
```

Everything else about creating the Characteristic remains the same as for a readable Characteristic:

```
// Characteristic Attributes are readable
```

```
var characterisitcPermissions = CBAttributePermissions.readable
```

```
// Set Characteristic UUID
```

```
let readWriteCharacteristicUuid = CBUUID( string: "00002a56-0000-1000-8000-00805f9b34fb")
```

```
// Create a writeable Characteristic
```

```
readWriteCharacteristic = CBMutableCharacteristic( type:
```

```
readWriteCharacteristicUuid,
```

```
properties: characteristicProperties,
```

```
value: nil,
```

```
permissions: characterisitcPermissions)
```

```
// Set the Characteristic as belonging to the Service service.characteristics = [  
readWriteCharacteristic ]
```

When a connected Central attempts to write to a Characteristic, the `peripheralManager didReceiveWrite` callback is triggered by the `CBPeripheralManagerDelegate` object. If the Characteristic has the `CBCharacteristicProperties.write` property, it is important to respond to the Central with a `CBATTError` status.

`CBATTError` status messages include the following:

## Table 8-1. Common `CBATTError` values

## Value Description

**success** Operation was successful

**writeNotPermitted** Write request was not permitted.

**readNotPermitted** Read request not permitted.

**requestNotSupported** Request was not supported.

**invalidOffset** Requested data offset is invalid.

More than one request may come in at a time, so it is useful to iterate through and deal with each request separately.

```
func peripheralManager(
    _ peripheral: CBPeripheralManager, didReceiveWrite requests:
    [CBATTRequest])

{
    for request in requests {
        peripheral.respond(to: request, withResult: CATTError.success) // do
        something with the incoming data
    }
}
```

## Putting It All Together

Create a new project called WriteCharacteristic and copy the files from the previous chapter's project.

The Peripheral will be modified to support writes.

### Models

Modify the BlePeripheral class to create a writable Characteristic and to handle incoming write requests:

#### Example 8-5. Models/BlePeripheral.swift

```
... // Service UUID
let serviceUuid = CBUUID(string: "0000180c-0000-1000-8000-00805f9b34fb")
// Characteristic UUIDs
let readWriteCharacteristicUuid = CBUUID(
    string: "00002a56-0000-1000-8000-00805f9b34fb")
// Read Characteristic
```

```
var readWriteCharacteristic:CBMutableCharacteristic!
```

```
...
```

```
/**
```

```
Build Gatt Profile.
```

```
This must be done after Bluetooth Radio has turned on
```

```
*/
```

```
func buildGattProfile() {
```

```
let service = CBMutableService(type: serviceUuid, primary: true) var
```

```
characteristicProperties = CBCharacteristicProperties.read
```

```
characteristicProperties.formUnion(
```

```
CBCharacteristicProperties.notify)
```

```
var characterisitcPermissions = CBAAttributePermissions.writeable
```

```
characterisitcPermissions.formUnion(CBAAttributePermissions.readable)
```

```
readWriteCharacteristic = CBMutableCharacteristic(
```

```
type: readWriteCharacteristicUuid,
```

```
properties: characteristicProperties, value: nil,
```

```
permissions: characterisitcPermissions)
```

```
service.characteristics = [ readWriteCharacteristic ]
```

```
peripheralManager.add(service)
```

```
}
```

```
...
```

```
/**
```

```
Connected Central requested to write to a Characteristic */
```

```
func peripheralManager(
```

```
_ peripheral: CBPeripheralManager,
```

```
didReceiveWrite requests: [CBATTRequest])
```

```
{
```

```
for request in requests {
```

```
peripheral.respond(to: request, withResult: CBATTErrror.success) if let value =
```

```
request.value {
```

```
delegate?.blePeripheral?(
```

```
valueWritten: value,
```

```
toCharacteristic: request.characteristic) }
```

```
}
```

```
}  
...
```

## Delegates

Add a method to the BlePeripheralDelegate to process successful writes to a Characteristic:

### Example 8-6. Delegates/BlePeripheralDelegate.swift

```
... /** Value written to Characteristic
```

- Parameters:

- value: the Data value written to the Characteristic
- characteristic: the Characteristic that was written to

```
*/
```

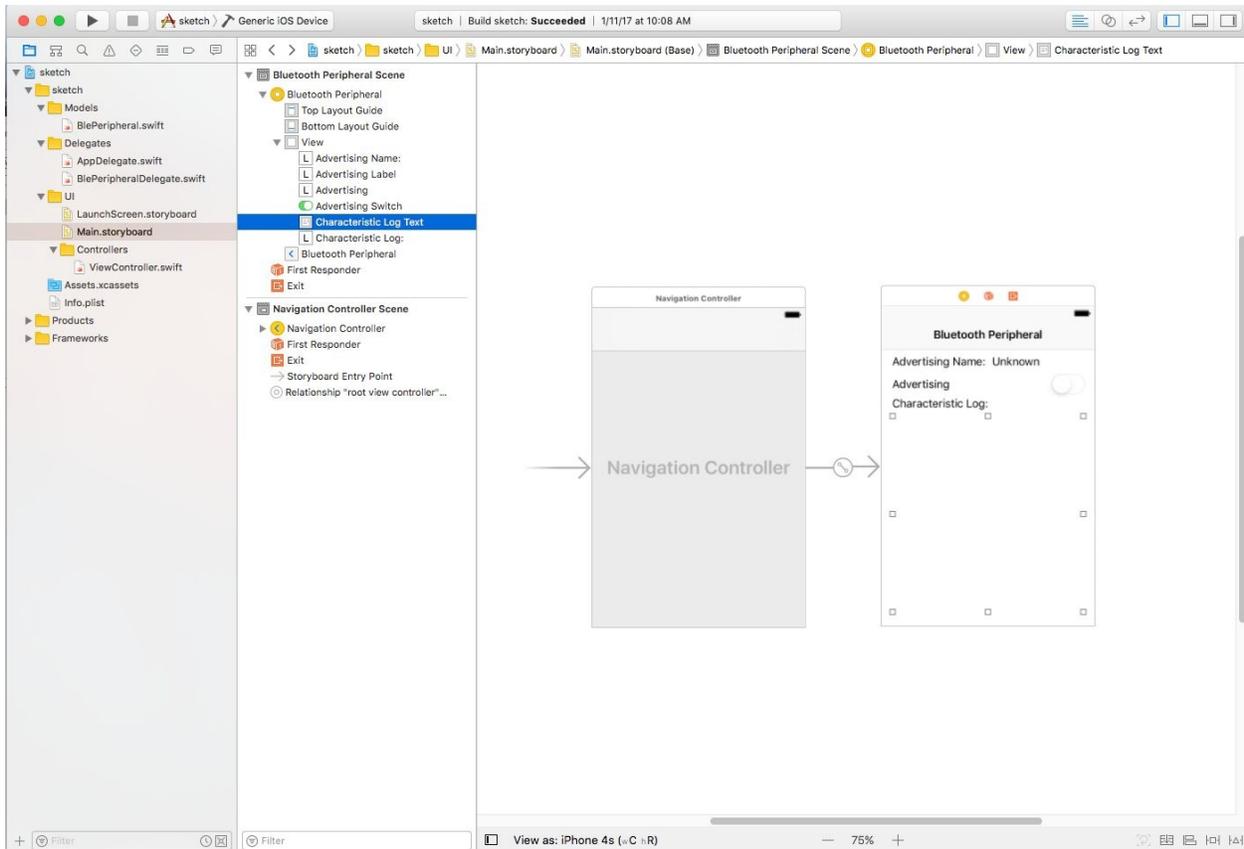
```
@objc optional func blePeripheral(  
valueWritten value: Data,
```

```
toCharacteristic: CBCharacteristic)
```

```
...
```

## Storyboard

Add a UILabel and UITextView to show the Characteristic Log in the UIView in the Main.storyboard to create the App's user interface (Figure 8-4):



**Figure 8-4. Project Storyboard Controllers**

Add a UITextView that logs the incoming Characteristic values, and a callback handler for the the BlePeripheral writes:

**Example 8-7. UI/Controllers/ViewController.swift**

```
...
@IBOutlet weak var characteristicLogText: UITextView!
...
```

```
/**
Value written to Characteristic
```

- Parameters:
- stringValue: the value read from the Charactersitic
- characteristic: the Characteristic that was written to

```
*/
func blePeripheral(
valueWritten value: Data,
```

toCharacteristic: CBCharacteristic)

```
{  
//let stringValue = String(data: value, encoding: .utf8) let hexValue =  
value.hexEncodedString()  
characteristicLogText.text = characteristicLogText.text + \  
  
"\n" + hexValue  
if !characteristicLogText.text.isEmpty {  
characteristicLogText.scrollRangeToVisible(NSMakeRange(0, 1)) }  
  
} ...
```

Compile and run. The updated app will host a simple GATT Profile featuring a writeonly Characteristic.

## Example code

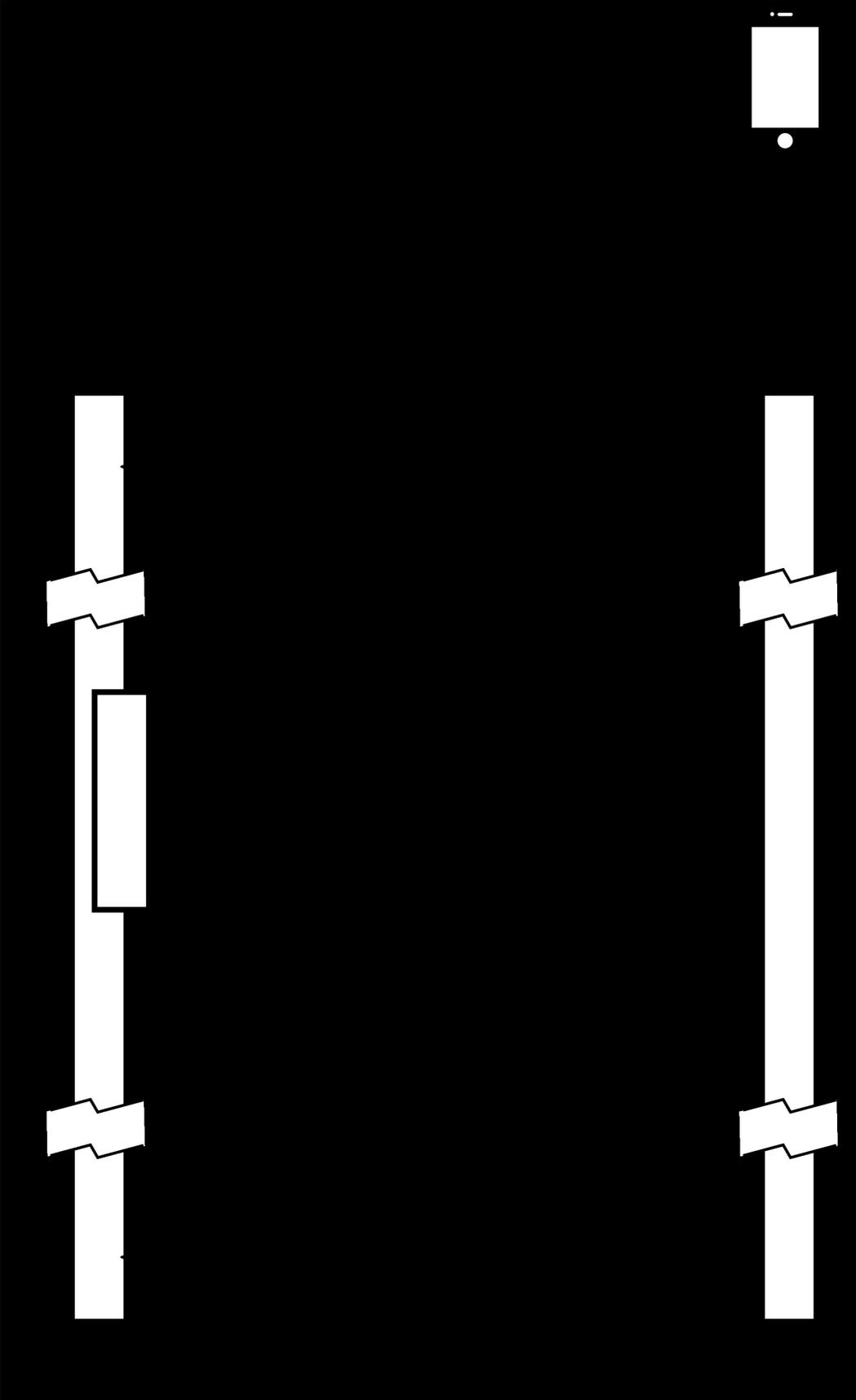
The code for this chapter is available online  
at: <https://github.com/BluetoothLowEnergyIniOSSwift/Chapter08>

## Using Notifications

Being able to read from the Central has limited value if the Central does not know when new data is available.

Notifications solve this problem. A Characteristic can issue a notification when it's value has changed. A Central that subscribes to these notifications will know when the Characteristic's value has changed, but not what that new value is. The Central can then read the latest data from the Characteristic.

The whole process looks something like this (Figure 9-1):



**Figure 9-1. The process of a Peripheral notifying a connected Central of changes to a Characteristic**

## Programming the Central

Before the Central can subscribe to the Characteristic's notifications, it is useful to know if the Characteristic supports notifications. To determine if notifications are enabled, get the properties of the Characteristic and isolate the notify property.

```
let isNotifiable = (characteristic.properties.rawValue & \
CBCharacteristicProperties.notify.rawValue) != 0
```

The Central can subscribe to or unsubscribe from a Characteristic's notifications by passing a boolean to the the `CBPeripheral.setNotifyValue()` method:

```
// subscribe to a Characteristic
```

```
peripheral.setNotifyValue(true, for: characteristic)
```

```
// unsubscribe from a Characteristic
```

```
peripheral.setNotifyValue(false, for: characteristic)
```

When the Characteristic has been subscribed to or unsubscribed from, the peripheral `didUpdateNotificationStateFor` callback is triggered in the `CBPeripheralDelegate`.

```
func peripheral(
    _ peripheral: CBPeripheral,
    didUpdateNotificationStateFor characteristic: CBCharacteristic, error: Error?)
{
}
```

Once subscribed, any changes to a Characteristic automatically trigger the peripheral `didUpdateValueFor` method in the `CBPeripheralDelegate`.

```
func peripheral(
    _ peripheral: CBPeripheral,
    didUpdateValueFor characteristic: CBCharacteristic, error: Error?)
{
}
```

## Putting It All Together

These features will be added to `BlePeripheral` to subscribe to notifications, and

we will alter TalkActivity to display a checkbox to subscribe and unsubscribe from the notifications and respond to notification alerts.

## Models

Add methods to BlePeripheral to test if a Characteristic is notifiable, to subscribe and unsubscribe from a Characteristic, and to handle the resulting callback from CBPeripheralManagerDelegate:

### Example 9-1. Models/BlePeripheral.swift

```
... /**
Subscribe to the connected characteristic.
When change is successful, delegate.subscriptionStateChanged() called */
func subscribeTo(characteristic: CBCharacteristic) {
self.peripheral.setNotifyValue(true, for: characteristic) }
/**
Unsubscribe from the connected characteristic.
When change is successful, delegate.subscriptionStateChanged() called */
func unsubscribeFrom(characteristic: CBCharacteristic) {
self.peripheral.setNotifyValue(false, for: characteristic)

}
...
/**
Check if Characteristic is notifiable

- Parameters:
- characteristic: The Characteristic to test
- returns: True if characteristic is notifiable

*/
static func isCharacteristic(
isNotifiable characteristic: CBCharacteristic) -> Bool { if
(characteristic.properties.rawValue & \

CBCharacteristicProperties.notify.rawValue) != 0 {
return true }
return false }

// MARK: CBPeripheralDelegate
```

```

/**
Characteristic has been subscribed to or unsubscribed from */

func peripheral(
    _ peripheral: CBPeripheral,
    didUpdateNotificationStateFor characteristic: CBCharacteristic, error: Error?)

{
    print("Notification state updated for: " +

"\(characteristic.uuid.uuidString)")
    print("New state: \(characteristic.isNotifying)") delegate?.blePeripheral?(

subscriptionStateChanged: characteristic.isNotifying, characteristic:
characteristic,
blePeripheral: self)

    if let errorValue = error {
        print("error subscribing to notification: ") print(errorValue.localizedDescription)
    }
}
...

```

## Delegates

Add a method to the BlePeripheralDelegate to notify subscribers that a Characteristic has been subscribed to or unsubscribed from:

### Example 9-2. Delegates/BlePeripheralDelegate.swift

```
... /**
```

A subscription state has changed on a Characteristic

- Parameters:
- subscribed: true if subscribed, false if unsubscribed
- characteristic: the Characteristic that was subscribed/unsubscribed
- blePeripheral: the BlePeripheral

```
*/
```

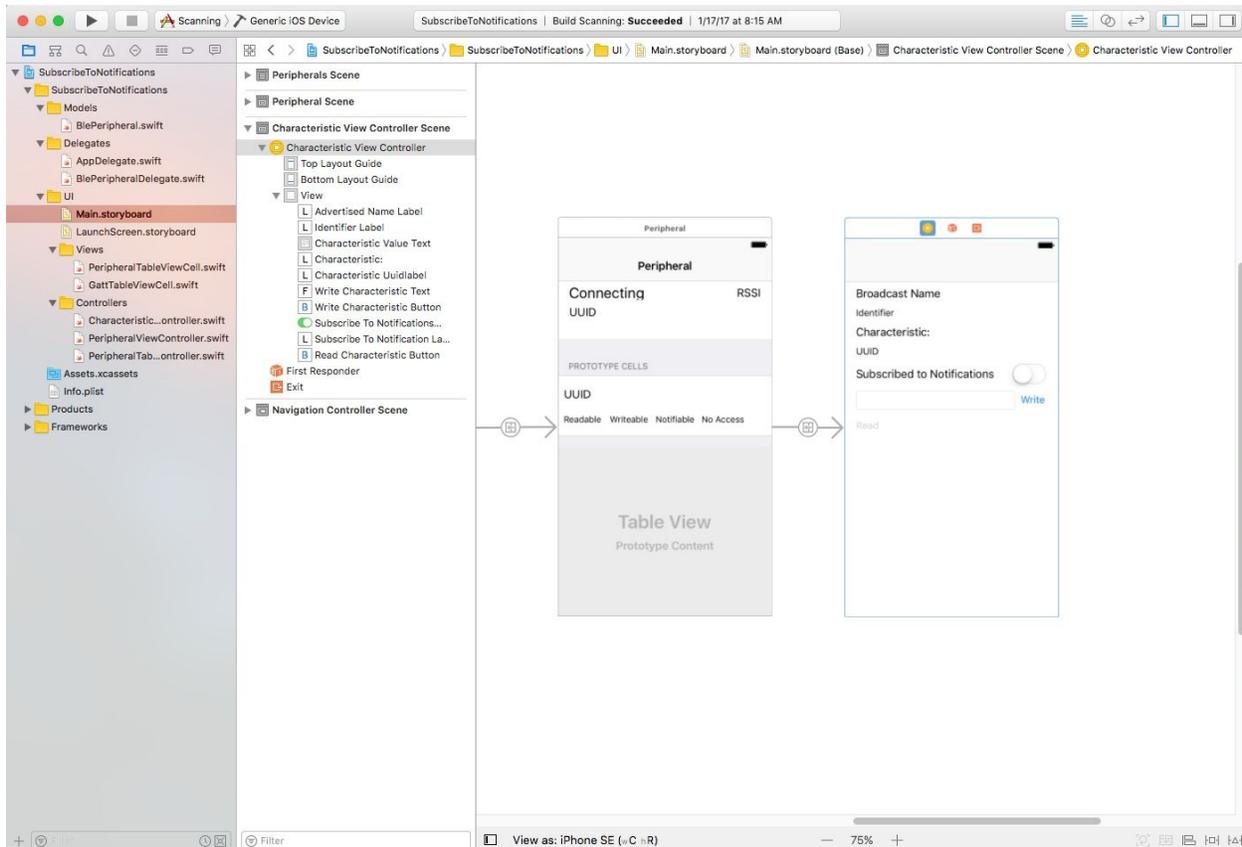
```
@objc optional func blePeripheral(
subscriptionStateChanged subscribed: Bool,
characteristic: CBCharacteristic,
```

blePeripheral: BlePeripheral)

...

## **Storyboard**

Create and link a UILabel in the GattTableViewCell to show the notification property and a UILabel and UISwitch to show the Characteristic subscription state (and a UILabel and UISwitch to show the Characteristic subscription state (2):



**Figure 9-2. Project Storyboard Views**

Modify the `GattTableViewCell` to hold a `UILabel` that shows the notification permissions for a `Characteristic`, as well as functionality to show or hide the `UILabel` to reflect the `Characteristic` permissions:

### Example 9-3. `UI/Views/GattTableViewCell.swift`

```
... @IBOutlet weak var notifiableLabel: UILabel!
```

```
...
```

```
func renderCharacteristic(characteristic: CBCharacteristic) { uuidLabel.text =
characteristic.uuid.uuidString
print(characteristic.uuid.uuidString)
var isReadable = false
var isWritable = false
var isNotifiable = false
if (characteristic.properties.rawValue & \
CBCharacteristicProperties.read.rawValue) != 0 {
print("readable")
```

```

isReadable = true
}
if (characteristic.properties.rawValue & \
CBCharacteristicProperties.write.rawValue) != 0 ||
(characteristic.properties.rawValue & \
CBCharacteristicProperties.writeWithoutResponse.rawValue) != 0 {
print("writable")
isWritable = true
}
if (characteristic.properties.rawValue & \
CBCharacteristicProperties.notify.rawValue) != 0 { print("notifiable")
isNotifiable = true
}
readableLabel.isHidden = !isReadable
writableLabel.isHidden = !isWritable
notifiableLabel.isHidden = !isNotifiable
if isReadable || isWritable || isNotifiable {
noAccessLabel.isHidden = true
} else {
noAccessLabel.isHidden = false
}
}
}
...

```

## Controllers

Modify the CharacteristicViewController to show a Subscribe to Notifications UISwitch and UILabel, plus interactions for the switch. Implement the BlePeripheralDelegate method subscriptionStateChanged to handle changes in the notification state of the Characteristic:

### Example 9-4. UI/Controllers/CharacteristicViewController.swift

```

... @IBOutlet weak var subscribeToNotificationLabel: UILabel! @IBOutlet
weak var subscribeToNotificationsSwitch: UISwitch!

...
func loadUI() {
advertisedNameLabel.text = blePeripheral.advertisedName identifierLabel.text =

```

```
\
```

```
blePeripheral.peripheral.identifier.uuidString
```

```
characteristicUuidlabel.text = \  
connectedCharacteristic.uuid.uuidString  
readCharacteristicButton.isEnabled = true
```

```
// characteristic is not readable  
if !BlePeripheral.isCharacteristic(  
isReadable: connectedCharacteristic) {  
readCharacteristicButton.isHidden = true  
characteristicValueText.isHidden = true }
```

```
// characteristic is not writeable  
if !BlePeripheral.isCharacteristic( isWritable: connectedCharacteristic) {  
writeCharacteristicText.isHidden = true writeCharacteristicButton.isHidden =  
true }
```

```
// characteristic is not writeable  
if !BlePeripheral.isCharacteristic(  
isNotifiable: connectedCharacteristic) {  
subscribeToNotificationsSwitch.isHidden = true  
subscribeToNotificationLabel.isHidden = true }  
}
```

```
...  
/**
```

User toggled the notification switch.

Request to subscribe or unsubscribe from the Characteristic \*/

```
@IBAction func onSubscriptionToNotificationSwitchChanged( _ sender:  
UISwitch)
```

```
{  
print("Notification Switch toggled")  
subscribeToNotificationsSwitch.isEnabled = false if sender.isOn {  
blePeripheral.subscribeTo(  
characteristic: connectedCharacteristic) } else {  
blePeripheral.unsubscribeFrom(  
characteristic: connectedCharacteristic) }
```

```

} ...
/**
Characteristic subscription status changed. Update UI */
func blePeripheral(
subscriptionStateChanged subscribed: Bool, characteristic: CBCharacteristic,
blePeripheral: BlePeripheral) {
if characteristic.isNotifying {
subscribeToNotificationsSwitch.isOn = true
} else {
subscribeToNotificationsSwitch.isOn = false
}
subscribeToNotificationsSwitch.isEnabled = true }
...
}

```

Compile and run. The app can scan for and connect to a Peripheral. Once connected, it can display the GATT profile of the Peripheral. It can connect to Characteristics, Subscribe to notifications, and receive data without polling.

When you hit the “Subscribe to Notifications” button, the text field should begin populating with random text generated on the Peripheral (Figure 9-3).

●●●○○ Fido 3G

20:12

⌘ 100%  ⚡

[← Peripherals](#) **Peripheral**

MyDevice

-65

501AFEEE-AA49-4AF1-8924-E56689A4I

180A

2A56

Readable

Notifiable

●●●○○ Fido 3G

20:13

⌘ 99%  ⚡

[← Peripheral](#)

MyDevice

501AFEEE-AA49-4AF1-8924-E56689A4F0A8

Characteristic:

2A56

Subscribed to Notifications



[Read](#)

SV0H0AT  
JLG3HBV  
H4XLYUB

**Figure 9-3. Apps screen showing GATT Profile of connected Peripheral with Notifications available in a Characteristic, and text read from a Characteristic**

# Programming the Peripheral

Often, battery life is at a premium on Bluetooth Peripherals. For this reason, it is useful to notify a connected Central when a Characteristic's value has changed, but not send the new data to the Central. Waking up the Bluetooth radio to send one byte consumes less battery than sending 20 or more bytes.

## Creating the Characteristic

A Characteristic that supports notifications must have a `CBCharacteristicProperties.notify` property:

```
var characteristicProperties = CBCharacteristicProperties.notify
```

It is possible to append other properties to additionally make the Characteristic readable or writable:

```
// notification support
```

```
var characteristicProperties = CBCharacteristicProperties.notify
```

```
// add read support
```

```
characteristicProperties.formUnion(CBCharacteristicProperties.read)
```

```
// add write support
```

```
characteristicProperties.formUnion(CBCharacteristicProperties.write)
```

Additionally, the Characteristic's Attributes must be readable and writeable:

```
// Characteristic Attributes are readable
```

```
var characterisiticPermissions = CBAttributePermissions.readable
```

```
characterisiticPermissions.formUnion(CBAttributePermissions.writeable)
```

Just as before, the Characteristic is created by instantiating a `CBMutableCharacteristic` object:

```
// Set Characteristic UUID
```

```
let readWriteNotifyCharacteristicUuid = CBUUID( string: "00002a56-0000-1000-8000-00805f9b34fb")
```

```
// Create a writeable Characteristic
```

```
let readWriteNotifyCharacteristic = CMutableCharacteristic( type:
```

```
readWriteNotifyCharacteristicUuid, properties: characteristicProperties, value:  
nil,
```

```
permissions: characterisiticPermissions)
```

## Responding to Callbacks

When a Central subscribes to a Characteristic, the peripheralManager didSubscribeTo method is triggered by the CBPeripheralManagerDelegate object. One of the parameters is a CBCentral, the Central that is subscribed to the Characteristic. It is important to save this reference to the Central as its a requirement in sending a notification.

```
func peripheralManager(  
  _ peripheral: CBPeripheralManager,  
  central: CBCentral,  
  didSubscribeTo characteristic: CBCharacteristic)  
  
{  
  // save a copy of the Central self.central = central
```

```
}
```

When a Central unsubscribes from a Characteristic, peripheralManager didUnsubscribeFrom method is triggered by the CBPeripheralManagerDelegate object:

```
func peripheralManager(  
  _ peripheral: CBPeripheralManager,  
  central: CBCentral,  
  didUnsubscribeFrom characteristic: CBCharacteristic)  
  
{  
  // remove reference to the Central self.central = null
```

```
}
```

Just prior to the Characteristic sending out a notification, the peripheralManagerIsReady method is triggered:

```
func peripheralManagerIsReady(  
  toUpdateSubscribers peripheral: CBPeripheralManager)  
{  
}
```

## Sending Notifications

Notifications are automatically sent when the `CBPeripheralManager` calls the `updateValue` method:

```
peripheralManager.updateValue(  
value,  
for: readWriteNotifyCharacteristic, onSubscribedCentrals: [central])
```

## Putting It All Together

These features will be added to `BlePeripheral` to allow notification subscriptions from Characteristics.

### Models

Add a reference to the connected Central, build a GATT Profile with a notification support, and callback handlers to process Characteristic subscription and unsubscription by a Central:

#### Example 9-5. Models/`BlePeripheral.swift`

```
... // MARK: GATT Profile  
  
// Service UUID  
let serviceUuid = CBUUID(string: "0000180c-0000-1000-8000-00805f9b34fb")  
// Characteristic UUIDs  
let readWriteNotifyCharacteristicUuid = CBUUID(  
  
string: "00002a56-0000-1000-8000-00805f9b34fb")  
// Read Characteristic  
var readWriteNotifyCharacteristic:CBMutableCharacteristic!  
  
// Connected Central var central:CBCentral!  
  
...  
/**  
Build Gatt Profile.  
This must be done after Bluetooth Radio has turned on  
*/  
  
func buildGattProfile() {  
let service = CBMutableService(type: serviceUuid, primary: true) var  
characteristicProperties = CBCharacteristicProperties.read
```

```

characteristicProperties.formUnion(

CBCharacteristicProperties.notify)
var characterisicPermissions = CBAAttributePermissions.writeable
characterisicPermissions.formUnion(CBAAttributePermissions.readable)

readWriteNotifyCharacteristic = CBMutableCharacteristic( type:
readWriteNotifyCharacteristicUuid, properties: characteristicProperties,
value: nil,
permissions: characterisicPermissions)

service.characteristics = [ readWriteNotifyCharacteristic ]
peripheralManager.add(service)
randomTextTimer = Timer.scheduledTimer(

timeInterval: 5,
target: self,
selector: #selector(setRandomCharacteristicValue), userInfo: nil,
repeats: true)

}
/**
Generate a random String
- Parameters
- length: the length of the resulting string

- returns: random alphanumeric string
*/
func randomString(length: Int) -> String {

let letters : NSString = \
"abcdefghijklmnopqrstuvwxyZABCDEFGHIJKLMNopQRSTUVWXYZ012345
let len = UInt32(letters.length)
var randomString = ""
for _ in 0 ..< length {
let rand = arc4random_uniform(len)
var nextChar = letters.character(at: Int(rand))
randomString += \
NSString(characters: &nextChar, length: 1) as String
}

```

```

return randomString
}

/**
Set Read Characteristic to some random text value */

func setRandomCharacteristicValue() {
let stringValue = randomString(
length: Int(arc4random_uniform(
UInt32(readCharacteristicLength - 1))) )
let value:Data = stringValue.data(using: .utf8)!
readWriteNotifyCharacteristic.value = value if central != nil {

peripheralManager.updateValue(
value,
for: readWriteNotifyCharacteristic, onSubscribedCentrals: [central])

}
print("writing " + stringValue + " to characteristic")

}
...
/**
Connected Central subscribed to a Characteristic */
func peripheralManager(
_peripheral: CBPeripheralManager,
central: CBCentral,
didSubscribeTo characteristic: CBCharacteristic)
{
self.central = central
delegate?.blePeripheral?(
subscriptionStateChangedForCharacteristic: characteristic, subscribed: true)
}

/**
Connected Central unsubscribed from a Characteristic */

func peripheralManager(
_peripheral: CBPeripheralManager,

```

```

central: CBCentral,
didUnsubscribeFrom characteristic: CBCharacteristic)

{
self.central = central
delegate?.blePeripheral?(

subscriptionStateChangedForCharacteristic: characteristic, subscribed: false) }
/**
Peripheral is about to notify subscribers of changes to a Characteristic
*/
func peripheralManagerIsReady(
toUpdateSubscribers peripheral: CBPeripheralManager)

{
print("Peripheral about to update subscribers")
}
...

```

## Delegates

Add a method to the BlePeripheralDelegate to relay changes in a Characteristic's subscription state:

### Example 9-6. Delegates/BlePeripheralDelegate.swift

```

... /**
A subscription state has changed on a Characteristic

- Parameters:
- characteristic: the Characteristic that was subscribed/unsubscribed
- subscribed: true if subscribed, false if unsubscribed

*/

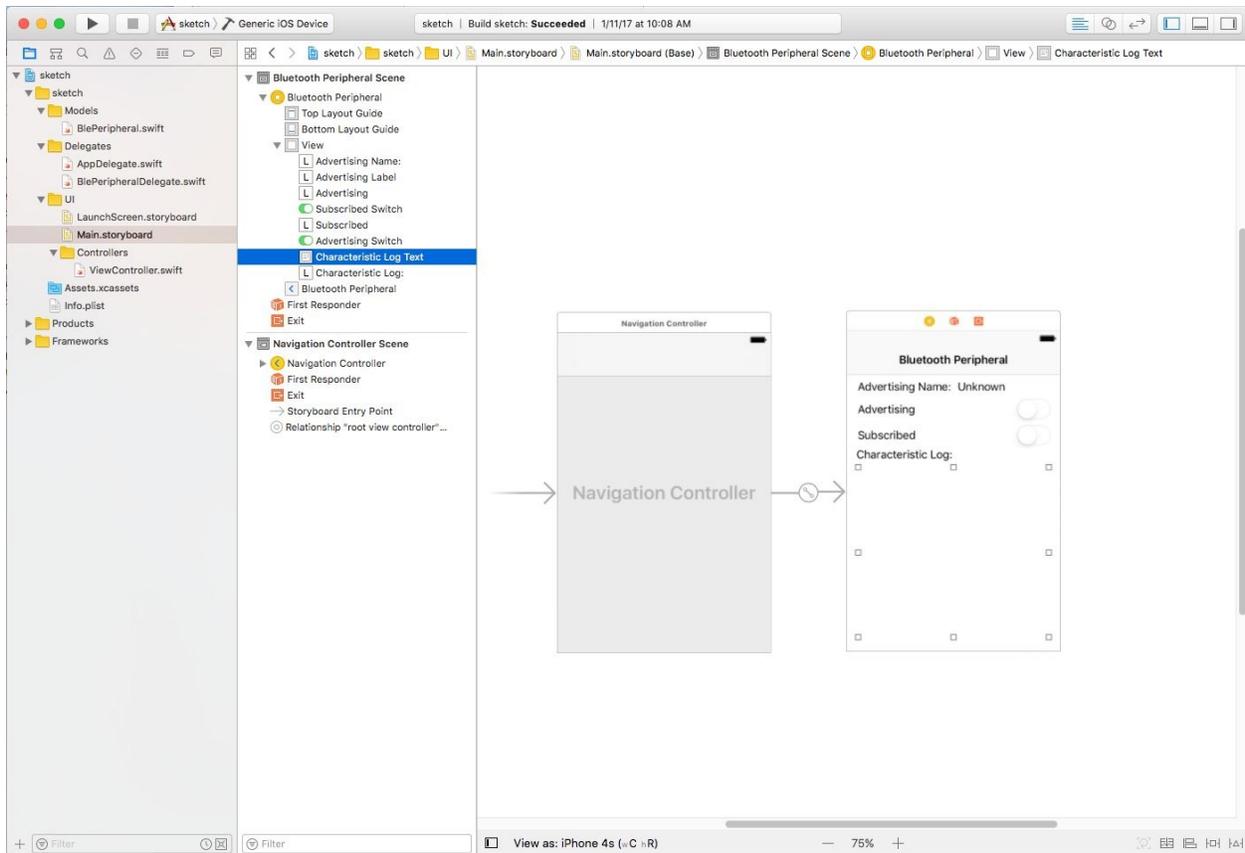
@objc optional func blePeripheral(
subscriptionStateChangedForCharacteristic \
characteristic: CBCharacteristic,

subscribed: Bool)
...

```

## Storyboard

Add a UILabel and UISwitch to show the Notification state (Figure 9-4).



**Figure 9-4. Project Storyboard**

Add a UISwitch to show the subscribed state of the Characteristic, and a callback handler to process the changes to the Characteristic subscription state.

**Example 9-7. UI/Controllers/ViewController.swift**

```

...
@IBOutlet weak var subscribedSwitch: UISwitch!
...
/**
View disappeared. Stop advertising
*/
override func viewWillDisappear(_ animated: Bool) {
subscribedSwitch.setOn(false, animated: true) advertisingSwitch.setOn(false,
animated: true)
}

```

...
/\*\*
A subscription state has changed on a Characteristic

- Parameters:

- characteristic: the Characteristic that was subscribed/unsubscribed
- subscribed: true if subscribed, false if unsubscribed

```
*/  
func blePeripheral(  
subscriptionStateChangedForCharacteristic: CBCharacteristic, subscribed: Bool)  
  
{  
subscribedSwitch.setOn(subscribed, animated: true)  
}  
...
```

Compile and run. The app can handle subscriptions to a Characteristic and send notifications when the Characteristic's value has changed (Figure 9-5).

## Bluetooth Peripheral

Advertising Name: MyDevice

Advertising



Subscribed



Characteristic Log:

FMZAp

GG0RPFNU7KYJCNJKp

G665p

**Figure 9-5. App screen Advertising Peripheral with updates a  
Characteristic**

## Example code

The code for this chapter is available online  
at: <https://github.com/BluetoothLowEnergyIniOSSwift/Chapter09>

## Streaming Data

The maximum packet size you can send over Bluetooth Low Energy is 20 bytes. More data can be sent by dividing a message into packets of 20 bytes or smaller, and sending them one at a time

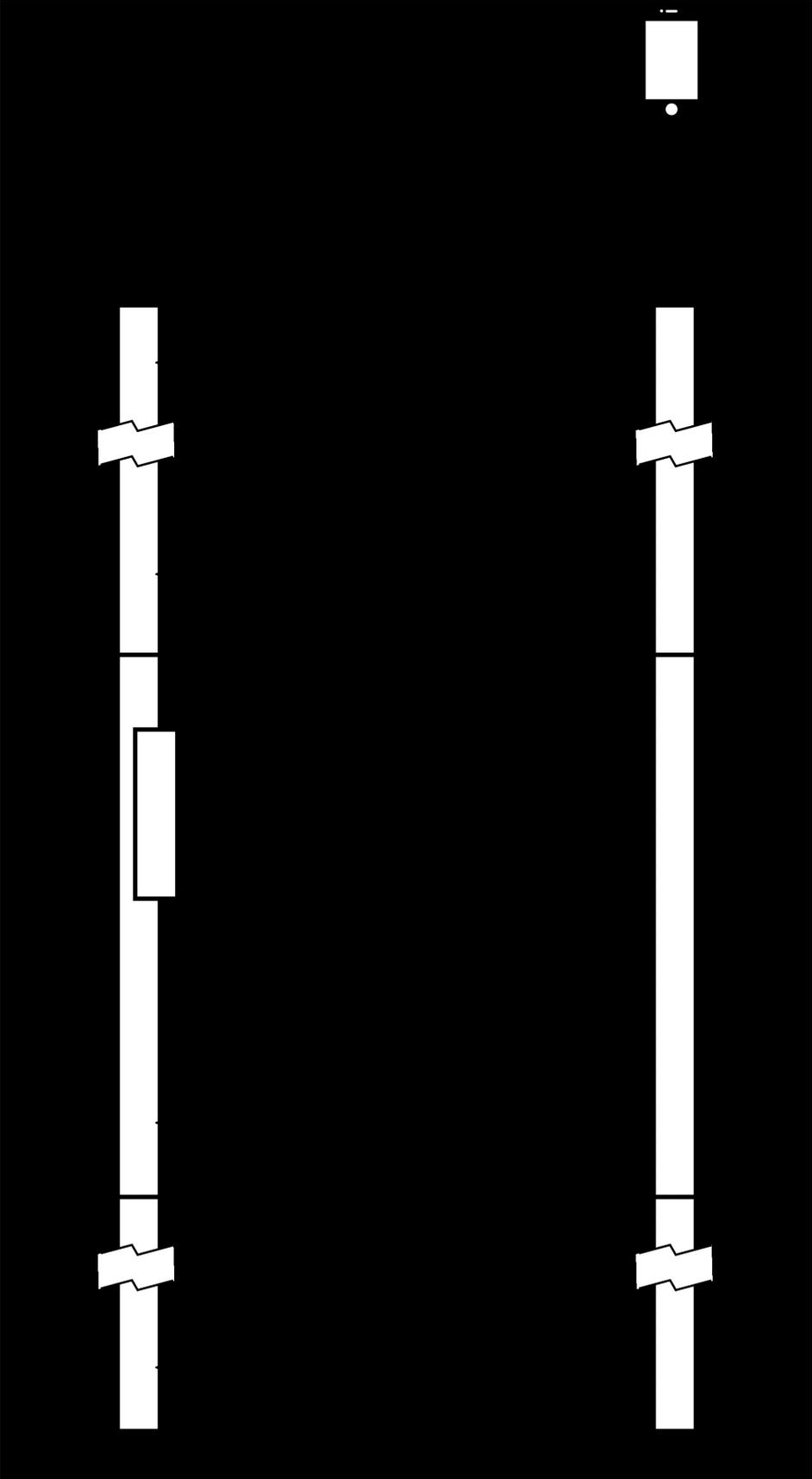
These packets can be sent at a certain speed.

Bluetooth Low Energy transmits at 1 Mb/s. Between the data transmission time and the time it may take for a Peripheral to process incoming data, there is a time delay between when one packet is sent and when the next one is ready to be sent.

To send several packets of data, a queue/notification system must be employed, which alerts the Central when the Peripheral is ready to receive the next packet.

There are many ways to do this. One way is to set up a Characteristic with read, write, and notify permissions, and to flag the Characteristic as “ready” after a write has been processed by the Peripheral. This sends a notification to the Central, which sends the next packet. That way, only one Characteristic is required for a single data transmission.

This process can be visualized like this (Figure 10-1).



**Figure 10-1. The process of using notifications to handle flow control on a multi-packed data transfer**

## Programming the Central

The Central in this example waits for the Peripheral to change the Characteristic value to the “ready” flow control message and to send a notification indicating the change. This process signals to the Central that the Characteristic is ready to receive the next packet of data.

Set up the parameters of the flow control and packet queue like this:

```
// Flow control response
let flowControlMessage = "ready"
// outbound value to be sent to the Characteristic var outboundByteArray:
[UInt8]!
// packet offset in multi-packet value var packetOffset = 0
```

The flow control works by requesting a Characteristic read event when a notification callback is triggered:

```
func peripheral(
  _ peripheral: CBPeripheral,
  didUpdateValueFor characteristic: CBCharacteristic, error: Error?)

{
  print("characteristic updated")
  if let value = characteristic.value {

    if let stringValue = String(data: value, encoding: .ascii) { if stringValue ==
      flowControlMessage {
        packetOffset += characteristicLength
        if packetOffset < outboundByteArray.count {

          writePartialValue(
            value: outboundByteArray,
            offset: packetOffset, to: characteristic)

        } else {
          // done writing message
        }
      }
    }
  }
}
```

```
}
```

The first packet of data is initialized and sent:

```
func writeValue(value: String, to characteristic: CBCharacteristic) { // get the
characteristic length
let writeableValue = value + "\0"
packetOffset = 0
// get the data for the current offset
outboundByteArray = Array(writeableValue.utf8)
writePartialValue(

value: outboundByteArray,
offset: packetOffset,
to: characteristic)

}
```

Subsequent packets are sent one at a time until there are no more left:

```
func writePartialValue(
value: [UInt8],
offset: Int,
to characteristic: CBCharacteristic)

{
// don't go past the total value size
var end = offset + characteristicLength
if end > outboundByteArray.count {

end = outboundByteArray.count
}
let transmissibleValue = Data(Array(outboundByteArray[offset..
```

## Putting It All Together

### Models

Add methods to the `BlePeripheral` to handle partial writes, properties to track flow control, and modify the `peripheral didUpdateValueFor` method to handle

the inbound flow control value:

### **Example 10-1. Models/BlePeripheral.swift**

```
... // MARK: Flow control
```

```
// Flow control response
```

```
let flowControlMessage = "ready"
```

```
// outbound value to be sent to the Characteristic var outboundByteArray:  
[UInt8]!
```

```
// packet offset in multi-packet value
```

```
var packetOffset = 0
```

```
...
```

```
/**
```

```
Write a text value to the BlePeripheral
```

- Parameters:

- value: the value to write to the connected Characteristic \*/

```
func writeValue(value: String, to characteristic: CBCharacteristic) { // get the  
characteristic length
```

```
let writeableValue = value + "\0"
```

```
packetOffset = 0
```

```
// get the data for the current offset
```

```
outboundByteArray = Array(writeableValue.utf8)
```

```
writePartialValue(  

```

```
value: outboundByteArray,  

```

```
offset: packetOffset,  

```

```
to: characteristic) }  

```

```
/**
```

```
Write a partial value to the BlePeripheral
```

- Parameters:

- value: the full value to write to the connected Characteristic

- offset: the packet offset

```
*/
```

```
func writePartialValue(  

```

```
value: [UInt8],  

```

```

offset: Int,
to characteristic: CBCharacteristic)

{
// don't go past the total value size
var end = offset + characteristicLength
if end > outboundByteArray.count {

end = outboundByteArray.count
}
let transmissibleValue = \

Data(Array(outboundByteArray[offset..<end])) print("writing partial value: \
(offset)-\<(end)") print(transmissibleValue)
var writeType = CBCharacteristicWriteType.withResponse if
BlePeripheral.isCharacteristic(

isWritableWithoutResponse: characteristic) { writeType =
CBCharacteristicWriteType.withoutResponse }
peripheral.writeValue(
transmissibleValue,
for: characteristic,
type: writeType)
print("write request sent")
}

...
/**
Value downloaded from Characteristic on connected Peripheral */
func peripheral(
_ peripheral: CBPeripheral,
didUpdateValueFor characteristic: CBCharacteristic, error: Error?)
{
print("characteristic updated") if let value = characteristic.value {
print(value.debugDescription) print(value.description)

// Note: if we need to work with byte arrays
// instead of Strings, we can do this
// let byteArray = [UInt8](value)
// or this:

```

```

// let byteArray:[UInt8] = Array(outboundValue.withCString) if let stringValue =
String(data: value, encoding: .ascii) {

print(stringValue)
// received response from Peripheral
delegate?.blePeripheral?(

characteristicRead: stringValue,
characteristic: characteristic,
blePeripheral: self)

if stringValue == flowControlMessage {
packetOffset += characteristicLength
if packetOffset < outboundByteArray.count {

writePartialValue(
value: outboundByteArray,
offset: packetOffset, to: characteristic)

} else {
print("value write complete")
// done writing message
delegate?.blePeripheral?(

valueWritten: characteristic,
blePeripheral: self) }
}
}
}
}
}

```

...

The resulting app can send larger amounts of data to a connected Peripheral by queueing and transmitting packets one at a time ( Figure 10-2).

●●●○○ Fido 3G

20:12

⌘ 100%  ⚡

[← Peripherals](#) **Peripheral**

MyDevice

-65

501AFEEE-AA49-4AF1-8924-E56689A4I

180A

2A56

Readable Writeable Notifiable

●●●○ Fido 3G

20:52

100%  ⚡

[← Peripheral](#)

MyDevice

501AFEEE-AA49-4AF1-8924-E56689A4F...

Characteristic: 2A56

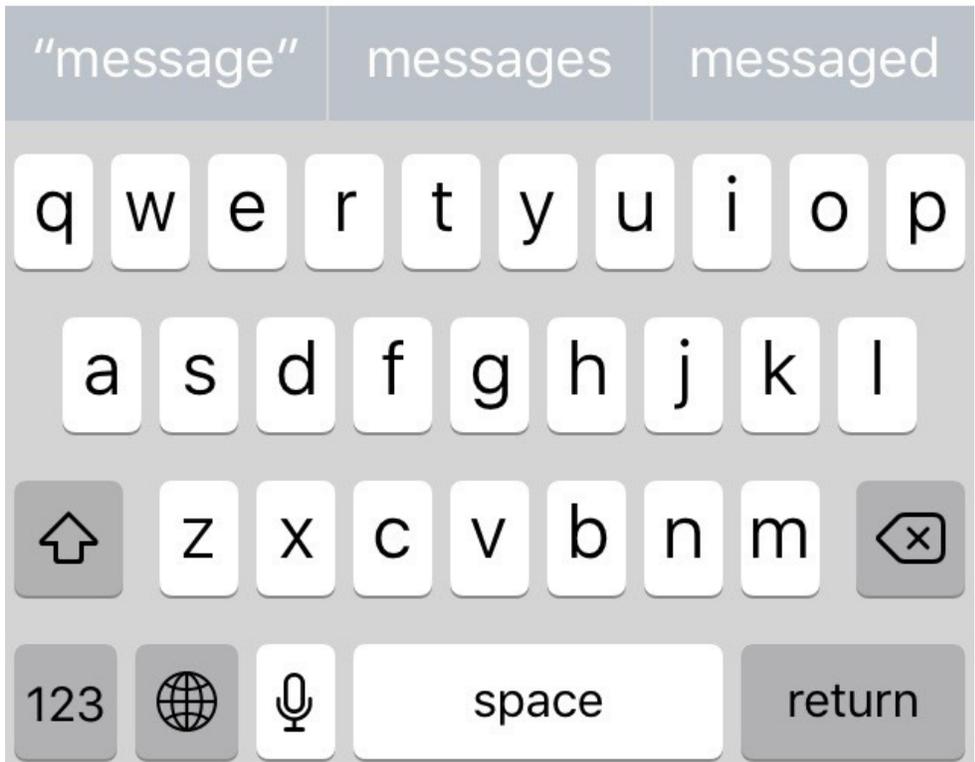
Subscribe to Characteristic



this is a super long message

[Write](#)

[Read](#)



**Figure 10-2. App screens showing GATT Profile for the Advertising Peripheral and multipart value queued to be sent to a Characteristic.**

## Programming the Peripheral

The Peripheral in this example processes a value written to a Characteristic, then sets the Characteristic's value to the "ready" flow control message, and sends a notification to the Central. In this way, the Central is triggered to read the Characteristic. When the Central reads the flow control message, it knows when to send the next packet of data.

When the Peripheral's Characteristic is written to, the `peripheralManager` `didReceiveWrite` method is triggered in the `CBPeripheralManagerDelegate` object. Once the incoming data is processed, the flow control value can be written to the Characteristic locally and the notification sent to the Central notifying it of the change.

The implementation looks like this:

Set up the parameters of the flow control:

```
let flowControlString = "ready"
```

The `flowControlString` is written to the Characteristic and a notification is sent after the Central initiates a write request:

```
func peripheralManager(
    _ peripheral: CBPeripheralManager, didReceiveWrite requests:
    [CBATTRequest])

{
    for request in requests {
        // Do something with the incoming request.value
        // notify the Central of a successful write
        peripheral.respond(to: request, withResult: CBALError.success) // convert flow
        control into a Data object
        let flowControlValue:Data = flowControlString.data(using: .utf8)! // send flow
        control value
        peripheralManager.updateValue(

        flowControlValue,
        for: readWriteNotifyCharacteristic,
        onSubscribedCentrals: [request.central])

    }
}
```

```
}
```

## Putting It All Together

Copy the previous chapter's project into a new project.

### Models

Add functionality to `BlePeripheral` to describe the flow control value, and to send the flow control message when a write request is triggered:

#### Example 10-2. Models/BlePeripheral.swift

```
... // HARK: Flow Control
let flowControlString = "ready"

...
/**
Connected Central requested to write to a Characteristic */

func peripheralManager(
    _ peripheral: CBPeripheralManager,
    didReceiveWrite requests: [CBATTRequest])

{
    for request in requests {
        peripheral.respond(to: request, withResult: CBALError.success) // convert flow
        control into a Data object
        let flowControlValue:Data = flowControlString.data(

using: .utf8)!
// send flow control value
peripheralManager.updateValue(

flowControlValue,
for: readWriteNotifyCharacteristic,
onSubscribedCentrals: [request.central])

if let value = request.value {
    delegate?.blePeripheral?(
        valueWritten: value,
        toCharacteristic: request.characteristic) }
}
```

```
}  
}
```

...

The resulting app can receive a queued stream of data from a Central by issuing a receive/response flow control on the Characteristic (Figure 10-3).

●●○○ Fido 3G

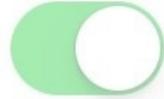
15:10

⌘ 49%  ⚡

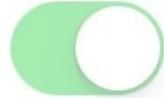
## Bluetooth Peripheral

Advertising Name: MyDevice

Advertising



Subscribed



Characteristic Log:

this is a super  
long message

**Figure 10-3. App screen showing multipart value queued to be sent to a Characteristic.**

## Example code

The code for this chapter is available online

at: <https://github.com/BluetoothLowEnergyIniOSSwift/Chapter10>

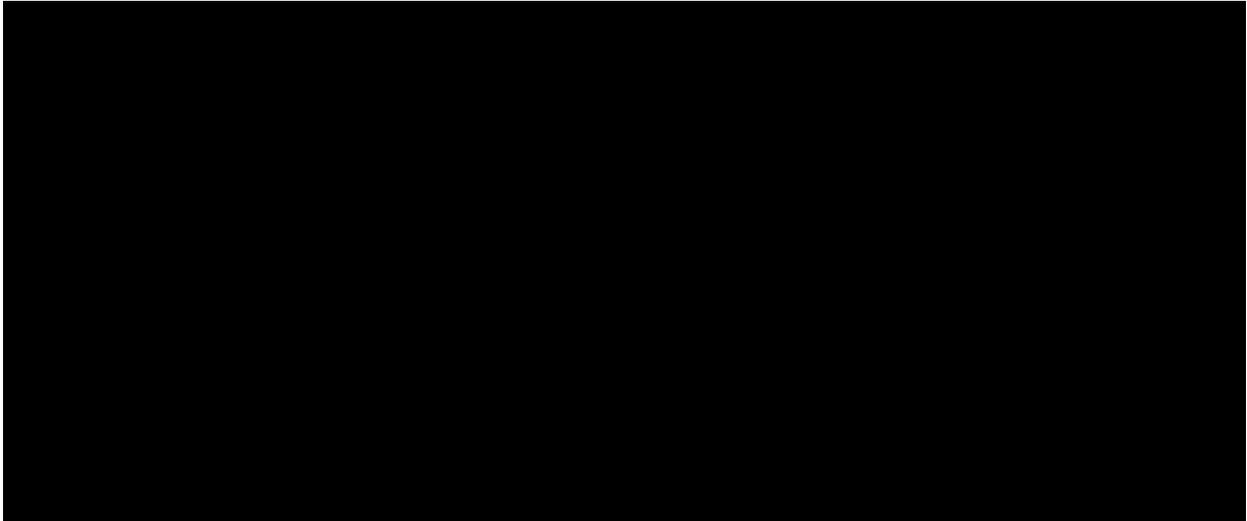
## **Project: iBeacon**

Beacons can be used for range finding or spacial awareness. iBeacons are a special type of Beacon that is widely supported by the industry. It supports certain data that identifies the iBeacons to makes range finding and spacial awareness easier across platforms.

Due to the nature of how radio signals diminish in intensity with distance, Bluetooth Peripherals can be used both for range finding and spacial awareness.

## Range Finding

Bluetooth signals can be used to approximate the distance between a Peripheral and a Central because the radio signal quality drops off in a predictable way with distance. The diagram below shows how the signal might drop as distance increases (Figure 11-1).



**Figure 11-1. Distance versus Bluetooth Signal Strength**

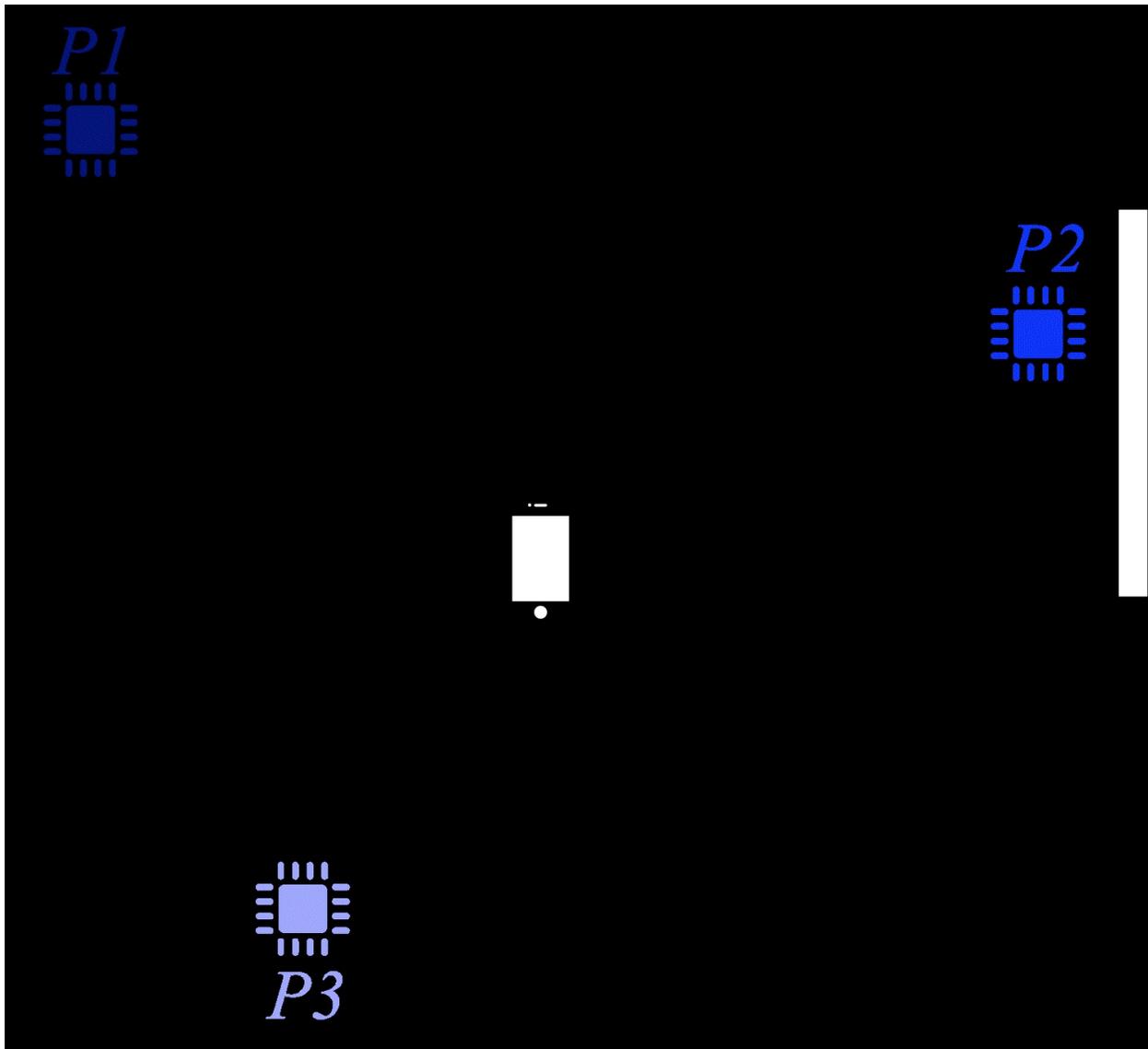
This drop-off rate, known as the Inverse-Square Law, is universal with electromagnetic radiation.

Due to radio interference and absorption from surrounding items, the radio signal propagation varies a lot from environment to environment, and even step to step. This makes it very difficult to know the precise distance between a Central and an iBeacon.

One or more Centrals can approximate their distance from a single iBeacon without connecting.

## **Spacial Awareness**

A Central can approximate its position in space using a process called trilateration. Trilateration works by a computing series of equations when both the distance from and location of nearby iBeacons are known (Figure 11-2).



**Figure 11-2. Example Central and iBeacon positions in a room**

This is a pretty math-intensive process, but it's all based on the Pythagorus Theorem. By calculating the shape of the triangles made from the relative positions of all the iBeacons and the Central, one can determine the location of the Central (iBeacons and the Central, one can determine the location of the Central ( 3).

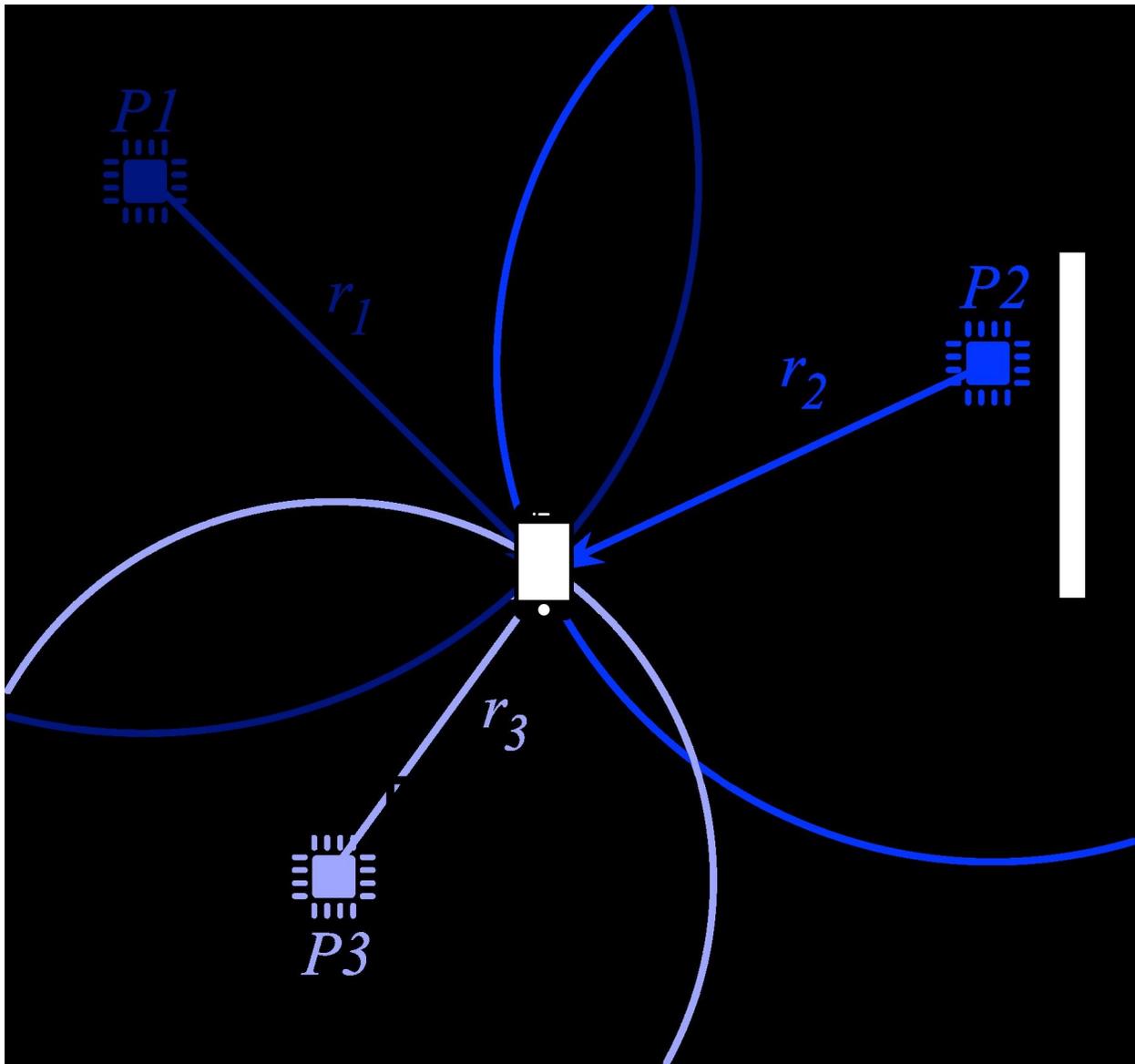


Figure 11-3. Distances from iBeacons to Central

# iBeacons

The Scan Result allows a Central to read information from a Peripheral without connecting to it, in much the same way that the advertising name is read.

Although iOS supports iBeacon scanning, it does not support iBeacon advertising. Therefore it is possible to build an app that discovers iBeacons but not possible to create an iBeacon in iOS

iBeacons are beacons that advertise information about their location and advertise intensity using the Scan Result feature of Bluetooth Low Energy. The Scan Result allows a Central to read information from a Peripheral without connecting to it, in much the same way that the Device name is advertised. There are only two tricks to creating an iBeacon client in iOS:

1. All iBeacons for the same location service have the same UUID. 2. iBeacons have Major and Minor numbers for a sort of layered identification. What the Major and Minor numbers represent are dependant on the iBeacon administrator, but typically a Major number represents iBeacons for a particular organization or building, and a Minor number represents iBeacons from a specific area or floor of a building, etc.

3. The distance to the iBeacon is calculated based on the RSSI. iBeacons do this by advertising certain data that can be referenced when looking up where the iBeacons are located.

An example implementation is a museum that has iBeacons at each exhibit in the museum. All iBeacons in the exhibit share the same UUID. The museum uses Major numbers to identify floors and Minor numbers to identify rooms of the exhibit.

The museum's smartphone app has an internal data set relating Major and Minor values to the floors and rooms of the exhibit. It scans for all iBeacons with a specific UUID. The nearby iBeacons are discovered and read. The discovered iBeacons' Major and Minor numbers are looked up to learn that the user is in a specific room on a specific floor in the museum. Relevant content is accessed from the museum's API and loaded in the smartphone app.

# Programming the Central

There are two parts to working with iBeacon on iOS:

1. Requesting Access to the CoreLocation Framework
2. Searching for iBeacons

## Requesting Access to CoreLocation

iOS provides an iBeacon API in the CoreLocation Framework. In order to work with iBeacons, the CoreLocation Framework must be imported, and the working class must implement CLLocationManagerDelegate:



**Figure 11-4. CoreLocation Frameworks linked into project**

Import the CoreLocation API in the code header to access both the Bluetooth and iBeacon APIs:

```
// import CoreLocation Framework
import CoreLocation
...
class GameViewController: UIViewController, CLLocationManagerDelegate { }
```

Before doing anything, the user must authorize the device to use the CoreLocation library. This is done by checking the current `CLAuthorizationStatus`, which states if the app is allowed to access the CoreLocation Framework:

```
// If location services are not enabled, request authorization if
(CLLocationManager.authorizationStatus() != \
CLAuthorizationStatus.authorizedWhenInUse) {
locationManager.requestWhenInUseAuthorization() } else {
// location manager already authorized. proceed }
```

If this is the first time the app is run, or if the user has changed the location settings, they will be prompted to authorize the location manager. Their choice will propagate an updated `CLLocationAuthorizationStatus`.

The new authorization status can be one of several defined by `CLLocationAuthorizationStatus`:

**Table 11-1. CLLocationAuthorizationStatus**

**Value Description**

**notDetermined**

**restricted**

**denied**

**authorizedAlways**

**authorizedWhenInUse**

The user has not yet made a choice regarding whether this app can use location services.

This app is not authorized to use location services due to active restrictions.

The user explicitly denied the use of location services for this app or location services are currently disabled in Settings.

This app is authorized to start location services at any time, including monitoring for location changes in the background.

This app is authorized to start most location services while running in the foreground, but not when in the background.

The new `CLLocationAuthorizationStatus` will come back when the `locationManagerDidChangeAuthorization` callback is triggered by the `CLLocationManagerDelegate`.

```
func locationManager(  
  _ manager: CLLocationManager,  
  didChangeAuthorization status: CLLocationAuthorizationStatus)  
  
  {  
    if status == .authorizedAlways {  
  
      // location manager is authorized all the time for this app. proceed } else if status  
      == .authorizedWhenInUse){  
      // location manager is authorized when the app is // in the foreground. proceed  
  
    } else {
```

```
// location manager is unauthorized
}  
}
```

Once the app is allowed to access the CoreLocation framework, it is possible to check if the iBeacon Ranging feature is available:

```
if CLLocationManager.isMonitoringAvailable(for: CLBeaconRegion.self) { if  
CLLocationManager.isRangingAvailable() {
```

```
// Ranging feature is available. Start scanning } else {  
// Ranging feature is unavailable  
}  
}
```

## Scanning for iBeacons

In iOS, the process of scanning for iBeacons is called Ranging. As similar iBeacons have the same UUID, a Range is created using a known UUID to narrow down the iBeacon search to those that are expected to contain useful information to the App:

```
// only look for specific iBeacon UUIDs (known as a Proximity UUID) let  
proximityUuid = UUID(uuidString: "E20A39F4-73F5-4BC4-A12F-  
17D1AD07A961") // build one or more iBeacon "region" to search for  
// using the Proximity UUID. Each has a unique identifier for reuse later let  
beaconRegion = CLBeaconRegion(proximityUUID: UUID(  
  
uuidString: proximityUuid!,  
identifier: "regionUniqueId")
```

iBeacons may contain Major or Minor numbers as well, which further identify groups of iBeacons. It is possible to create an iBeacon Region that isolates both the Proximity UUID and the Major number:

```
let beaconRegion = CLBeaconRegion( proximityUUID: proximityUuid, major:  
1122,  
identifier: "anotherRegionUniqueId")
```

Or a Region that isolates the UUID, Major number, and Minor number:

```
let beaconRegion = CLBeaconRegion( proximityUUID: proximityUuid, major:
1122,
minor: 3344,
identifier: "anotherRegionUniqueId")
```

Begin Ranging for iBeacons using the the startMonitoring and startRangingBeacons functions in the CLLocationManager class:

```
let locationManager = CLLocationManager()
locationManager.startMonitoring(for: beaconRegion)
locationManager.startRangingBeacons(in: beaconRegion)
```

When iBeacons are discovered during Ranging, the locationManager didRangeBeacons callback is triggered by the CLLocationManagerDelegate.

This method will return a CLBeacon array periodically. This array may have zero or more elements, representing all the iBeacons it found during the Ranging process. This may include iBeacons that it found during the previous Ranging scan, so watch out for duplicates.

```
func locationManager(
_ manager: CLLocationManager, didRangeBeacons beacons: [CLBeacon], in
region: CLBeaconRegion)
```

```
{
//beaconswillcontainanarrayof0ormoreCLBeacons
//EachscanmaycontainCLBeaconsthatwerecontainedinthe
//lasttimelocationManagerdidRangeBeaconswastriggered
}
```

The iBeacon major number and minor number can be retrieved from the CLBeacon object:

### **Property Data Type Value**

**.major** NSNumber iBeacon Major Number

**.minor** NSNumber iBeacon Minor Number

**.proximityUUID** UUID iBeacon Proximity UUID

For security reasons, iOS does not reveal the transmission power of an iBeacon, which can be used to determine the distance between the Central and the iBeacon.

An iBeacon's distance from the Ranging Central can be approximated using the CLBeacon.proximity property

#### Table 11-2. CLProximity

| Value | Signal Strength | Approximate Distance |
|-------|-----------------|----------------------|
|-------|-----------------|----------------------|

|                  |        |                         |
|------------------|--------|-------------------------|
| <b>immediate</b> | Strong | Up to a few centimeters |
|------------------|--------|-------------------------|

|             |        |                    |
|-------------|--------|--------------------|
| <b>near</b> | Medium | Up to a few meters |
|-------------|--------|--------------------|

|            |      |                        |
|------------|------|------------------------|
| <b>far</b> | Weak | More than a few meters |
|------------|------|------------------------|

|                |                |                         |
|----------------|----------------|-------------------------|
| <b>unknown</b> | Extremely weak | Impossible to determine |
|----------------|----------------|-------------------------|

This can be implemented in code like this:

```
func getProximityString(fromBeacon: CLBeacon) -> String { switch
beacon.proximity {
case .unknown:

return "Unknown"
case .immediate:
return "Immediate"
case .near:
return "Near"
case .far:
return "Far"
}
}
```

If the transmission power of the iBeacon is known, that can be useful for determining how far away the iBeacon is.

### Calculate Distance From iBeacons

Approximate the distance from a Central to an iBeacon using the RSSI reported at that distance, a reference RSSI at 1 meter, and an approximate propagation constant, using this equation:

$$d \approx 10^{A - R / 10n}$$

where d = distance between iBeacon peripheral and central A = RSSI when central and peripheral are 1 meter apart

R = RSSI at distance d

n = The radio propagation constant; typically between 2.7 and 4.3

This is expressed in code as follows:

```
func getDistanceFromRssi(  
  rssi: Int, referenceRssi: Int, propagationConstant: Float) -> Double  
  
  {  
    letexponent=Double(referenceRssi-rssi)/(10*propagationConstant);  
    letdistance=pow(10.0,exponent)  
    returndistance  
  
  }
```

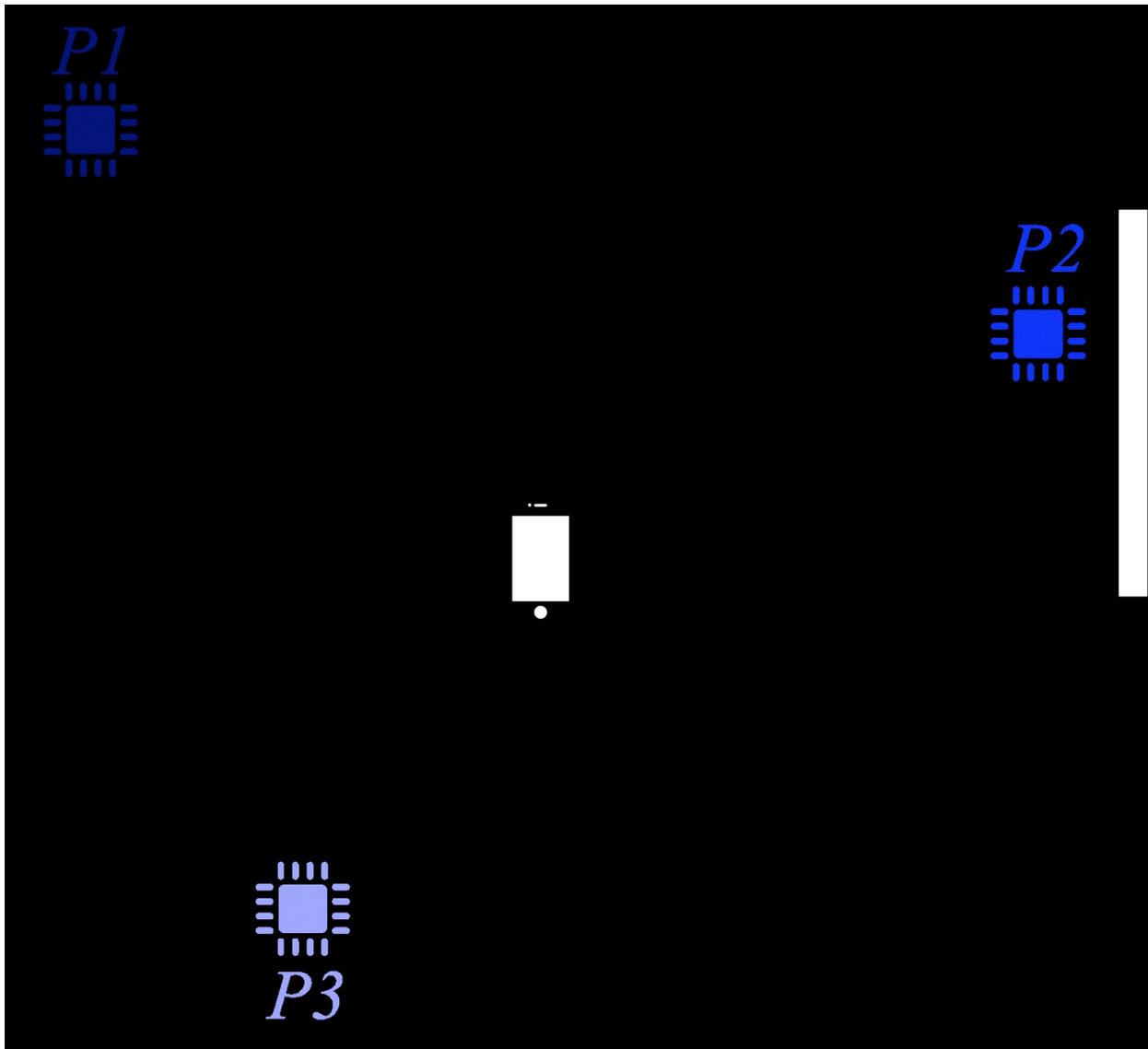
Due to radio interference and absorption from surrounding items, the radio propagation constant, n, varies a lot from environment to environment and even step to step. This makes it very difficult to know the exact distance between a Central and an iBeacon. The radio propagation constant can be approximated by testing the iBeacon's RSSI at 1 meter in conditions similar to what is expected in the field.

## **Spacial Awareness**

Following a few equations derived from Pythagorus' Theorem, it is possible to determine the location of the Central from 3 known iBeacon locations.

### **List Known Variables**

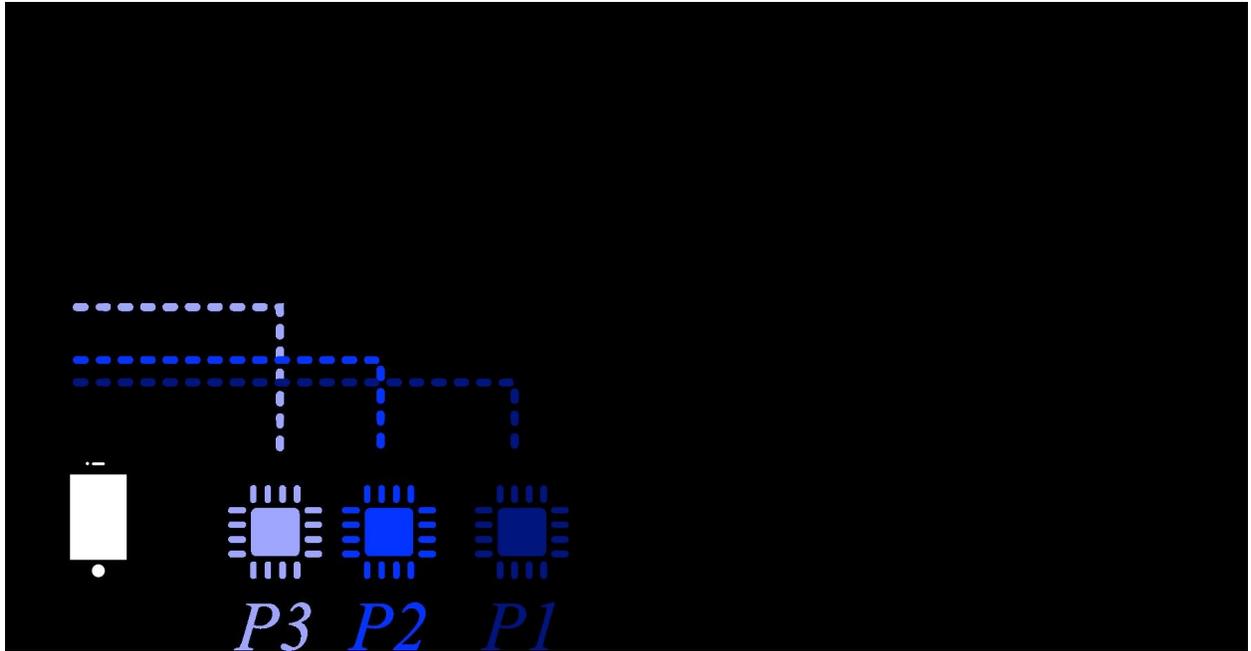
Imagine a room with known iBeacon's locations, P1, P2, and P3 as x,y coordinates (Figure 11-5).

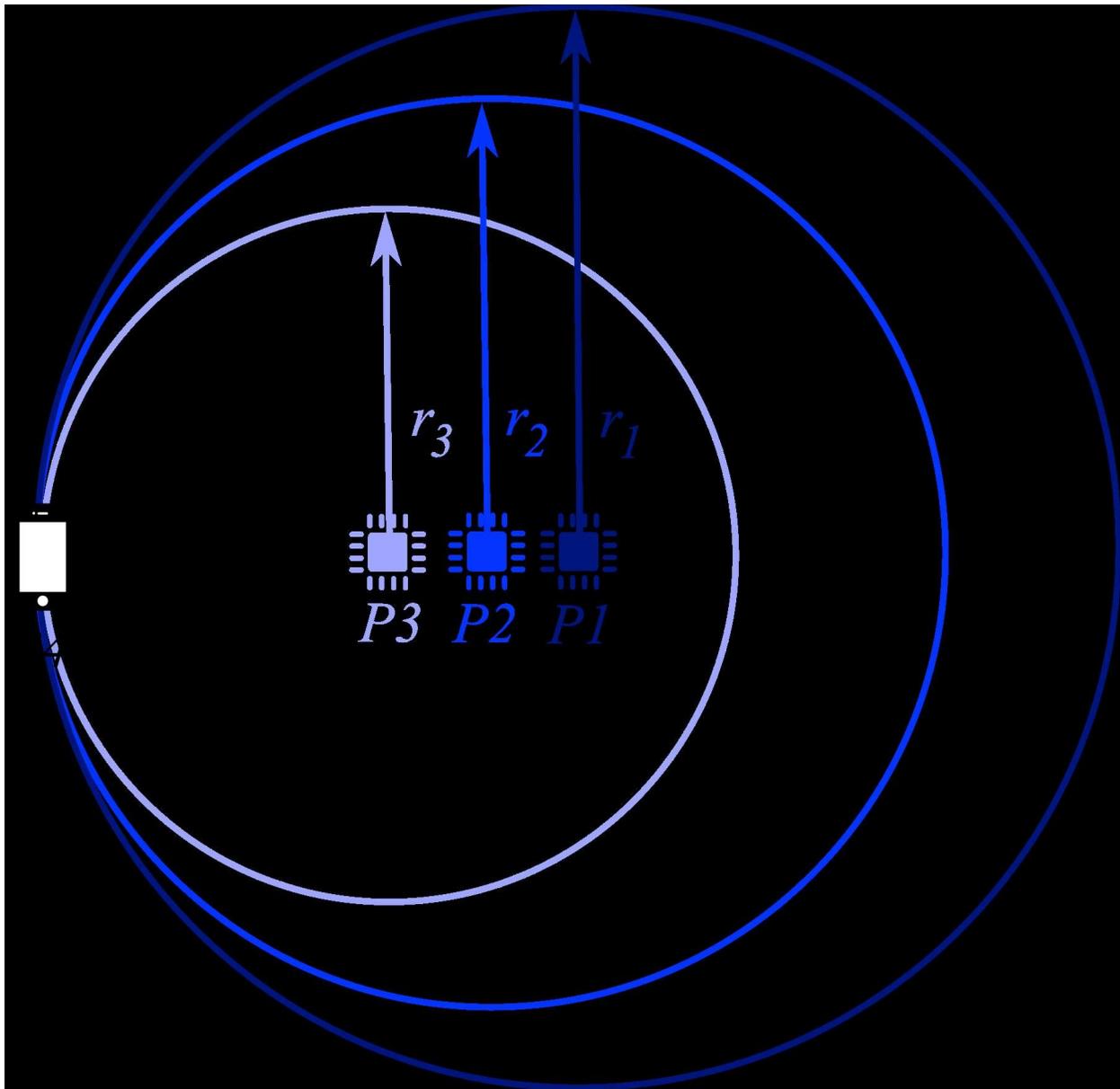


**Figure 11-5. Example iBeacon positions in a room**

```
// three iBeacons with known distances from a fixed spot let p1 = new  
CGPoint(x: 10, y: 10)  
let p2 = new CGPoint(x: 50, y: 30)  
let p3 = new CGPoint(x: 35, y: 50)
```

Using the `getDistanceFromRssi` method, the distance between each iBeacon and the Central has been derived as  $r_1$ ,  $r_2$ ,  $r_3$  (Figure 11-6).



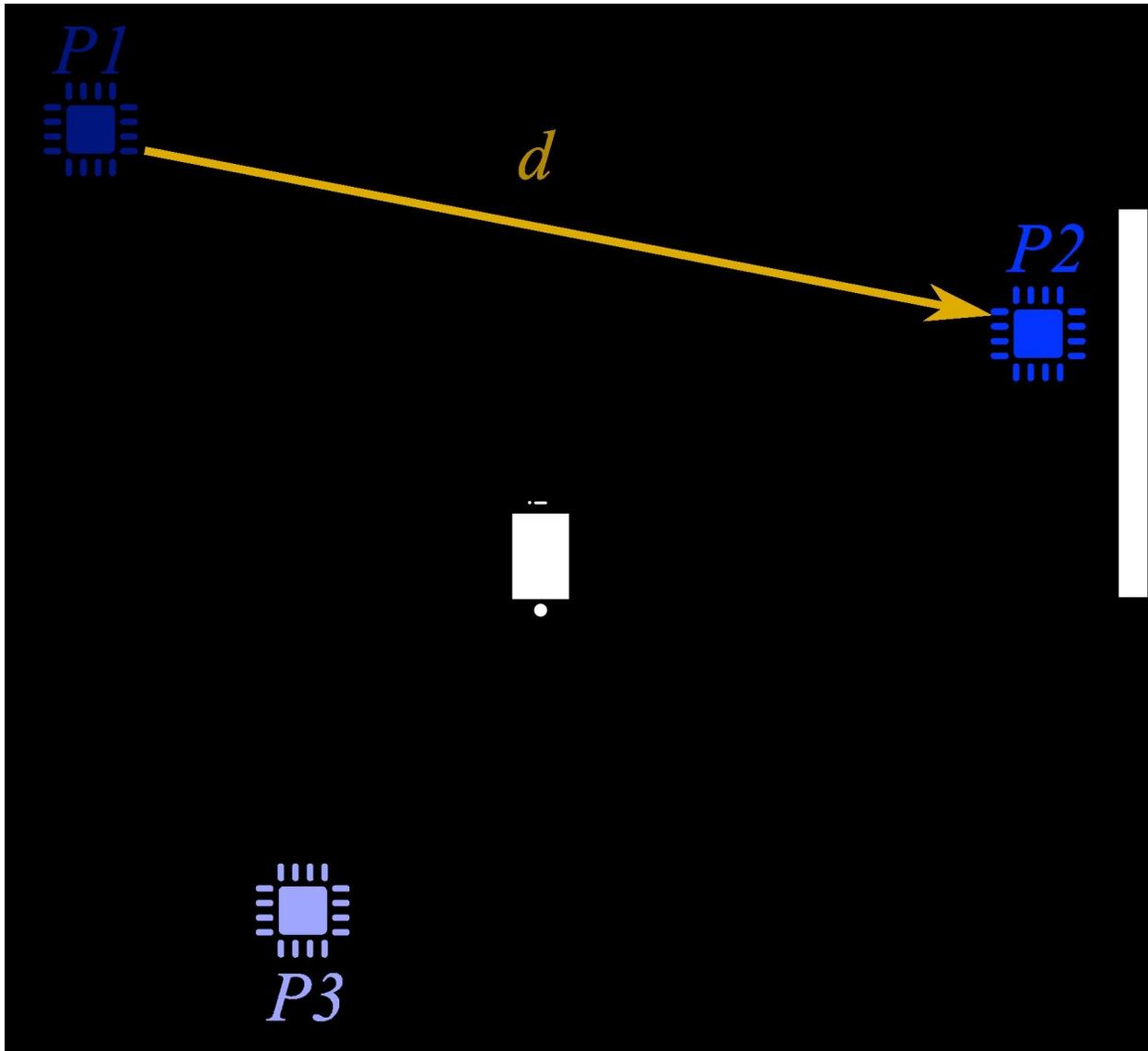


**Figure 11-6. Beacon distances from Central derived from their signal strengths**

// distance to each of the iBeacons has already been calculated let  $r_1 = 13.2$ ,  
 let  $r_2 = 10.8$ ,  
 let  $r_3 = 7.7$ ;

### Calculate Distance Between iBeacons

Calculate the distance between iBeacons  $P_1$  and  $P_2$  using Pythagorean Theorem (Figure 11-7).



**Figure 11-7. Derived distance from iBeacon 1 to iBeacon 2**

$$d = P2 - P1$$

Which is short-hand for this:

$$d = (P2_x - P1_x)^2 + (P2_y - P1_y)^2$$

This is expressed in code as follows:

```
let adjacent = p2.x - p1.x
let opposite = p2.y - p1.y
let d = sqrt( pow(adjacent, 2) + pow(opposite, 2) )
```

**Calculate the Unit Vector between P1 and P2**

A unit vector represents a direction, but not a distance. Here we calculate the unit vector between P1 and P2, which shows which direction P2 is relative to P1 (Figure 11-8).

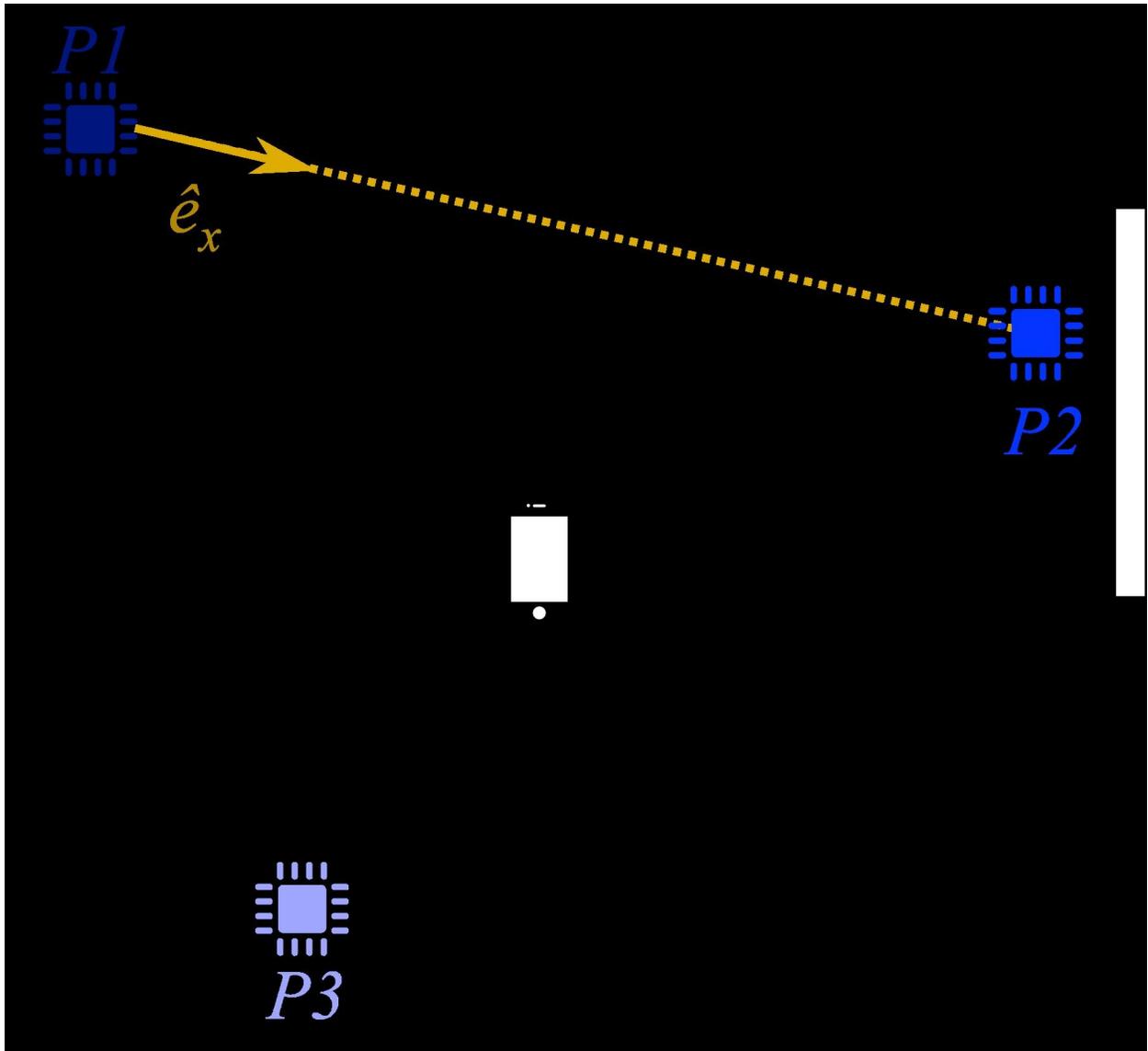


Figure 11-8. Derived direction from iBeacon 1 to iBeacon 2

$$\hat{e}_x$$

=

$$\frac{P2 - P1}{\|P2 - P1\|}$$

$\hat{e}_x$

=

$$\frac{P2 - P1}{\|P2 - P1\|}$$

(

$d$

,

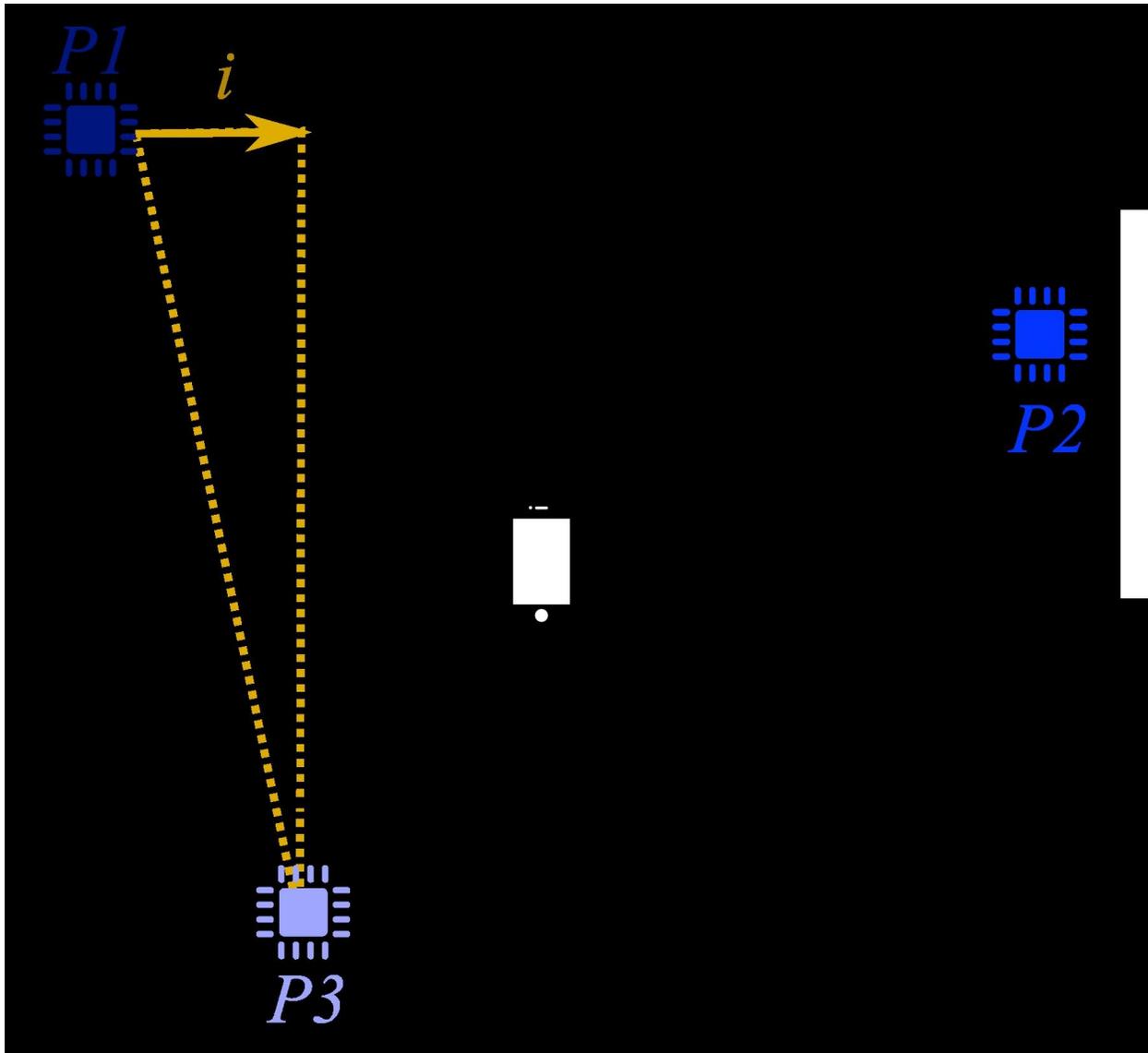
$$P2_y - P1_{yx} - P1_x$$

$d$ )

let  $exx = (p2.x - p1.x) / d$  let  $exy = (p2.y - p1.y) / d$

### **Calculate Magnitude of Distance from P1 to P3**

Find the magnitude  $i$  of the distance between P1 and P3 in the x direction (Find the magnitude  $i$  of the distance between P1 and P3 in the x direction ( 9).



**Figure 11-9. Derived horizontal component of distance from Beacon 1 to Beacon 2**

$$i = e_x \cdot (P3 - P1)$$

$$i = e_{xx}(P3_x - P1_x) + e_{xy}(P3_y - P1_y)$$

$$\text{let } i = \text{exx} * (\text{p3.x} - \text{p1.x}) + \text{exy} * (\text{p3.y} - \text{p1.y})$$

### Calculate Unit Vector between P1 and P3

Calculate the unit vector  $\hat{e}_y$  between P1 and P3, showing the direction between P1 and P3 (Figure 11-10).

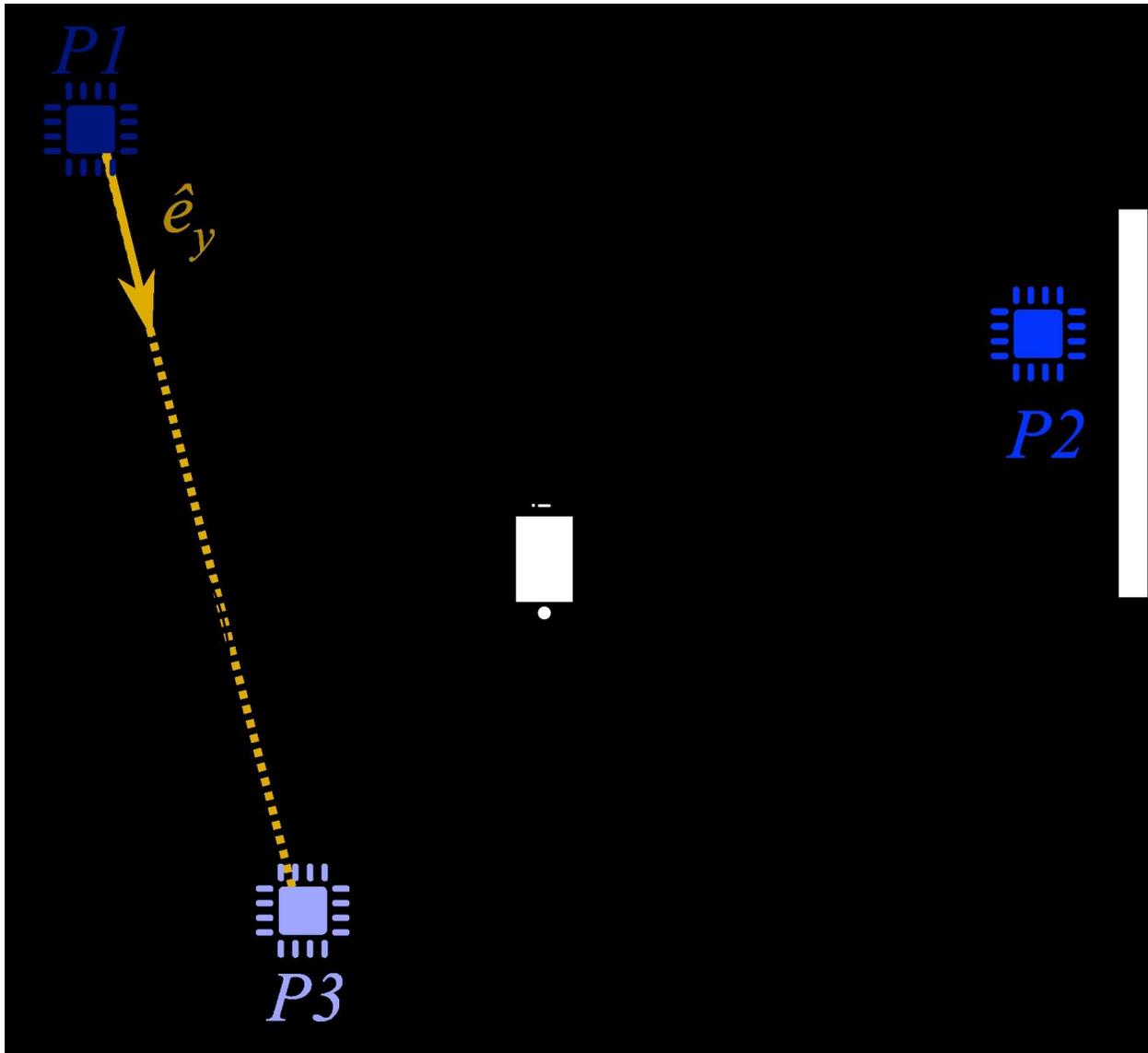


Figure 11-10. Derived direction from iBeacon 1 and iBeacon 3

$\hat{e}_y$

=

$$P_3 - P_1 - \hat{i}_x$$

$P_3 - P_1 - \hat{i}_x$

*i*

*ex*

*e*

*y*

=

$P3_x - P1_x - ie_{xx}, y - P1_y - ie_{xy}P3$

2 2 2 2

$(P3_x - P1_x - ie_{xx}) + (P3_y - P1_y - ie_{xy})(P3_x - P1_x - ie_{xx}) + (P3_y - P1_y - ie_{xy})$

let  $eyx = (p3.x - p1.x - i * exx) / \text{sqrt}(\text{pow}(p3.x - p1.x - i * exx, 2) +$

$\text{pow}(p3.y - p1.y - i * exy, 2)$

)

let  $eyy = \text{sqrt}(\text{pow}(p3.x - p1.x - i * exx, 2) +$

$\text{pow}(p3.y - p1.y - i * exy, 2) +$

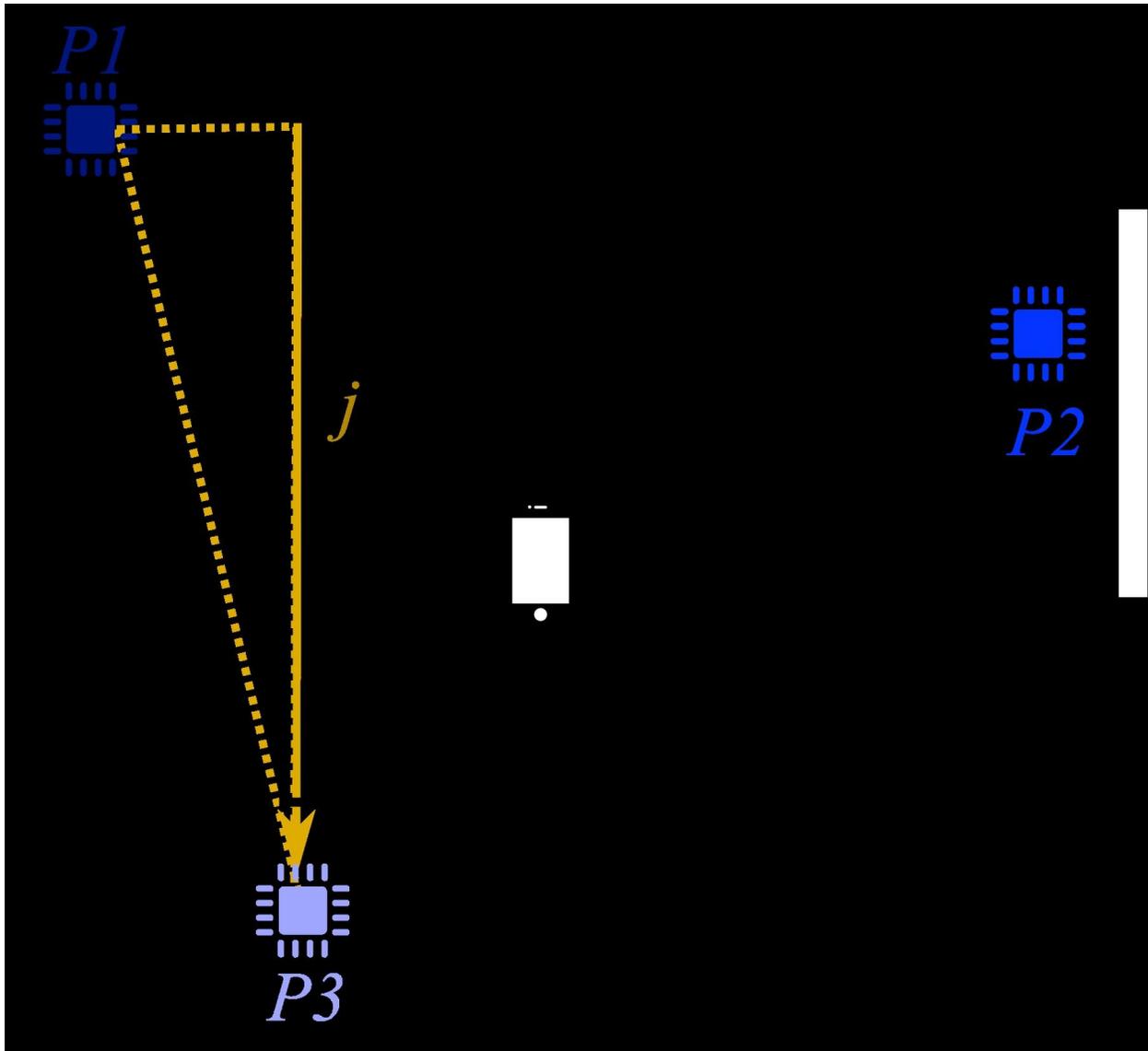
$\text{pow}(p3.y - p1.y - i * exy, 2)$

)

let  $eyy = (p3.y - p1.y - i * exy) / \text{eyy\_denominator}$

## Calculate Magnitude of Distance between P1 and P3

Calculate magnitude *j* of the distance between P1 and P3 in the *y* direction, showing how far apart they are (Figure 11-11).



**Figure 11-11. Derived Vertical component of distance between iBeacon 1 and iBeacon 3**

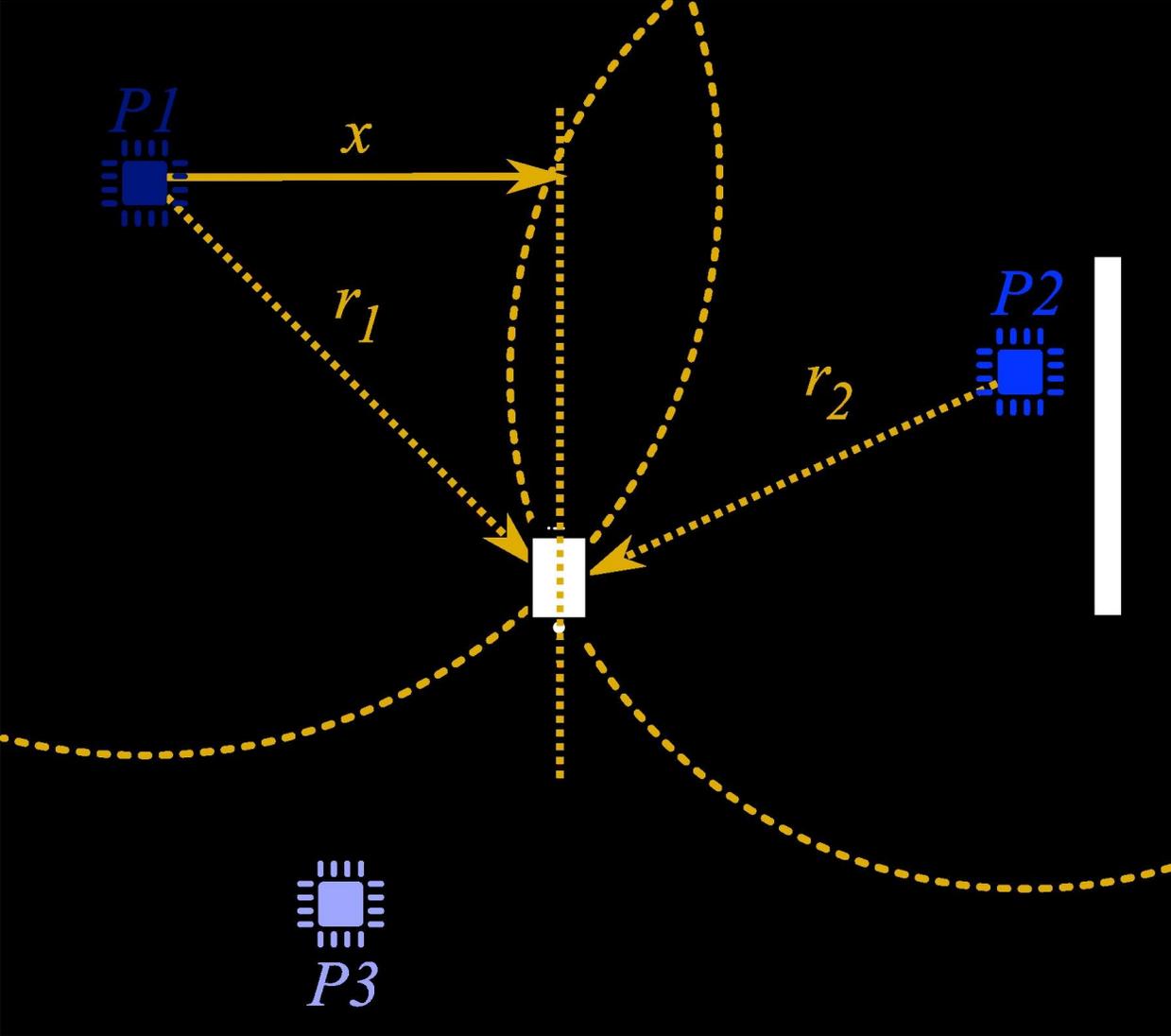
$$j = e_y \cdot (P3 - P1)$$

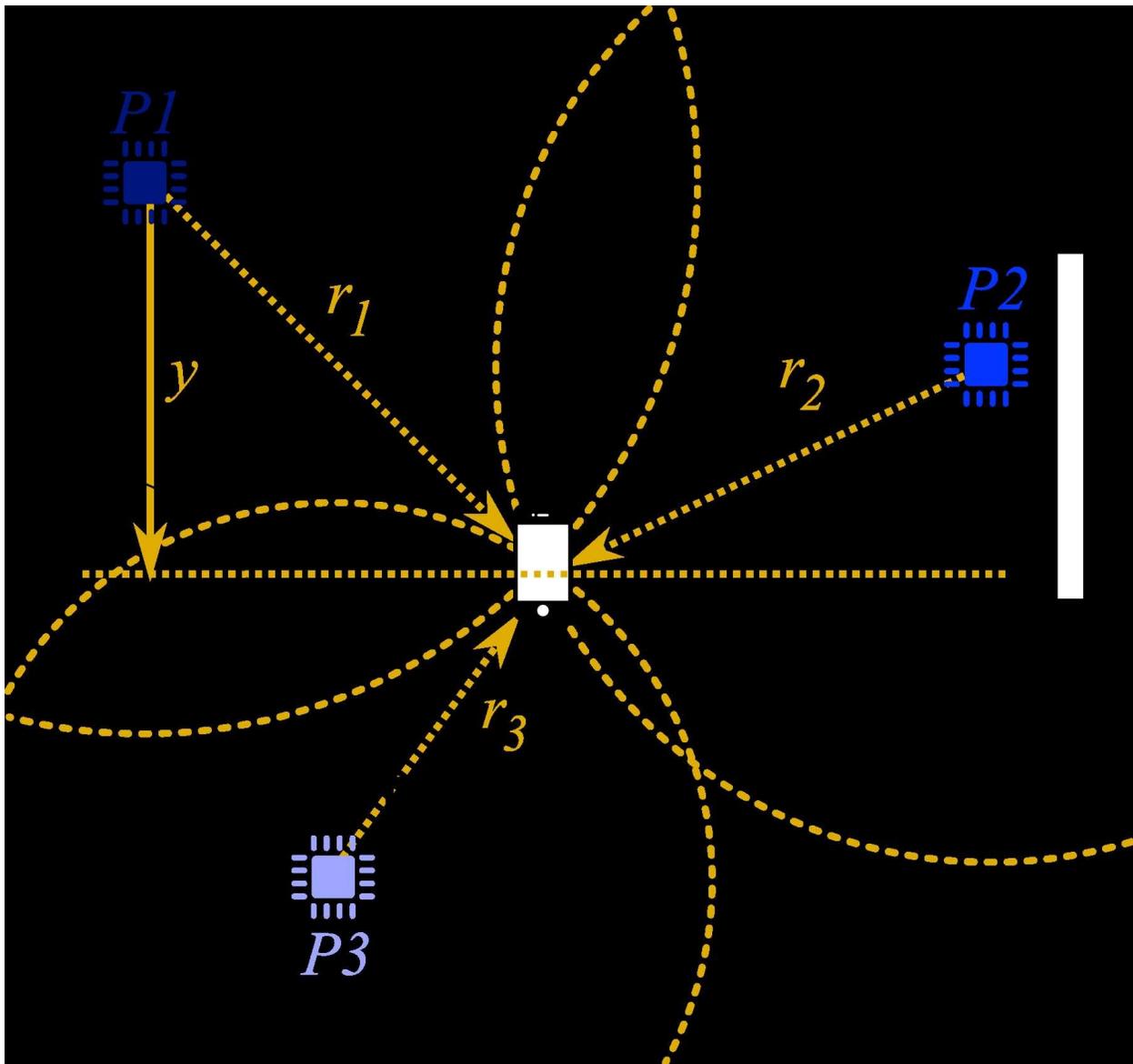
$$j = e_{yx}(P3_x - P1_x) + e_{yy}(P3_y - P1_y)$$

$$\text{let } j = \text{exy} * (\text{p2.x} - \text{p3.x}) + \text{eyy} * (\text{p3.y} - \text{p1.y})$$

### Find Relative Distances from Central to iBeacons

Use the relative distances from the Central to iBeacons P1, P2, and P3 to find relative distances x and y from each iBeacon (Figure 11-12).





**Figure 11-12. Derived positions of each iBeacon with horizontal position of Central**

This is done by solving for x and y in the following equations:

$x$

$=$

$$\frac{r^2 - r_1^2 + d_{12}^2}{2d}$$

$y$

$=$

$$\frac{r^2 - r_2^2 + i^2 + j^2}{2j}$$

$x$

let  $x = (\text{pow}(\text{Double}(r1), 2) - \text{pow}(\text{Double}(r2), 2) + \text{pow}(d, 2)) / (2 * d)$  let  $y = (\text{pow}(\text{Double}(r1), 2) - \text{pow}(\text{Double}(r3), 2) + \text{pow}(i, 2) + \text{pow}(j, 2)) / (2 * j) - i * x / j$

## Calculate Absolute Position of Central at P4

Finally, find the position of the Central at P4 by adding the distance between P4 and P1 to the relative position of P1 (Figure 11-13).

### Figure 11-13. Derived Central position

$P_4 = \sqrt{(x_1 + x_4)^2 + (y_1 + y_4)^2}$

```

x
y
+ y e^
y)

```

```

let finalX = p1.x + x * exx + y * eyx
let finalY = p1.y + x * exy + y * eyy
let centralPosition = new CGPoint(x: finalX, y: finalY)

```

That is how to trilaterate the position of a Central from three known iBeacon positions.

## Putting It All Together

The following code will create an App that graphs the position of nearby iBeacons, lists their properties, and attempts to find its own location relative to the iBeacons.

Create a new app called BeaconLocator. Create the following package structure and copy files from the previous examples where they exist. The final file structure should resemble this (Figure 11-14).

### Figure 11-14. Project Structure

#### Frameworks

Import the CoreLocation Framework (Figure 11-15).



Figure 11-15.

#### CoreLocation Framework linked into project

Import the CoreLocation APIs in the code header:

```
import CoreLocation
```

#### Models

The iBeacon class encapsulates a known iBeacon type and allows for the insertion of the physical location and transmission power of those iBeacons.

#### Example 11-1. Models/iBeacon.swift

```
import UIKit
import CoreLocation
```

```

class IBeacon: NSObject {
static let uuid = UUID(
uuidString: "e20a39f4-73f5-4bc4-a12f-17d1ad07a961")

// Bluetooth transmit power programmed into the device in dB let referenceRssi
= -57
// Beacon Peripheral
var beacon:CLBeacon!
// Beacon's position in physical space
var xPosition_m:Double = 0
var yPosition_m:Double = 0
/**

Initialize the IBeacon
*/
init(withBeacon beacon: CLBeacon) { self.beacon = beacon
}
/**

Get Beacon distance from Central in meters */
func getDistance() -> Int {

return beacon.proximity.rawValue }
/**

Compare beacons
- Parameters:
- to: An IBeacon to compare
- Returns: true if same beacon, false otherwise.
*/
func isEqual(to: IBeacon) -> Bool {
if let thisBeacon = beacon {

if let thatBeacon = to.beacon {
return thisBeacon.major == thatBeacon.major && \ thisBeacon.minor ==
thatBeacon.minor && \ thisBeacon.proximityUUID ==
thatBeacon.proximityUUID

}
}
}

```

```
return false
```

```
}
```

The CentralLocator trilaterates the user's device from known iBeacon locations.

### **Example 11-2. Models/CentralLocator.swift**

```
import UIKit
```

```
class CentralLocator: NSObject {
```

```
/**
```

```
Trilaterates a Central position from three known Beacon positions */
```

```
static func trilaterate(_ iBeacons: [IBeacon]) -> CGPoint? { if iBeacons.count < 3 {
```

```
return nil
```

```
}
```

```
// establish known values let p1 = iBeacons[0] let p2 = iBeacons[1] let p3 = iBeacons[2] let r1 = p1.getDistance() let r2 = p2.getDistance() let r3 = p3.getDistance()
```

```
//unit vector in a direction from point1 to point 2 let p1p2Distance = sqrt(pow(p2.xPosition_m - p1.xPosition_m, 2) + pow(p2.yPosition_m - p1.yPosition_m, 2) )
```

```
let exx = (p2.xPosition_m - p1.xPosition_m) / p1p2Distance let exy = (p2.yPosition_m - p1.yPosition_m) / p1p2Distance
```

```
//signed magnitude of the x component
```

```
let i = exx * (p3.xPosition_m - p1.xPosition_m) + exy * \
```

```
(p3.yPosition_m - p1.yPosition_m)
```

```
//the unit vector in the y direction.
```

```
let eyx = (p3.xPosition_m - p1.xPosition_m - i * exx) / sqrt(
```

```
pow(p3.xPosition_m - p1.xPosition_m - i * exx, 2) + pow(p3.yPosition_m - p1.yPosition_m - i * exy, 2) )
```

```
let eyy_denominator = sqrt(
```

```
pow(p3.xPosition_m - p1.xPosition_m - i * exx, 2) + pow(p3.yPosition_m - p1.yPosition_m - i * exy, 2) )
```

```
let eyy = (p3.yPosition_m - p1.yPosition_m - i * exy) / \ eyy_denominator
```

```
//the signed magnitude of the y component
```

```
let j = exy * (p2.xPosition_m - p3.xPosition_m) + eyy * \ (p3.yPosition_m -
```

```

p1.yPosition_m)
// coordinates
let x = (pow(Double(r1), 2) - pow(Double(r2), 2) + \ pow(p1p2Distance, 2) ) / (2
* p1p2Distance)
let y = (pow(Double(r1), 2) - pow(Double(r3), 2) + pow(i, 2) + \ pow(j, 2)) / (2 *
j) - i * x / j

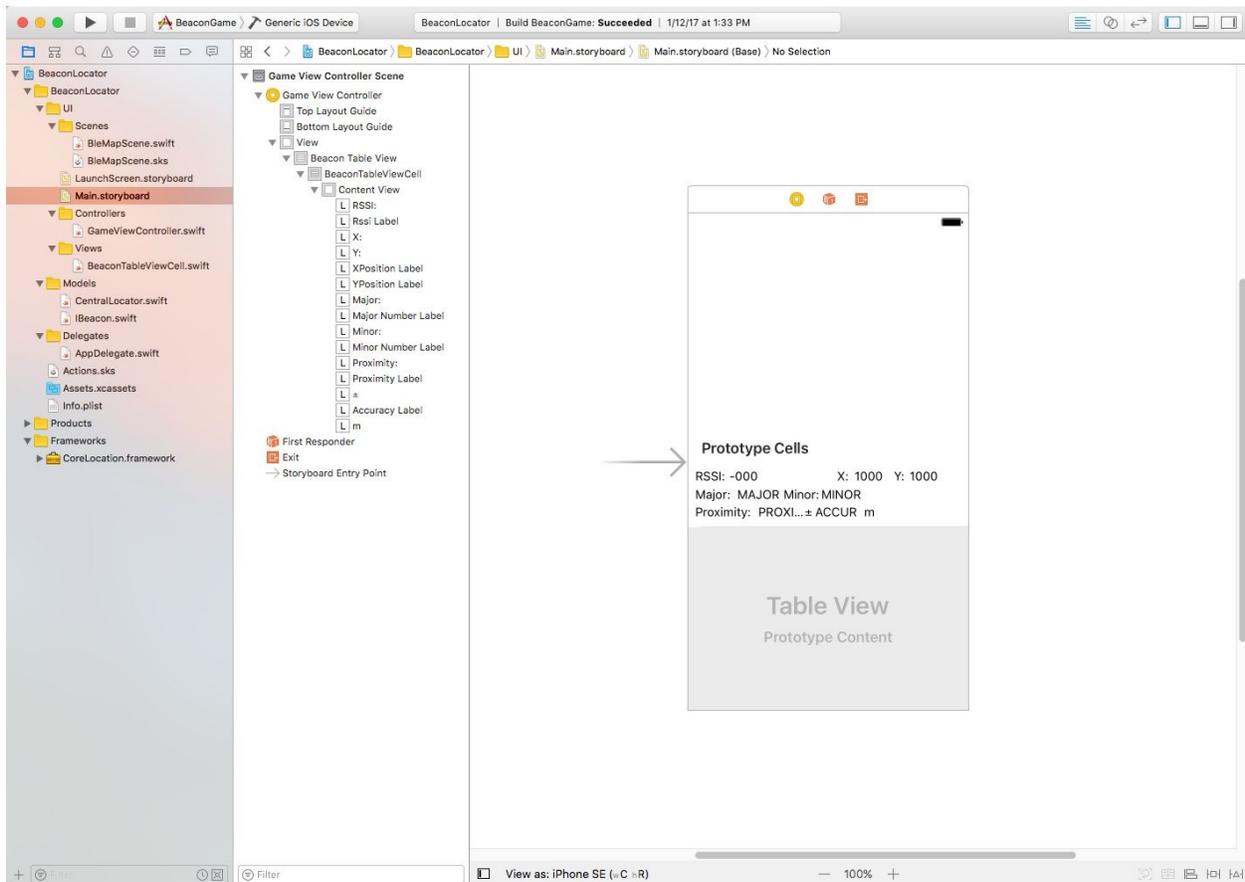
// result coordinates
let finalX = p1.xPosition_m + x * exx + y * eyx let finalY = p1.yPosition_m + x
* exy + y * eyy

return CGPoint(x: finalX, y: finalY) }

```

## Storyboard

Create and link a UITableView, UITableViewCell, and UILabels in the UIView in the Main.storyboard to create the App's user interface. Be sure that the UITableViewCell is of class "BeaconTableViewCell" and the Reuse Identifier is "BeaconTableViewCell" (Figure 11-16).



**Figure 11-16. Project Storyboard Views**

Create a `BeaconTableViewCell` that displays information about a discovered iBeacon in a series of `UILabel`s.

**Example 11-3. UI/Views/BeaconTableViewCell.swift**

```
import UIKit
import CoreLocation
class BeaconTableViewCell: UITableViewCell {

// MARK: UI Elements
@IBOutlet weak var rssiLabel: UILabel! @IBOutlet weak var xPositionLabel:
UILabel! @IBOutlet weak var yPositionLabel: UILabel! @IBOutlet weak var
majorNumberLabel: UILabel! @IBOutlet weak var minorNumberLabel:
UILabel! @IBOutlet weak var proximityLabel: UILabel! @IBOutlet weak var
accuracyLabel: UILabel!

/**
Render the cell with Beacon properties
*/
```

```

func renderBeacon(_ iBeacon: IBeacon) {
    majorNumberLabel.text = iBeacon.beacon.major.stringValue
    minorNumberLabel.text = iBeacon.beacon.minor.stringValue rssiLabel.text =
    String(iBeacon.beacon.rssi)
    proximityLabel.text = String(iBeacon.beacon.proximity.rawValue)
    accuracyLabel.text = String(iBeacon.beacon.accuracy)

    xPositionLabel.text = String(iBeacon.xPosition_m) yPositionLabel.text =
    String(iBeacon.yPosition_m) }

```

## Controllers

Create an `iBeaconMapLayout` class that displays `iBeacon` and `Central` positions on the screen.

### Example 11-4. UI/Controllers/GameViewController.swift

```

import UIKit
import SpriteKit import GameplayKit import CoreLocation

class GameViewController: UIViewController, UITableViewDataSource, \
CLLocationManagerDelegate {

    // MARK: UI Elements
    @IBOutlet weak var beaconTableView: UITableView! // Beacon Map Scene
    var scene:BleMapScene!
    // Beacon TableView Cell Reuse Identifier
    let beaconCellReuseIdentifier = "BeaconTableViewCell"

    // MARK: Beacons and Bluetooth

    // Location Manager includes Beacon functionality let locationManager =
    CLLocationManager() // discovered Beacons
    var foundBeacons = [IBeacon]()
    // Central's position
    var centralPosition:CGPoint!
    // number of beacons mapped
    var numBeaconsLoaded = 0
    // Beacon Region - the proximityUUID matches // the identifier of the iBeacon
    let beaconRegion = CLBeaconRegion(

    proximityUUID: IBeacon.uuid!,

```

```

identifier: "beaconRange")

/**
View Loaded
*/

override func viewDidLoad() {
super.viewDidLoad()
// load the MapScene
if let skView = view as! SKView? {

skView.isMultipleTouchEnabled = true
scene = BleMapScene(size: skView.bounds.size) scene.scaleMode = .aspectFill
skView.presentScene(scene)

}
// set up Location Manager
locationManager.delegate = self;
// If location services are not enabled, request authorization if
(CLLocationManager.authorizationStatus) != \

CLAuthorizationStatus.authorizedWhenInUse) {
locationManager.requestWhenInUseAuthorization()

} else {
startScanning()
}
}

/**
Start scanning for Beacons
*/

func startScanning() {
print("starting scan")
locationManager.startMonitoring(for: beaconRegion)
locationManager.startRangingBeacons(in: beaconRegion)

}
/**

```

```

Map the Beacons and Central if possible. */
func mapBeacons() {
for beacon in foundBeacons {

mapBeacon(beacon)
}
if foundBeacons.count > 3 {

// for this demo, we are done locating the Central
locationManager.stopMonitoring(for: beaconRegion)
locationManager.stopRangingBeacons(in: beaconRegion) mapCentralPosition()

}
}

/**
Put a Beacon on the map
*/

func mapBeacon(_ iBeacon: IBeacon) {
if scene != nil {
let position = CGPoint(
x: iBeacon.xPosition_m,
y: iBeacon.yPosition_m)
scene.addBeacon(
position: position,
radius: CGFloat(iBeacon.getDistance())) }
}

/**
Put the Central on the map
*/

func mapCentralPosition() {
centralPosition = CentralLocator.trilaterate(foundBeacons) if let centralPosition
= centralPosition {

print("mapping location")
if let scene = scene {
scene.addCentral(position: centralPosition) } }
}

```

```

}
// MARK: CLLocationManagerDelegate callbacks

/**
Beacons discovered by Location Manager */

func locationManager(
    _ manager: CLLocationManager, didRangeBeacons beacons: [CLBeacon], in
    region: CLBeaconRegion)

{
    // inspect previously discovered beacons
    print(beacons.count)
    for beacon in beacons {

        print(beacon.proximityUUID.uuidString) let iBeacon = IBeacon(withBeacon:
        beacon) // if a beacon is already discovered, update. // Otherwise add to the list
        var beaconFound = false
        for i in 0..

```

```

{
if (status == .authorizedAlways) ||

(status == CLAuthorizationStatus.authorizedWhenInUse) { print("location
manager authorized")
if CLLocationManager.isMonitoringAvailable(

for: CLBeaconRegion.self) {
if CLLocationManager.isRangingAvailable() { startScanning()

} else {
print("ranging unavailable")
}
} else {
print("location manager unauthorized")
}
}

// MARK: UITableViewDataSource

/**
Number of Table sections
*/

func numberOfSections(in tableView: UITableView) -> Int { return 1
}

/**
Number of iBeacons in table */

func tableView(
_ tableView: UITableView,
numberOfRowsInSection section: Int) -> Int

{
return foundBeacons.count
}

/**

```

Render Beacon Table cell

\*/

```
func tableView(  
_ tableView: UITableView,  
cellForRowAt indexPath: IndexPath) -> UITableViewCell
```

```
{  
// create a new Beacon Table View cell let cell =  
tableView.dequeueReusableCell(  
  
withIdentifier: beaconCellReuseIdentifier, for: indexPath) as!  
BeaconTableViewCell // render cell  
let iBeacon = foundBeacons[indexPath.row]  
cell.renderBeacon(iBeacon)  
return cell  
}
```

/\*\*

User selected a cell

\*/

```
private func tableView(  
tableView: UITableView,  
didSelectRowAtIndexPath indexPath: IndexPath)
```

```
{  
tableView.deselectRow(at: indexPath, animated: true)  
}
```

## Scenes

Create a BeaconMapScene, which will plot the iBeacon and Central locations on a map.

### Example 11-5. Scenes/BeaconMapScene.swift

```
import SpriteKit  
import GameplayKit  
class BleMapScene: SKScene {
```

```
// sprite scale
```

```
let iconScale = CGFloat(0.1)
```

```
// size of room in meters - must same x/y ratio as screen bounds let roomBounds
```

```
= CGSize(width: 50, height: 37)
```

```
/**
```

```
Initialize Map Scene with a coder (required but not implemented) */
```

```
required init?(coder aDecoder: NSCoder) {  
    fatalError("init(coder:) has not been implemented")  
}
```

```
/**
```

```
Initialize Map Scene
```

```
*/
```

```
override init(size: CGSize) { super.init(size: size)  
    backgroundColor = SKColor.white
```

```
}
```

```
/**
```

```
Update scene
```

```
*/
```

```
override func update(_ currentTime: TimeInterval) { super.update(currentTime)  
}
```

```
/**
```

```
Add Central to map
```

```
- Parameters:
```

```
- position: the x,y location of the central in meters from origin
```

```
*/
```

```
func addCentral(position: CGPoint) {
```

```
    // load sprite
```

```
    let central = SKSpriteNode(imageNamed: "central")
```

```
    // scale and position sprite
```

```
    central.size = CGSize(  
    width: central.size.width * iconScale,  
    height: central.size.height * iconScale)
```

```
    central.position = getMapPosition(position)
```

```
    // add sprite to map
```

```
    addChild(central)
```

```
}
```

```
/**
```

```
Add a Beacon to the Map
```

```
- Parameters:
```

```
- position: the x,y location of the Beacon in meters from origin
```

```
- radius: the beacon's distance from the central in meters
```

```
*/
```

```
func addBeacon(position: CGPoint, radius: CGFloat) {
```

```
// load sprite
```

```
let peripheral = SKSpriteNode(imageNamed: "peripheral") // scale sprite
```

```
peripheral.size = CGSize(
```

```
width: peripheral.size.width * iconScale,
```

```
height: peripheral.size.height * iconScale)
```

```
let mapPosition = getMapPosition(position)
```

```
// position and add sprite to map
```

```
peripheral.position = mapPosition
```

```
addChild(peripheral)
```

```
// add a circle for central's distance from beacon let circle = SKShapeNode(
```

```
circleOfRadius: CGFloat(getMapScale(radius))) circle.position = mapPosition
```

```
circle.strokeColor = SKColor.blue
```

```
addChild(circle)
```

```
}
```

```
/**
```

```
Convert Real world coordinates into map coordinates */
```

```
private func getMapPosition(_ position: CGPoint) -> CGPoint { var xPosition =
```

```
getMapScale(position.x)
```

```
var yPosition = getMapScale(position.y)
```

```
// offset by 50% of screen
```

```
xPosition -= 0 // (size.width / 2)
```

```
yPosition -= (size.height + 50)
```

```
yPosition *= -1
```

```
return CGPoint(x: xPosition, y: yPosition)
```

```
}
```

```
/**
```

```
Convert real world distances into map distances */
```

```
private func getMapScale(_ scalar: CGFloat) -> CGFloat { return scalar *  
size.width / roomBounds.width  
}  
}
```

The resulting app will be able to locate iBeacons and, if there are three or more iBeacons nearby, it can approximate its own location (Figure 11-17).



**Figure 11-17. App screen showing derived iBeacon and Central positions**

# Programming the Peripheral

An iBeacon is relatively simple to implement in iOS using the CoreLocation and CoreBluetooth Frameworks.

## Import CoreBluetooth and CoreLocation

Link the CoreBluetooth and CoreLocation Frameworks from the project Settings (Figure 11-18).



**Figure 11-18. CoreBluetooth and CoreLocation Frameworks linked into project**

Import the CoreBluetooth and CoreLocation APIs in the code header to access both the Bluetooth and iBeacon APIs. The working class must also implement CBPeripheralManagerDelegate:

```
Import CoreBluetooth
import CoreLocation
```

```
class ViewController: UIViewController, CBPeripheralManagerDelegate { ...
}
```

## Implementation

Instantiate the CBPeripheralManager.

```
// Instantiate a CBPeripheralManager with the current object as the delegate let
peripheralManager = CBPeripheralManager(
delegate: self,
queue: nil, options: nil)
```

Define the CBPeripheralManagerDelegate callback that's triggered when the

Bluetooth radio activates:

```
func peripheralManagerDidUpdateState(_ peripheral: CBPeripheralManager) {
    peripheralManager = peripheral
    switch peripheral.state {
    case CBManagerState.poweredOn:

        startAdvertising()
    case CBManagerState.poweredOff:
        stopAdvertising()
    default: break
    }
    delegate?.iBeaconPeripheral?(stateChanged: peripheral.state) }
```

Each iBeacon uses a UUID plus Major and Minor numbers for identification. iBeacons belonging to the same organizational unit also share the same UUID. For instance all iBeacons belonging to a grocery store chain.

// UUID

```
let beaconUuid = UUID(uuidString: "e20a39f4-73f5-4bc4-a12f-17d1ad07a961")
```

Major numbers are typically used to identify iBeacons sharing the same purpose typically have the same Major number, ones that identify each grocery store in that chain.

// Major Number

```
let majorNumber:CLBeaconMajorValue = UInt16(1122)
```

The Minor numbers are typically unique between Major numbers, so that a user can track their movement around the grocery store using the iBeacon minor numbers for identification.

// Minor Number

```
let minorNumber:CLBeaconMinorValue = UInt16(3344)
```

The transmission power is important in both in setting the range of the iBeacon and in allowing the Central to determine how close it is to an iBeacon

// Transmission power

```
let transmissionPower_db:NSNumber? = -56
```

Create an iBeacon region:

// Range identifier

```
let rangeIdentifier = "customUniqueIdentifier"
```

```
let beaconRegion = CLBeaconRegion( proximityUUID: beaconUuid!, major:
majorNumber,
minor: minorNumber,
identifier: rangeIdentifier)
```

Based on the iBeacon Region created earlier, create the Advertisement Data that includes the iBeacon UUID, Major Number, Minor Number, and transmission power:

```
let advertisementDictionary = beaconRegion.peripheralData(
withMeasuredPower: transmissionPower_db)
var advertisementData = [String: Any]()
for (key, value) in advertisementDictionary {
advertisementData[key as! String] = value
}
peripheralManager.startAdvertising(advertisementData as [String:Any])
```

Define the callback function that's called when the iBeacon begins advertising:

```
func peripheralManagerDidStartAdvertising( _ peripheral:
CBPeripheralManager, error: Error?)

{
if error != nil {
print ("Error advertising peripheral")
print(error.debugDescription)
}
self.peripheralManager = peripheral
delegate?.iBeaconPeripheral?(startedAdvertising: error)
}
```

Stop advertising when the user closes the App:

```
override func viewWillDisappear(_ animated: Bool) {
peripheralManager.stopAdvertising()
}
```

## Putting It All Together

Create a new project called iBeacon with the following project structure (Create a new project called iBeacon with the following project structure ( 18).

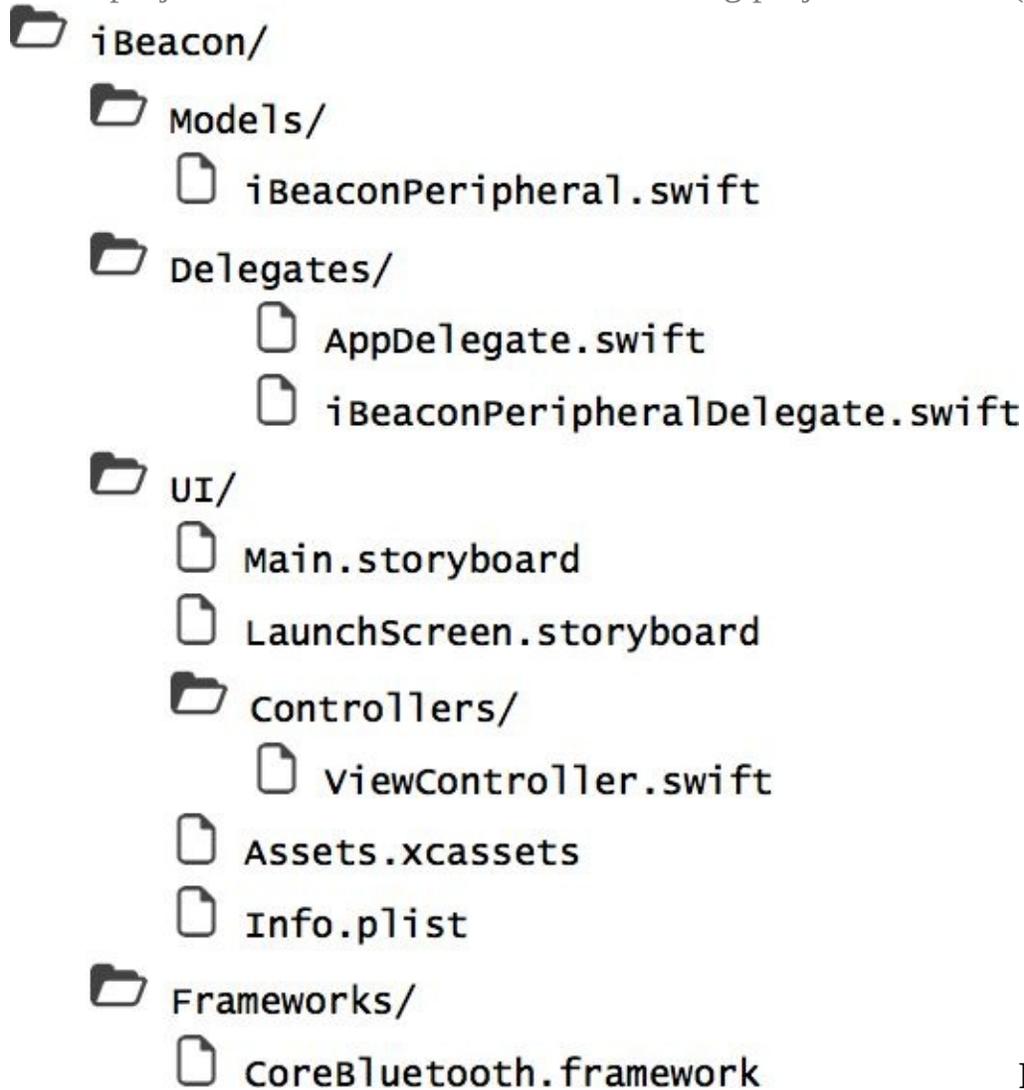


Figure 11-18.

## Project Structure

### Frameworks

Import the CoreBluetooth and CoreLocation Frameworks (Figure 11-19):



Figure 11-19.

## CoreBluetooth and CoreLocation Frameworks linked into project

Import the CoreBluetooth and CoreLocation APIs in the code header:

```
import CoreBluetooth import CoreLocation
```

## Models

Create an `iBeaconPeripheral` class that has a Major Number, Minor Number, and UUID. It will begin advertising as soon as the Bluetooth radio turns on:

### Example 11-6. Models/iBeaconPeripheral.swift

```
import UIKit
import CoreBluetooth import CoreLocation

class IBeaconPeripheral: NSObject, CBPeripheralManagerDelegate {
// MARK: iBeacon properties

// Advertized name
let advertisingName = "LedRemote"
// Device identifier
let beaconIdentifier = "com.example.ibeacon" // UUID
let beaconUuid = UUID(

uuidString: "e20a39f4-73f5-4bc4-a12f-17d1ad07a961") // Major Number
let majorNumber:CLBeaconMajorValue = UInt16(1122) // Minor Number
let minorNumber:CLBeaconMinorValue = UInt16(3344) // Transmission power
let transmissionPower_db:NSNumber? = -56
// MARK: Peripheral State

// Beacon Region
var beaconRegion: CLBeaconRegion!
// Peripheral Manager
var peripheralManager:CBPeripheralManager! // Connected Central
var central:CBCentral!
// delegate
var delegate:IBeaconPeripheralDelegate!

/**
Initialize BlePeripheral with a corresponding Peripheral

- Parameters:
- delegate: The BlePeripheralDelegate
- peripheral: The discovered Peripheral

*/
```

```

init(delegate: IBeaconPeripheralDelegate?) { super.init()
self.delegate = delegate
print("initializing beacon region") if let beaconUuid = beaconUuid {

beaconRegion = CLBeaconRegion(
proximityUUID: beaconUuid, major: majorNumber,
minor: minorNumber,
identifier: beaconIdentifier)

print("initializing peripheral manager") peripheralManager =
CBPeripheralManager( delegate: self,
queue: nil,
options: nil)
} else {
print("invalid UUID")
}

/**
Stop advertising, shut down the Peripheral */

func stopAdvertising() {
peripheralManager.stopAdvertising()
delegate?.iBeaconPeripheralStoppedAdvertising?()

}
/**
Start Bluetooth Advertising. This must be after building the GATT profile

*/
func startAdvertising() {
print("loading peripheral data")
let advertisementDictionary = beaconRegion.peripheralData(

withMeasuredPower: transmissionPower_db)
print("building advertisement data")
var advertisementData = [String: Any]()
for (key, value) in advertisementDictionary {

advertisementData[key as! String] = value
}
}

```

```

print("beginning advertising")
peripheralManager.startAdvertising(
    advertisementData as [String:Any])
}
// MARK: CBPeripheralManagerDelegate

/**
Peripheral will become active
*/

func peripheralManager(
    _ peripheral: CBPeripheralManager, willRestoreState dict: [String : Any])
{
    print("restoring peripheral state")
}

/**
Peripheral started advertising
*/

func peripheralManagerDidStartAdvertising( _ peripheral:
    CBPeripheralManager, error: Error?)
{
    if error != nil {
        print ("Error advertising peripheral")
        print(error.debugDescription)
    }
    self.peripheralManager = peripheral
    delegate?.iBeaconPeripheral?(startedAdvertising: error)
}

/**
Bluetooth Radio state changed
*/

func peripheralManagerDidUpdateState( _ peripheral: CBPeripheralManager)

```

```

{
peripheralManager = peripheral
switch peripheral.state {
case CBManagerState.poweredOn:
startAdvertising()
case CBManagerState.poweredOff:
stopAdvertising()
default: break
}
delegate?.iBeaconPeripheral?(stateChanged: peripheral.state)
}

```

## Delegates

Create an `iBeaconPeripheralDelegate` class that relays `iBeacon` state changes:

### Example 11-7. Delegates/`iBeaconPeripheralDelegate.swift`

```

import UIKit
import CoreBluetooth
@objc protocol IBeaconPeripheralDelegate : class {
/**
iBeacon State Changed

- Parameters:
- rssi: the RSSI
- blePeripheral: the BlePeripheral

*/
@objc optional func iBeaconPeripheral( stateChanged state: CBManagerState)
/**
iBeacon started advertising

- Parameters:
- error: the error message, if any
*/
@objc optional func iBeaconPeripheral(startedAdvertising error: Error?)

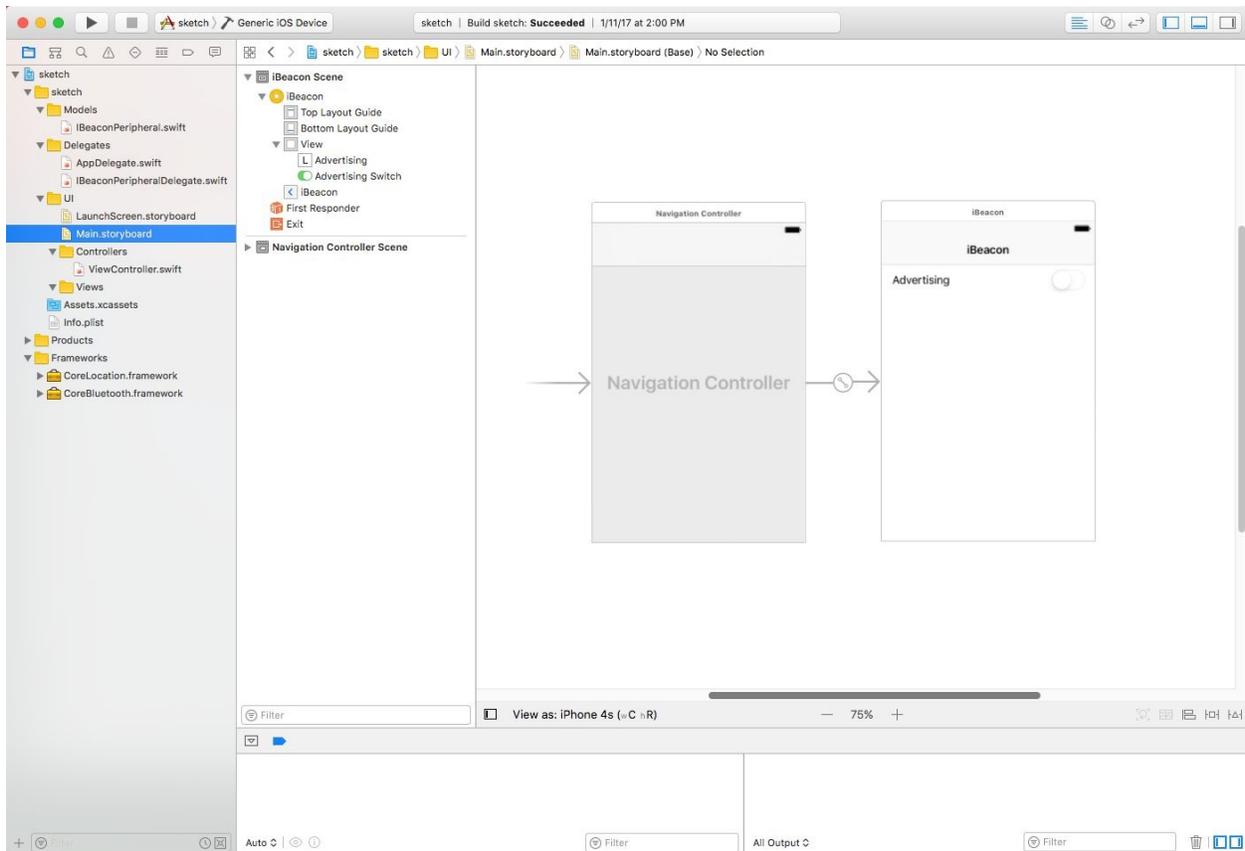
/**
iBeacon stopped advertising
*/

```

@objc optional func iBeaconPeripheralStoppedAdvertising()

### **Storyboard**

Create the UISwitches, UITextViews, and UILabels in the UIView in the Main.storyboard to create the App's user interface (Figure 11-20).



**Figure 11-20. Project Storyboard Controllers**

The ViewController instantiates the iBeacon, starts Advertising as soon as the Bluetooth radio turns on, and shows the Advertising state in a UISwitch :

**Example 11-8. UI/Controllers/ViewController.swift**

```
import UIKit
import CoreBluetooth
import AVFoundation
class ViewController: UIViewController, IBeaconPeripheralDelegate {

// MARK: UI Elements
@IBOutlet weak var advertisingSwitch: UISwitch!
// MARK: BlePeripheral
// BlePeripheral
var iBeacon:IBeaconPeripheral!

/**
UIView loaded
*/
```

```

override func viewDidLoad() { super.viewDidLoad()
}

/**
View appeared. Start the Peripheral
*/

override func viewDidAppear(_ animated: Bool) { iBeacon =
IBeaconPeripheral(delegate: self)
}

/**
View will appear. Stop transmitting random data */

override func viewWillDisappear(_ animated: Bool) { iBeacon.stopAdvertising()
}

/**
View disappeared. Stop advertising
*/

override func viewDidDisappear(_ animated: Bool) {
advertisingSwitch.setOn(false, animated: true) }
// MARK: IBeaconPeripheralDelegate
/**
RemoteLed state changed

- Parameters:
- state: the CBManagerState representing the new state */

func iBeaconPeripheral(stateChanged state: CBManagerState) { switch (state) {
case CBManagerState.poweredOn:

print("Bluetooth on")
case CBManagerState.poweredOff:
print("Bluetooth off")
default:
print("Bluetooth not ready yet...")
}
}
}

```

```

/**
RemoteLed started advertising

- Parameters:
- error: the error message, if any
*/

func iBeaconPeripheral(startedAdvertising error: Error?) { if error != nil {
print("Problem starting advertising: " + error.debugDescription)

} else {
print("advertising started")
advertisingSwitch.setOn(true, animated: true)

}
}

/**
RemoteLed started advertising
*/

func iBeaconPeripheralStoppedAdvertising() { print("advertising stopped")
}
}

```

The resulting app can broadcast as a single iBeacon (Figure 11-21).

●●●○○ Fido 3G

16:38

⌘ 100%  ⚡

## iBeacon

Advertising



**Figure 11-21. App screen showing Advertising state of the iBeacon**

## Example code

The code for this chapter is available online  
at: <https://github.com/BluetoothLowEnergyIniOSSwift/Chapter11>

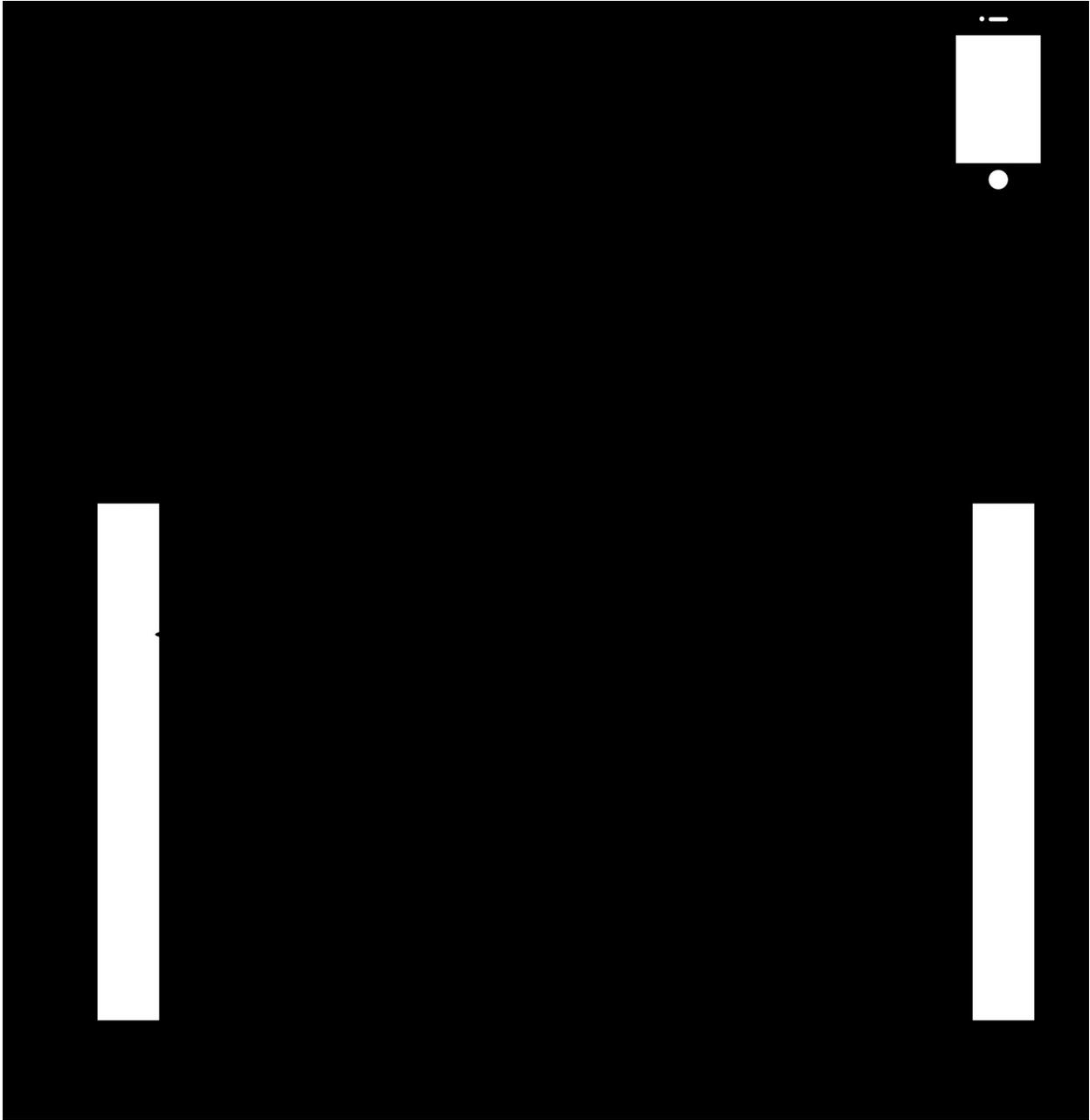
## **Project: Echo Client and Server**

An Echo Server is the “Hello World” of network programming. It has the minimum features required to transmit, store, and respond to data on a network - the core features required for any network application.

And yet it must support all the features you’ve learned so far in this book - advertising, reads, writes, notifications, segmented data transfer, and encryption. It’s a sophisticated program!

The Echo Server works like this:

In this example, the Peripheral acts as a server, the “Echo Server” and the Central acts as a client (Figure 12-1).



**Figure 12-1. How an Echo Server works**

This project is based heavily on code seen up until Chapter 10, so there shouldn't be any surprises.

## Programming the Central

The Echo Client will read and write messages on a Characteristic and will notify the Central when it is ready for new messages.

The Echo Server sends and receives text. Text is complicated because computers communicate using binary data, not text. As a result, there are a couple things that need to be done to protect the data from errors.

### Data Formatting

Bluetooth has a 20-byte maximum packet size. Longer messages must be divided and sent in parts. Both the client and server need a way to know if the data being transmitted belongs to part of a larger transmission.

One easy way to do this with text is to append a newline character to the end of the outbound transmission.

```
let partialMessage = "Hello World" let message = partialMessage + "\n"
```

When the message is broken apart for transmission, the parts are reassembled properly and the new line at the end separates new messages into new lines. Everything else works the same way as any other Bluetooth app.

### GATT Profile

The GATT Profile will be set up as a digital input/output under the Automation IO (0x1815) Service, with commands to and responses from the Peripheral on separate Characteristics.

```
// the Service UUID
```

```
static let serviceUuid = CBUUID(string: "180C")
```

```
// The Characteristic UUID used to read echoes from the Peripheral static let  
responseCharacteristicUuid = CBUUID(string: "2A56")
```

```
// The Characteristic UUID used to write text to the Peripheral static let  
writeCharacteristicUuid = CBUUID(string: "2A57")
```

### Enable Bluetooth Radio

To turn on the Bluetooth radio programmatically, also enable Bluetooth admin permissions:

let centralManager = CBCentralManager(delegate: self, queue: nil) When the Bluetooth radio turns on, begin scanning:

```
func centralManagerDidUpdateState(_ central: CBCentralManager) {  
  
    switch (central.state) {  
    case .poweredOn:  
        central.scanForPeripherals(withServices: [serviceUuid], options: nil) default:  
    }  
    }  
}
```

Whenever a device is discovered, the centralManager didDiscover callback is triggered by the CBPeripheralManager. If the Advertised name matches the desired EchoServer name, connect:

```
func getNameFromAdvertisementData(  
    advertisementData: [String : Any]) -> String?  
{  
    // grab the kCBAAdvDataLocalName from the advertisementData // to see if  
    // there's an alternate broadcast name if  
    advertisementData["kCBAAdvDataLocalName"] != nil { return  
        (advertisementData["kCBAAdvDataLocalName"] as! String)  
    }  
    return nil  
}
```

```
func centralManager(  
    _ central: CBCentralManager,  
    didDiscover peripheral: CBPeripheral,  
    advertisementData: [String : Any],  
    rssi RSSI: NSNumber)  
  
{  
    // find the advertised name  
    if let advertisedName = getNameFromAdvertisementData(  
  
        advertisementData: advertisementData) {  
        if advertisedName == "EchoServer" {
```

```
centralManager.connect(peripheral, options: nil) }  
}  
}
```

## Connecting

Once a Peripheral is connected, the centralManager didConnect callback is triggered by the CBPeripheralManager. Use this to build a map of the GATT profile, starting with the Services:

```
var connectedPeripheral:CBPeripheral! func centralManager(  
_ central: CBCentralManager,  
didConnect peripheral: CBPeripheral) {  
// store a copy of the connected Peripheral so it isn't tossed  
connectedPeripheral = peripheral;  
connectedPeripheral.delegate = self;  
peripheral.discoverServices(serviceUuid) }
```

## Discovering GATT

Once services are discovered, the peripheral didDiscoverServices callback is triggered by the CBPeripheralDelegate. Since the EchoServer is known to have a Service with UUID 0x180C, search for that service and request discovery of the two known Characteristics, 0x2a56 and 0x2a57:

```
func peripheral(  
_ peripheral: CBPeripheral, didDiscoverServices error: Error?)  
  
{  
// update the local copy of the Peripheral  
connectedPeripheral = peripheral  
if error != nil {  
  
// error  
} else {  
for service in peripheral.services! {  
if service.uuid == serviceUuid {  
connectedPeripheral.discoverCharacteristics( [readCharacteristicUuid,  
writeCharacteristicUuid],
```

```
for: service)
}
}
}
```

A list of Characteristics for each Service will come back in the peripheral `didDiscoverCharacteristicsFor` callback in `CBPeripheralManagerDelegate`. Use this to subscribe to the Read Characteristics and to save references to each Characteristic:

```
func peripheral(
    _ peripheral: CBPeripheral,
    didDiscoverCharacteristicsFor service: CBService, error: Error?)

{
    // update local copy of the peripheral
    connectedPeripheral = peripheral
    connectedPeripheral.delegate = self
    // grab the service
    let serviceIdentifier = service.uuid.uuidString
    if let characteristics = service.characteristics {

        for characteristic in characteristics {
            if characteristic.uuid.uuidString == writeCharacteristicUuid { writeCharacteristic
            = characteristic

            } else if characteristic.uuid.uuidString == \
            readCharacteristicUuid {
                readCharacteristic = characteristic
                connectedPeripheral.setNotifyValue(

                true,
                for: characteristic)
            }
        }
    }
}
```

## Writing Messages

To write a message to the Characteristic, convert the String message into a Data object and write the value using the `CBPeripheral.writeValue` method:

```
// Flow control response
let flowControlMessage = "ready"
// outbound value to be sent to the Characteristic var outboundByteArray:
[UInt8]!
// packet offset in multi-packet value var packetOffset = 0

func writeValue(value: String) {
// Append a newline to the end of the string for formatting purposes let
writeableValue = value + "\n\0"
packetOffset = 0
// get the data for the current offset
outboundByteArray = Array(writeableValue.utf8)
// begin the write process
writePartialValue(value: outboundByteArray, offset: packetOffset)

}
```

```
func writePartialValue(value: [UInt8], offset: Int) {
// don't go past the total value size
var end = offset + characteristicLength
if end > outboundByteArray.count {

end = outboundByteArray.count
}
let transmissibleValue = Data(Array(outboundByteArray[offset..
```

## Receiving Messages

When the EchoServer Peripheral responds with on the Read Characteristic (0x2a56), the peripheral `didUpdateValueFor` callback is triggered from the `CBPeripheralManagerDelegate`. Send the next packet of data in response, until there is nothing left to send:

```

func peripheral(
  _ peripheral: CBPeripheral,
  didUpdateValueFor characteristic: CBCharacteristic, error: Error?)

{
  if let value = characteristic.value {
    let byteArray = [UInt8](value)
    if let stringValue = String(data: value, encoding: .ascii) {

      packetOffset += characteristicLength
      if packetOffset < outboundByteArray.count { writePartialValue(
        value: outboundByteArray,
        offset: packetOffset)
      }
    }
  }
}

```

## Putting It All Together

The following code will create an App that connects to a Peripheral, allows a user to type a message, send that message to the Peripheral, and then prints the Peripheral's response.

Create a new project called EchoClient with the following structure (Figure 12-2).



Figure 12-2.

## Project Structure

### Frameworks

Import the CoreBluetooth Framework (Figure 12-3).

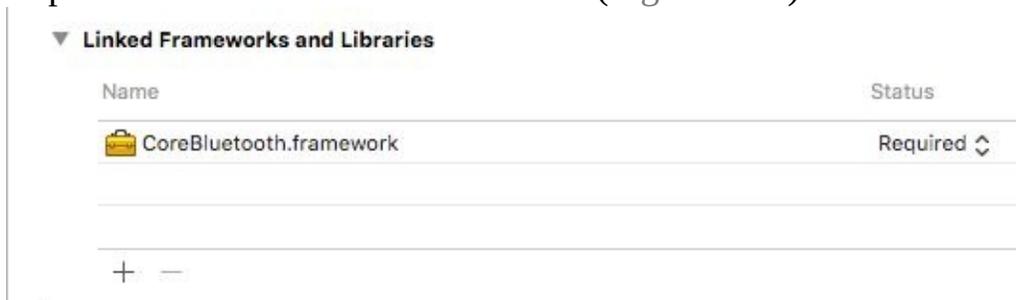


Figure 12-3.

### CoreBluetooth Framework linked into project

Import the CoreBluetooth APIs in the code header:

```
import CoreBluetooth
```

### Models

The BleCommManager turns the Bluetooth radio on and scans for nearby Peripherals.

### **Example 12-1. Models/EchoServer.swift**

```
import UIKit
import CoreBluetooth
class EchoServer:NSObject, CBPeripheralDelegate {
// MARK: Peripheral properties

// The Broadcast name of the Perihperal
static let advertisedName = "EchoServer"
// the Service UUID
static let serviceUuid = CBUUID(string: "180C")
// The Characteristic UUID used to write text to the Peripheral static let
readCharacteristicUuid = CBUUID(string: "2A56") // the Characteristic UUID
used to read echoes from the Peripheral static let writeCharacteristicUuid =
CBUUID(string: "2A57") // the size of the characteristic
let characteristicLength = 20

// MARK: Flow control
// FLOW control response
let flowControlMessage = "ready"

// outbound value to be sent to the Characteristic var outboundByteArray:
[UInt8]!
// packet offset in multi-packet value var packetOffset = 0

// MARK: connected device

// EchoServerDelegate
var delegate:EchoServerDelegate! // connected Peripheral
var connectedPeripheral:CBPeripheral! // connected Characteristic
var readCharacteristic:CBCharacteristic! var
writeCharacteristic:CBCharacteristic

/**
Initialize EchoServer with a corresponding Peripheral

- Parameters:
- delegate: The EchoServerDelegate
- peripheral: The discovered Peripheral
```

```

*/
init(delegate: EchoServerDelegate, peripheral: CBPeripheral) { super.init()
connectedPeripheral = peripheral
connectedPeripheral.delegate = self
self.delegate = delegate

}

/**
Notify the EchoServer that the peripheral has been connected */

func connected(peripheral: CBPeripheral) {
connectedPeripheral = peripheral
connectedPeripheral.delegate = self
connectedPeripheral.discoverServices([EchoServer.serviceUuid])

}

/**
Get a advertised name from an advertisementData packet. This may be different
than the actual Peripheral name */

static func getNameFromAdvertisementData(
advertisementData: [String : Any]) -> String?
{
// grab thekCBAAdvDataLocalName from the advertisementData // to see if
there's an alternate broadcast name
if advertisementData["kCBAAdvDataLocalName"] != nil { return
(advertisementData["kCBAAdvDataLocalName"] as! String) }
return nil
}

/**
Write a text value to the EchoServer

- Parameters:
- value: the value to write to the connected Characteristic
*/
func writeValue(value: String) {
// get the characteristic length
let writableValue = value + "\n\0"

```

```

packetOffset = 0
// get the data for the current offset
outboundByteArray = Array(writeableValue.utf8) writePartialValue(value:
outboundByteArray, offset: packetOffset) }

/**
Write a partial value to the EchoServer
- Parameters:
- value: the full value to write to the connected Characteristic
- offset: the packet offset

*/
func writePartialValue(value: [UInt8], offset: Int) { // don't go past the total value
size
var end = offset + characteristicLength
if end > outboundByteArray.count {

end = outboundByteArray.count
}
let transmissibleValue = \

Data(Array(outboundByteArray[offset..

```

- Parameters:

- characteristic: The Characteristic to test

- returns: True if characteristic is readable \*/

```
static func isCharacteristic(  
isReadable characteristic: CBCharacteristic) -> Bool {  
if (characteristic.properties.rawValue &
```

```
CBCharacteristicProperties.read.rawValue) != 0 {  
print("readable")  
return true  
}  
return false  
}
```

/\*\*

Check if Characteristic is writeable

- Parameters:

- characteristic: The Characteristic to test

- returns: True if characteristic is writeable \*/

```
static func isCharacteristic(  
isWritable characteristic: CBCharacteristic) -> Bool
```

```
{  
print("testing if characteristic is writeable")  
if (characteristic.properties.rawValue &
```

```
CBCharacteristicProperties.write.rawValue) != 0 ||  
(characteristic.properties.rawValue &  
CBCharacteristicProperties.writeWithoutResponse.rawValue) != 0
```

```
{  
print("characteristic is writeable") return true
```

```
}  
print("characteristic is not writeable") return false
```

```
}  
/**
```

Check if Characteristic is writeable with response

- Parameters:

- characteristic: The Characteristic to test

- returns: True if characteristic is writeable with response \*/

```
static func isCharacteristic(  
isWritableWithResponse characteristic: CBCharacteristic) -> Bool {  
if (characteristic.properties.rawValue &
```

```
CBCharacteristicProperties.write.rawValue) != 0 {  
return true }  
return false }
```

/\*\*

Check if Characteristic is writeable without response

- Parameters:

- characteristic: The Characteristic to test

- returns: True if characteristic is writeable without response

\*/

```
static func isCharacteristic(  
isWritableWithoutResponse characteristic: CBCharacteristic) -> Bool {  
if (characteristic.properties.rawValue &
```

```
CBCharacteristicProperties.writeWithoutResponse.rawValue) != 0 {  
return true }  
return false }
```

/\*\* Check if Characteristic is notifiable

- Parameters:

- characteristic: The Characteristic to test

- returns: True if characteristic is notifiable \*/

```
static func isCharacteristic(  
isNotifiable characteristic: CBCharacteristic) -> Bool {  
if (characteristic.properties.rawValue &
```

```
CBCharacteristicProperties.notify.rawValue) != 0 {  
print("characteristic is notifiable")  
return true  
}  
return false
```

```

}

// MARK: CBPeripheralDelegate

/**
Characteristic has been subscribed to or unsubscribed from */

func peripheral(
    _ peripheral: CBPeripheral,
    didUpdateNotificationStateFor characteristic: CBCharacteristic, error: Error?)

{
    connectedPeripheral = peripheral
    connectedPeripheral.delegate = self
    print("Notification state updated for: " + \

    "\(characteristic.uuid.uuidString)") print("New state: \
(characteristic.isNotifying)") if let errorValue = error {

    print("error subscribing to notification: ") print(errorValue.localizedDescription)
    }
    }

/**
Value was written to the Characteristic */

func peripheral(
    _ peripheral: CBPeripheral,
    didWriteValueFor descriptor: CBDescriptor, error: Error?)

{
    print("data written")
    }

/**
Value downloaded from Characteristic on connected Peripheral */

func peripheral(
    _ peripheral: CBPeripheral,
    didUpdateValueFor characteristic: CBCharacteristic, error: Error?)

```

```

{
print("characteristic updated")
if let value = characteristic.value {

print(value.debugDescription)
print(value.description)
let byteArray = [UInt8](value)
if let stringValue = String(data: value, encoding: .ascii) {

print(stringValue)
packetOffset += characteristicLength
print("new packet offset: \(packetOffset)")
print("new packet offset: \(packetOffset)")
if packetOffset < outboundByteArray.count {

print("sending new packet: " + \

"\(packetOffset)-\(byteArray.count)") writePartialValue(
value: outboundByteArray,
offset: packetOffset)

}
if delegate != nil {
delegate.echoServer(messageReceived: stringValue) }
}
}
}

/**
Services were discovered on the connected Peripheral */

func peripheral(
_peripheral: CBPeripheral,
didDiscoverServices error: Error?)

{
print("services discovered")
connectedPeripheral = peripheral
connectedPeripheral.delegate = self
if error != nil {

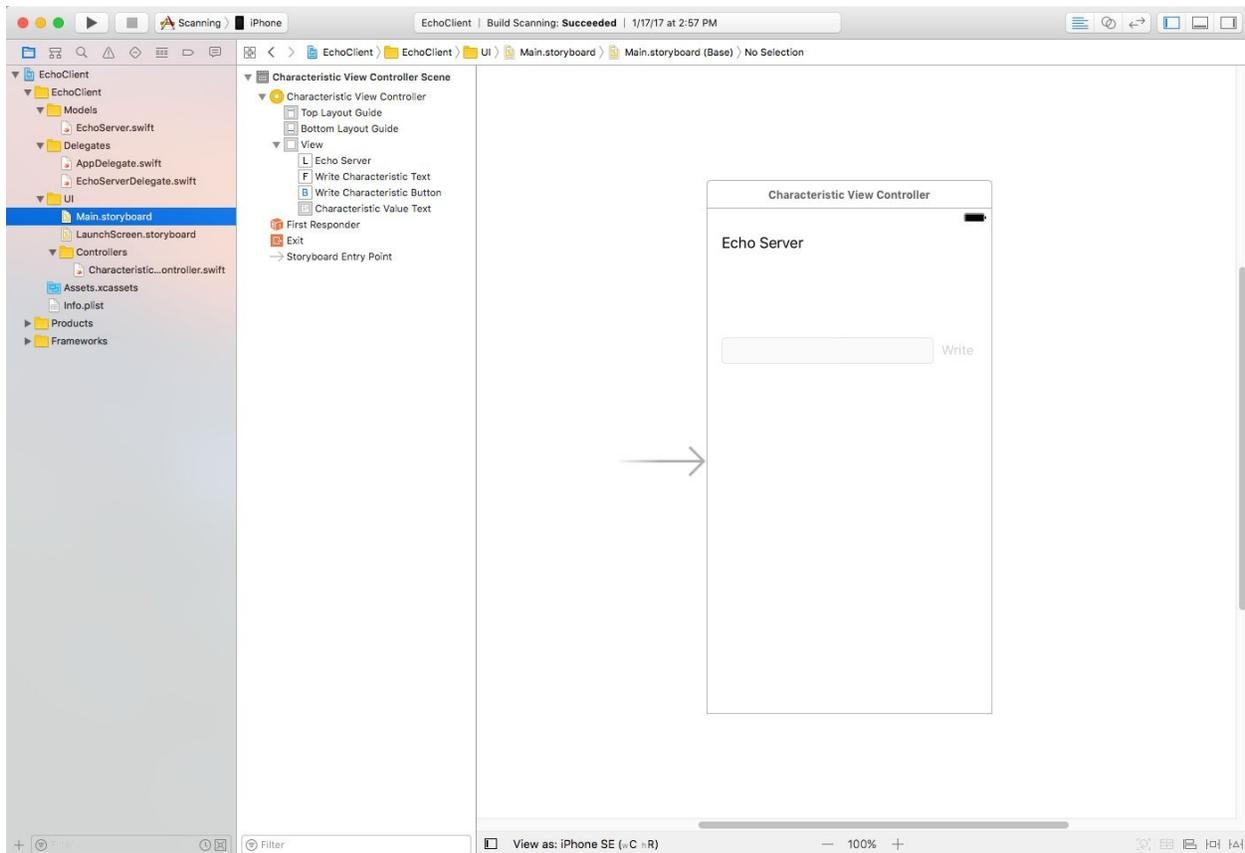
```

```

print("Discover service Error: \(error)") } else {
print("Discovered Service")
for service in peripheral.services! {
if service.uuid == EchoServer.serviceUuid {
connectedPeripheral.discoverCharacteristics(
[EchoServer.readCharacteristicUuid, EchoServer.writeCharacteristicUuid], for:
service)
}
}
print(peripheral.services!)
print("DONE")
}
/**
Characteristics were discovered
for a Service on the connected Peripheral
*/
func peripheral(
_peripheral: CBPeripheral,
didDiscoverCharacteristicsFor service: CBService,
error: Error?)
{
print("characteristics discovered")
connectedPeripheral = peripheral
connectedPeripheral.delegate = self
// grab the service
let serviceIdentifier = service.uuid.uuidString
print("service: \(serviceIdentifier)")
if let characteristics = service.characteristics { print("characteristics found: \(
characteristics.count)") for characteristic in characteristics {
print("-> \(characteristic.uuid.uuidString)") if characteristic.uuid.uuidString == \(
EchoServer.writeCharacteristicUuid.uuidString) {
print("matching uuid found for characteristic") writeCharacteristic =
characteristic
} else if characteristic.uuid.uuidString == \(
EchoServer.readCharacteristicUuid.uuidString) {
readCharacteristic = characteristic if EchoServer.isCharacteristic( isNotifiable:
characteristic) {
connectedPeripheral.setNotifyValue( true,
for: characteristic)
}
}
}
}

```





**Figure 12-4. Project Storyboard Controllers**

The App will indicate a connection status prior to connecting to the Peripheral. After connection, UIButtons and UITextFields are displayed that enable interaction with the Peripheral.

**Example 12-3. UI/Controllers/CharacteristicViewController.swift**

```
import UIKit
import CoreBluetooth
class CharacteristicViewController: UIViewController, \
CBCentralManagerDelegate, EchoServerDelegate {

// MARK: UI Elements
@IBOutlet weak var characteristicValueText: UITextView! @IBOutlet weak var
writeCharacteristicButton: UIButton! @IBOutlet weak var
writeCharacteristicText: UITextField!

// MARK: Bluetooth stuff
```

```

// Bluetooth features
var centralManager:CBCentralManager! // the EchoServer
var echoServer:EchoServer!

/**
View loaded
*/

override func viewDidLoad() {
super.viewDidLoad()
centralManager = CBCentralManager(delegate: self, queue: nil)

}
/**
Write button pressed
*/
@IBAction func onWriteCharacteristicButtonTouchUp(_ sender: UIButton) {
print("write button pressed")
if let string = writeCharacteristicText.text {

print(string)
echoServer.writeValue(value: string)
writeCharacteristicText.text = ""

}
}
// MARK: EchoServerDelegate

/**
Message received from EchoServer. Update UI
*/

func echoServer(messageReceived stringValue: String) {
characteristicValueText.insertText(stringValue)
let stringLength = characteristicValueText.text.characters.count
characteristicValueText.scrollRangeToVisible(

NSMakeRange(stringLength-1, 0))
}

```

```

/**
Characteristic was connected on the EchoServer. Update UI */

func echoServer(
connectedToCharacteristics characteristics: [CBCharacteristic])
{
for characteristic in characteristics {
print(" characteristic: -> " + \
"\(characteristic.uuid.uuidString): " +
"\(characteristic.properties.rawValue)")
if EchoServer.isCharacteristic(isWriteable: characteristic) {
writeCharacteristicText.isEnabled = true
writeCharacteristicButton.isEnabled = true
}
}
}

// MARK: CBCentralManagerDelegate
/**
centralManager is called each time a new Peripheral is discovered

- parameters
- central: the CentralManager for this UIView
- peripheral: A discovered Peripheral
- advertisementData: Bluetooth advertisement discovered with Peripheral
- rssi: the radio signal strength indicator for this Peripheral

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

{
print("Discovered \(peripheral.identifier.uuidString) " +

"\(peripheral.name)")
echoServer = EchoServer(delegate: self, peripheral: peripheral) // find the
advertised name

```

```

if let advertisedName = EchoServer.getNameFromAdvertisementData(
advertisementData: advertisementData)

{
if advertisedName == EchoServer.advertisedName { print("connecting to
peripheral...") centralManager.connect(peripheral, options: nil)

}
}
}
/**

```

Peripheral connected.

- Parameters:

- central: the reference to the central
- peripheral: the connected Peripheral \*/

```

func centralManager(
_ central: CBCentralManager, didConnect peripheral: CBPeripheral)

{
print("Connected Peripheral: \(peripheral.name)") // Do any additional setup
after loading the view. echoServer.connected(peripheral: peripheral)

}
/**

```

Peripheral disconnected

- Parameters:

- central: the reference to the central
- peripheral: the connected Peripheral

\*/

```

func centralManager(
_ central: CBCentralManager,
didDisconnectPeripheral peripheral: CBPeripheral, error: Error?)

{
// disconnected. Leave

```

```
print("disconnected")
writeCharacteristicButton.isEnabled = false
```

```
}
/**
```

Bluetooth radio state changed

- Parameters:

- central: the reference to the central \*/

```
func centralManagerDidUpdateState(_ central: CBCentralManager) {
print("Central Manager updated: checking state") switch (central.state) {
case .poweredOn:
```

```
print("bluetooth on")
central.scanForPeripherals(
withServices: [EchoServer.serviceUuid], options: nil)
```

default:

```
print("bluetooth unavailable")
}
}
}
```

The resulting Central App can scan for and connect to a Bluetooth Low Energy Peripheral. Once connected, the Central can send and receive messages to the Peripheral.

If a user types “hello” into the bottom TextView and hit the Subscribe button, the word “hello” will appear above in the upper Text View (Figure 12-5).

●●●○○ Fido 3G

21:07

✪ 100%  ⚡

# Echo Server

hello

Write



**Figure 12-5. App screen showing interface to read from and write to the Echo Server**

# Programming the Peripheral

The Echo Server will handle incoming writes on one Characteristic (0x2a57) and echo back messages on another Characteristic (0x2a56).

It will also host a minimal GATT Profile that includes a battery percentage, device name, model number, and serial number.

## Advertising and GATT Profile

The Peripheral must advertise and host a GATT Profile, which will include a minimal GATT profile.

The Peripheral will host a read-only, notifiable Characteristic on UUID 0x2a56 and a write-only Characteristic on UUID 0x2a57:

```
// service UUIDs
```

```
let serviceUuid = CBUUID(string: "0000180c-0000-1000-8000-00805f9b34fb")
```

```
// Characteristic UUIDs
```

```
let readCharacteristicUuid = CBUUID(  
string: "00002a56-0000-1000-8000-00805f9b34fb")
```

```
let writeCharacteristicUuid = CBUUID(  
string: "00002a57-0000-1000-8000-00805f9b34fb")
```

```
// Read Characteristic
```

```
var readCharacteristic:CBMutableCharacteristic!
```

```
// Write Characteristic
```

```
var writeCharacteristic:CBMutableCharacteristic!
```

```
let service = CBMutableService(type: serviceUuid, primary: true) var rProperties  
= CBCharacteristicProperties.read
```

```
rProperties.formUnion(CBCharacteristicProperties.notify) var rPermissions =  
CBAttributePermissions.writeable
```

```
rPermissions.formUnion(CBAttributePermissions.readable)
```

```
readCharacteristic = CBMutableCharacteristic( type: readCharacteristicUuid,  
properties: rProperties,  
value: nil,  
permissions: rPermissions)
```

```
let wProperties = CBCharacteristicProperties.write let wPermissions =
```

```

CBAAttributePermissions.writeable writeCharacteristic =
CBMutableCharacteristic(

type: writeCharacteristicUuid,
properties: wProperties,
value: nil,
permissions: wPermissions)

// add Characteristics to the Service
service.characteristics = [ readCharacteristic, writeCharacteristic ] // add Service
to Peripheral
peripheralManager.add(service)

It will advertise as "EchoServer" to be discoverable by the corresponding
Central:
// Advertized name
let advertisingName = "EchoServer"

// Build Advertisement Data
let serviceUuids = [serviceUuid]
let advertisementData:[String: Any] = [

CBAAdvertisementDataLocalNameKey: advertisingName,
CBAAdvertisementDataServiceUUIDsKey: serviceUuids ]
// Start advertising
peripheralManager.startAdvertising(advertisementData)

```

## Handling Reads, Writes, and Subscriptions

When a Central subscribes to the read Characteristic (0x2a56), the peripheralManager didSubscribeTo callback is triggered. Here a reference to the connected Central is stored for later use:

```

/**
Connected Central subscribed to a Characteristic */

func peripheralManager(
_peripheral: CBPeripheralManager,
central: CBCentral,

```

```
didSubscribeTo characteristic: CBCharacteristic)
```

```
{  
self.central = central  
}
```

When a Central unsubscribes, the reference is removed:

```
/**  
Connected Central unsubscribed from a Characteristic */
```

```
func peripheralManager(  
_ peripheral: CBPeripheralManager,  
central: CBCentral,  
didUnsubscribeFrom characteristic: CBCharacteristic)
```

```
{  
self.central = central  
}
```

Once Central initiates a connection, it can subscribe to the read Characteristic, triggering the peripheralManager didReceiveRead callback. It can also send write requests to the write Characteristic, triggering the peripheralManager didReceiveWrite callback.

Write requests are handled by setting the read Characteristic (0x2a56) to the newly written value, thereby notifying the subscribed Centrals.

```
/**  
Connected Central requested to write to a Characteristic */
```

```
func peripheralManager(  
_ peripheral: CBPeripheralManager,  
didReceiveWrite requests: [CBATTRequest])
```

```
{  
for request in requests {  
peripheral.respond(to: request, withResult: CBALError.success) print("new  
request")  
if let value = request.value {
```

```

// update readCharacteristic
print("updating characteristic")
readCharacteristic.value = value
// notify subscribers
print("notifying characteristic")
peripheralManager.updateValue(

value,
for: readCharacteristic,
onSubscribedCentrals: [central])

}
}
}

```

Read requests are handled by responding with a status message regarding the outcome of the read request and the value of the Characteristic if the read request was successful:

```

/**
Connected Central requested to read from a Characteristic */

func peripheralManager(
_peripheral: CBPeripheralManager, didReceiveRead request: CBATTRequest)

{
let characteristic = request.characteristic
if (characteristic.uuid == readCharacteristic.uuid) {

if let value = characteristic.value {
if request.offset > value.count {
peripheralManager.respond(
to: request,
withResult: CBATTError.invalidOffset)
return

}
let range = Range(uncheckedBounds: (
lower: request.offset,
upper: value.count - request.offset))

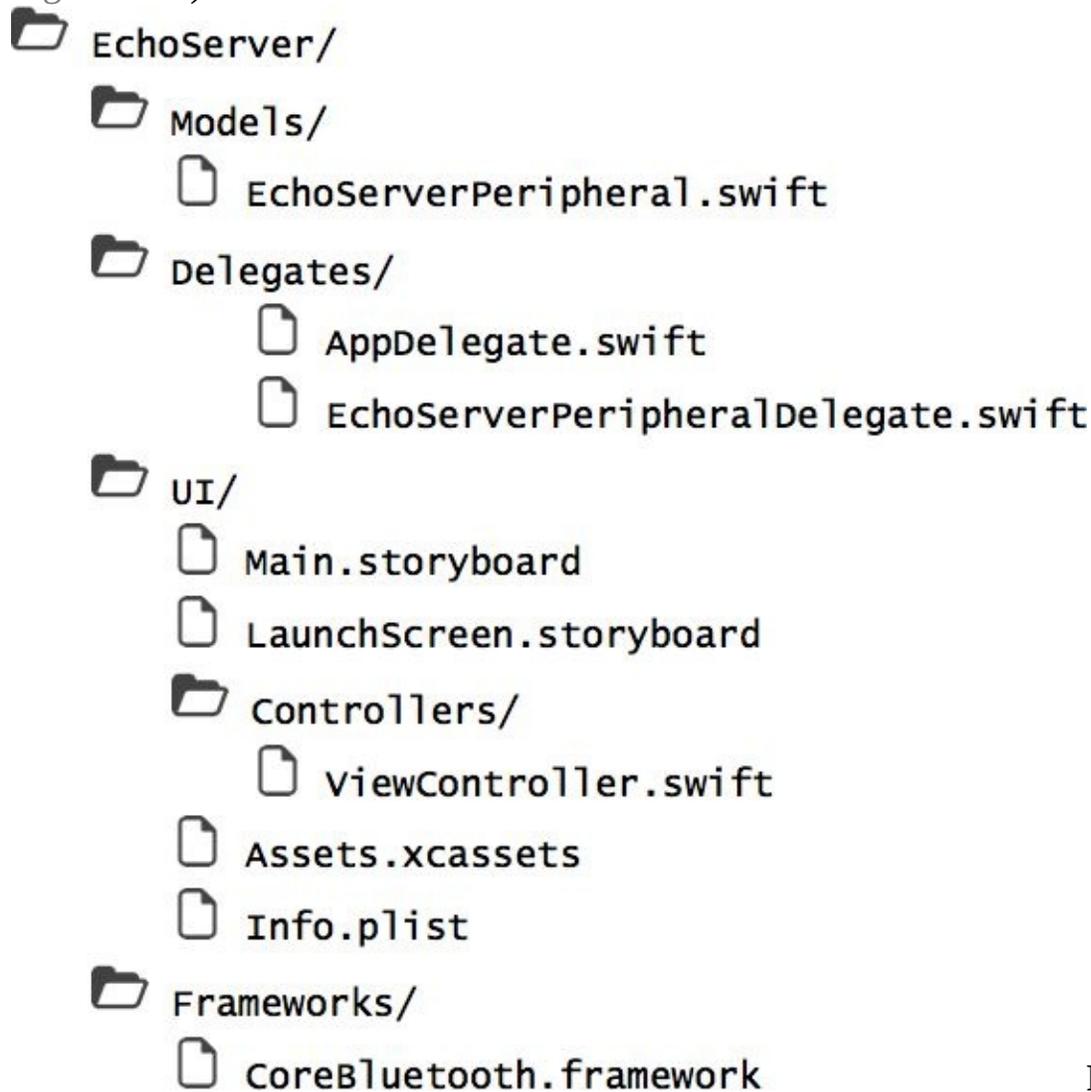
```

```
request.value = value.subdata(in: range)
peripheral.respond(to: request, withResult: CBATTErrorsuccess) }
}
}
```

## Putting It All Together

The following code will create an App that advertises a Peripheral, allows a Central to connect and write values to a writeable Characteristic, and will respond by copying that value to a readable, notifiable Characteristic.

Create a new project called EchoServer and create the following file structure (Figure 12-6):



12-6. Project Structure

Figure

## Frameworks

Import the CoreBluetooth Framework (Figure 12-7).

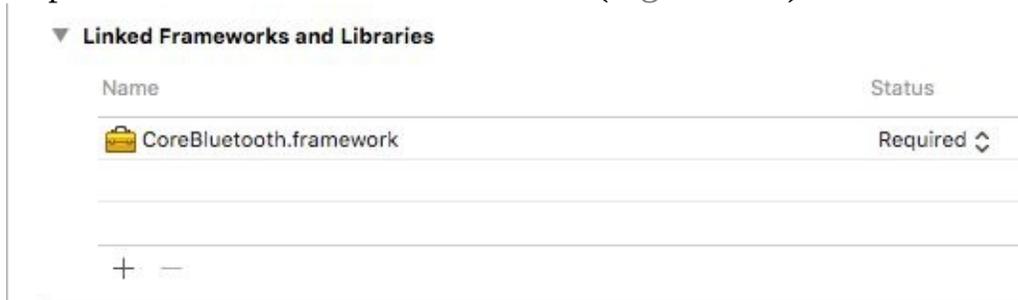


Figure 3-7.

### CoreBluetooth Framework linked into project

Import the CoreBluetooth APIs in the code header:

```
import CoreBluetooth
```

## Models

The EchoServer class will define the GATT Profile of the Echo Server, and will react to subscriptions, reads, and write events. Write events will set read Characteristic to the newly written Characteristic value and notify the connected Central of the change.

### Example 12-4. Models/EchoServerPeripheral.swift

```
import UIKit
import CoreBluetooth
class EchoServerPeripheral : NSObject, CBPeripheralManagerDelegate {
// MARK: Peripheral properties
// Advertized name
let advertisingName = "EchoServer"
// Device identifier
let peripheralIdentifier = "8f68d89b-448c-4b14-aa9a-f8de6d8a4753"
// MARK: GATT Profile
// Service UUID
let serviceUuid = CBUUID(string: "0000180c-0000-1000-8000-00805f9b34fb")
// Characteristic UUIDs
let readCharacteristicUuid = CBUUID(
string: "00002a56-0000-1000-8000-00805f9b34fb") let writeCharacteristicUuid
= CBUUID(
string: "00002a57-0000-1000-8000-00805f9b34fb")
// Read Characteristic
var readCharacteristic:CBMutableCharacteristic!
// Write Characteristic
```

```

var writeCharacteristic:CBMutableCharacteristic!

// the size of a Characteristic let readCharacteristicLength = 20 let
writeCharacteristicLength = 20

// MARK: Peripheral State
// Peripheral Manager
var peripheralManager:CBPeripheralManager!
// Connected Central var central:CBCentral!
// delegate
var delegate:EchoServerPeripheralDelegate!
/**
Initialize BlePeripheral with a corresponding Peripheral

- Parameters:
- delegate: The BlePeripheralDelegate
- peripheral: The discovered Peripheral

*/
init(delegate: EchoServerPeripheralDelegate?) { super.init()
// empty dispatch queue let dispatchQueue:DispatchQueue! = nil

// Build Advertising options
let options:[String : Any] = [
//
CBPeripheralManagerOptionShowPowerAlertKey: true,
// Peripheral unique identifier
fier CBPeripheralManagerOptionRestoreIdentifierKey: peripheralIdenti

]
peripheralManager = CBPeripheralManager(
delegate: self,
queue: dispatchQueue,
options: options)
self.delegate = delegate
}

/**
Stop advertising, shut down the Peripheral */

```

```
func stop() {
  peripheralManager.stopAdvertising()
}
```

```
/**
  Start Bluetooth Advertising.
  This must be after building the GATT profile
  */
```

```
func startAdvertising() {
  let serviceUuids = [serviceUuid]
  let advertisementData:[String: Any] = [
    CBAdvertisementDataLocalNameKey: advertisingName,
    CBAdvertisementDataServiceUUIDsKey: serviceUuids ]
  peripheralManager.startAdvertising(advertisementData) }
/**
```

```
Build Gatt Profile.
This must be done after Bluetooth Radio has turned on
*/
```

```
func buildGattProfile() {
  let service = CBMutableService(type: serviceUuid, primary: true)

  var rProperties = CBCharacteristicProperties.read
  rProperties.formUnion(CBCharacteristicProperties.notify) var rPermissions =
  CBAttributePermissions.writable
  rPermissions.formUnion(CBAttributePermissions.readable) readCharacteristic =
  CBMutableCharacteristic(
```

```
type: readCharacteristicUuid,
properties: rProperties,
value: nil,
permissions: rPermissions)
```

```
let wProperties = CBCharacteristicProperties.write let wPermissions =
CBAttributePermissions.writable writeCharacteristic =
CBMutableCharacteristic(
```

```
type: writeCharacteristicUuid,
properties: wProperties,
```

```

value: nil,
permissions: wPermissions)

service.characteristics = [ readCharacteristic, writeCharacteristic
]
peripheralManager.add(service) }
// MARK: CBPeripheralManagerDelegate

/** Peripheral will become active
*/

func peripheralManager(
 _ peripheral: CBPeripheralManager, willRestoreState dict: [String : Any])

{
print("restoring peripheral state")
}

/**
Peripheral added a new Service
*/

func peripheralManager(
 _ peripheral: CBPeripheralManager, didAdd service: CBService,
error: Error?)

{
print("added service to peripheral") if error != nil {

print(error.debugDescription) }
}

/**
Peripheral started advertising
*/

func peripheralManagerDidStartAdvertising( _ peripheral:
CBPeripheralManager, error: Error?)

```

```

{
if error != nil {
print ("Error advertising peripheral") print(error.debugDescription)

}
self.peripheralManager = peripheral delegate?.echoServerPeripheral?
(startedAdvertising: error) }

/**
Connected Central requested to read from a Characteristic */

func peripheralManager(
_peripheral: CBPeripheralManager,
didReceiveRead request: CBATTRequest)

{
let characteristic = request.characteristic if (characteristic.uuid ==
readCharacteristic.uuid) {

if let value = characteristic.value {
if request.offset > value.count {
peripheralManager.respond(
to: request,
withResult: CBATTErrror.invalidOffset) return

}
let range = Range(uncheckedBounds: ( lower: request.offset,
upper: value.count - request.offset)) request.value = value.subdata(in: range)

peripheral.respond(
to: request,
withResult: CBATTErrror.success)

}
}
}

/**
Connected Central requested to write to a Characteristic */

```

```

func peripheralManager(
  _ peripheral: CBPeripheralManager,
  didReceiveWrite requests: [CBATTRequest])

{
  for request in requests {
    peripheral.respond(to: request, withResult: CBALError.success) print("new
    request")
    if let value = request.value {

      print("notifying delegate")
      delegate?.echoServerPeripheral?(
        valueWritten: value,
        toCharacteristic: request.characteristic)

      // update readCharacteristic print("updating characteristic")
      readCharacteristic.value = value // notify subscribers
      print("notifying characteristic") peripheralManager.updateValue(

        value,
        for: readCharacteristic, onSubscribedCentrals: nil)

    }
  }
}

/**
  Connected Central subscribed to a Characteristic */

func peripheralManager(
  _ peripheral: CBPeripheralManager,
  central: CBCentral,
  didSubscribeTo characteristic: CBCharacteristic)

{
  self.central = central
}

/** Connected Central unsubscribed from a Characteristic */

```

```

func peripheralManager(
    _ peripheral: CBPeripheralManager,
    central: CBCentral,
    didUnsubscribeFrom characteristic: CBCharacteristic)

{
    self.central = central
}

/**
Peripheral is about to notify subscribers
of changes to a Characteristic
*/

func peripheralManagerIsReady(
toUpdateSubscribers peripheral: CBPeripheralManager)
{
    print("Peripheral about to update subscribers")
}

/**
Bluetooth Radio state changed
*/

func peripheralManagerDidUpdateState( _ peripheral: CBPeripheralManager)
{
    peripheralManager = peripheral
    switch peripheral.state {
    case CBManagerState.poweredOn:
        buildGattProfile()
        startAdvertising()
    default: break
    }
    delegate?.echoServerPeripheral?(stateChanged: peripheral.state)
} }

```

## Delegates

The BlePeripheralDelegate relays state changes and events from the Echo

Server, such as when the Echo Server begins Advertising and when data has been written:

### **Example 12-5. Delegates/EchoServerPeripheralDelegate.swift**

```
import UIKit
import CoreBluetooth
@objc protocol EchoServerPeripheralDelegate : class {
/**
Echo Server State Changed

- Parameters:
- rssi: the RSSI
- blePeripheral: the BlePeripheral

*/
@objc optional func echoServerPeripheral( stateChanged state:
CBManagerState)
/**
Echo Server started advertising

- Parameters:
- error: the error message, if any */

@objc optional func echoServerPeripheral( startedAdvertising error: Error?)
/**

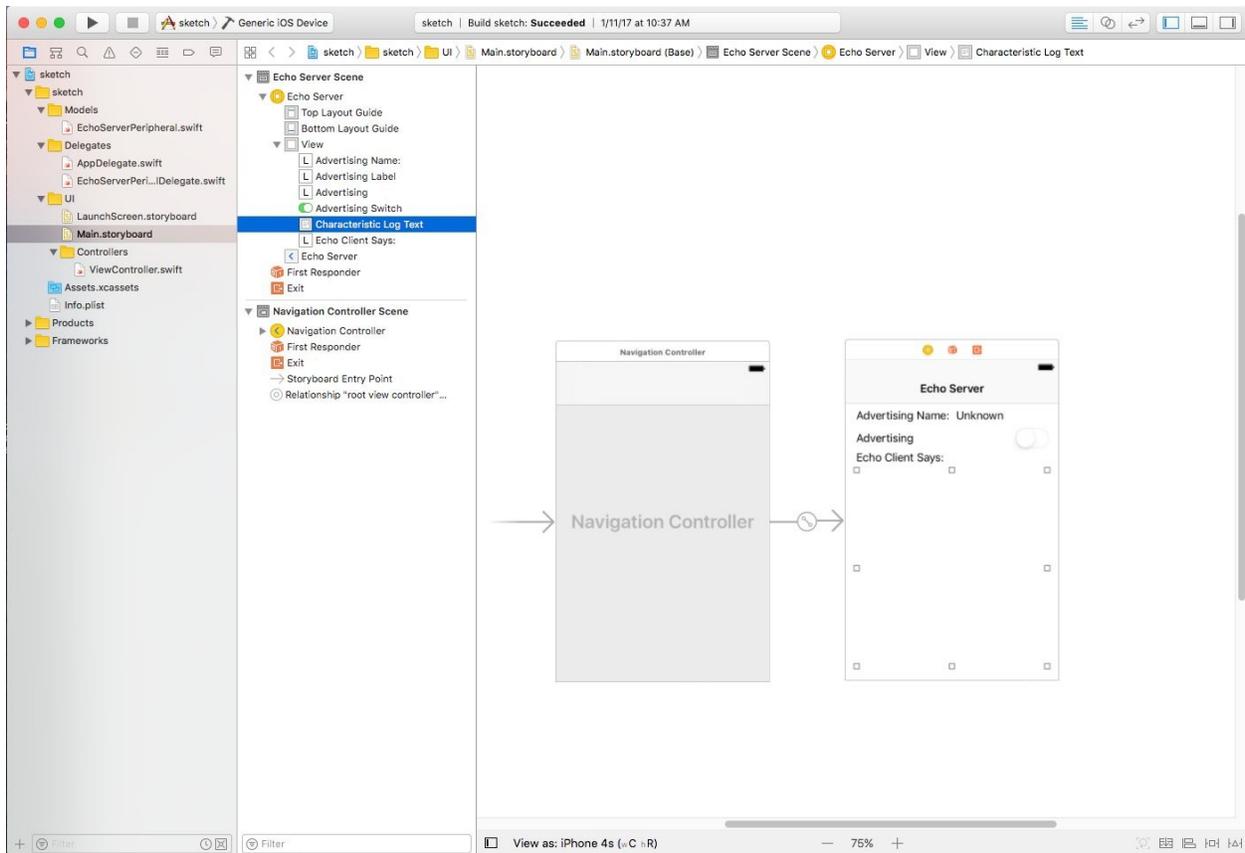
Value written to Characteristic
- Parameters:
- value: the Data value written to the Charactersitic
- characteristic: the Characteristic that was written to */

@objc optional func echoServerPeripheral(
valueWritten value: Data,
toCharacteristic: CBCharacteristic)

}
```

### **Storyboard**

Create the UISwitches, UITextViews, and UILabels in the UIView in the Main.storyboard to create the App's user interface (Figure 12-8).



**Figure 12-8. Project Storyboard Controllers**

The View will display a log of the incoming text written to the EchoServer's write Characteristic (0x2a57) as well as a switch indicating the Advertising state of the EchoServer (Figure 12-9).

**Example 12-6. UI/Controllers/ViewController.swift**

```
import UIKit
import CoreBluetooth
class ViewController: UIViewController, EchoServerPeripheralDelegate {

// MARK: UI Elements
@IBOutlet weak var advertisingLabel: UILabel! @IBOutlet weak var
advertisingSwitch: UISwitch! @IBOutlet weak var characteristicLogText:
UITextView!

// MARK: BlePeripheral
// BlePeripheral
var echoServer:EchoServerPeripheral!
```

```

/**
UIView loaded
*/

override func viewDidLoad() { super.viewDidLoad()
}

/**
View appeared. Start the Peripheral
*/

override func viewWillAppear(_ animated: Bool) { echoServer =
EchoServerPeripheral(delegate: self) advertisingLabel.text =
echoServer.advertisingName }

/**
View will appear. Stop transmitting random data */

override func viewWillDisappear(_ animated: Bool) { echoServer.stop()
}

/**
View disappeared. Stop advertising
*/

override func viewDidDisappear(_ animated: Bool) {
advertisingSwitch.setOn(false, animated: true)
}

// MARK: BlePeripheralDelegate
/**
Echo Server state changed

- Parameters:
- state: the CBManagerState representing the new state */

func echoServerPeripheral(stateChanged state: CBManagerState) { switch
(state) {
case CBManagerState.poweredOn:

```

```

print("Bluetooth on")
case CBManagerState.poweredOff:
print("Bluetooth off")
default:
print("Bluetooth not ready yet...")
}
}
/**
EchoServerPeripheral started advertising

```

```

- Parameters:
- error: the error message, if any
*/

```

```

func echoServerPeripheral(startedAdvertising error: Error?) { if error != nil {
print("Problem starting advertising: " + error.debugDescription)

} else {
print("advertising started")
advertisingSwitch.setOn(true, animated: true)

}
}
/**

```

Value written to Characteristic

```

- Parameters:
- stringValue: the value read from the Characteristic
- characteristic: the Characteristic that was written to

```

```

*/
func echoServerPeripheral(
valueWritten value: Data,
toCharacteristic: CBCharacteristic)

{
print("converting data to String")
let stringValue = String(data: value, encoding: .utf8) if let stringValue =
stringValue {

```

```
print("writing to textview")
characteristicLogText.text = characteristicLogText.text + \ "\n" + stringValue
if !characteristicLogText.text.isEmpty {
characteristicLogText.scrollToVisible( NSMakeRange(0, 1))
} }
}
}
```

The resulting Peripheral App can respond to incoming Characteristic write requests by echoing the value back through a second Characteristic, so that the connected Central can read it back (Figure 12-9).

## Echo Server

Advertising Name: EchoServer

Advertising



Echo Client Says:

hello

**Figure 12-9. App screen showing interface to read from and write to the Echo Server**

## Example code

The code for this chapter is available online  
at: <https://github.com/BluetoothLowEnergyIniOSSwift/Chapter12>

## **Project: Remote Control LED**

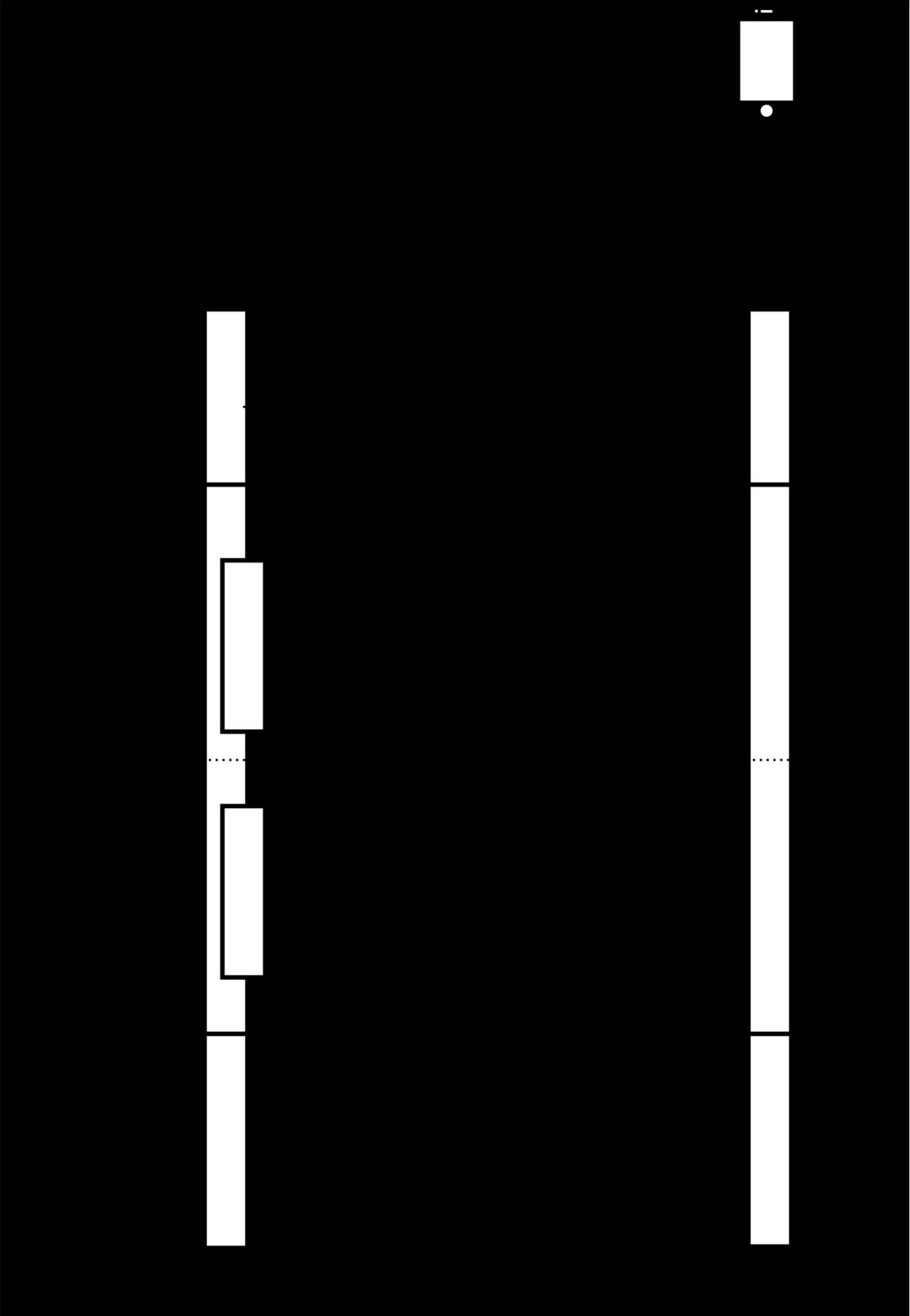
So far, this book has worked a lot with text data rather than binary data, because it's easy to text without using specialized tools such as oscilloscopes or logic analyzers.

Most real-world projects transmit binary instead of text. Binary is much more efficient in transmitting information.

Because binary data is the language of computers, it is easier to work with than text. There is no need to worry about character sets, null characters, or cut-off words.

This project will show how to remotely control an LED on a Peripheral using software on a Central.

The LED Remote works like this (Figure 13-1).



### Figure 13-1. How a Remote Control LED works

In all the other examples, text was being sent between Central and Peripheral. In order for the Central and Peripheral to understand each other, they need shared language between them. In this case, a data packet format.

### Sending Commands to Peripheral

When the Central sends a message, it should be able to specify if it is sending a command or an error. We can do this in two bytes, like this (Figure 13-2).

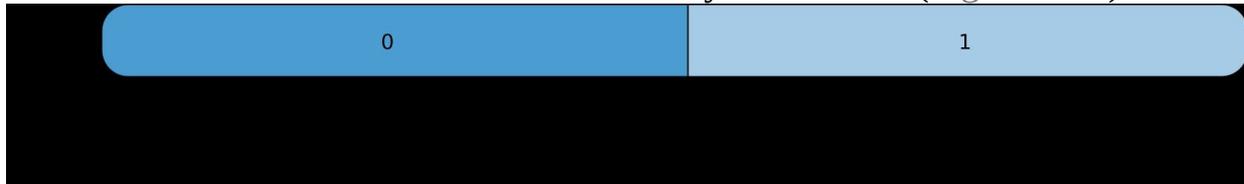


Figure 13-2. Packet structure for commands

The Peripheral reads the footer byte of the incoming message to determine the type of message, i.e., an error or a command. For example, define the message types as:

#### Table 13-1. Footer Values

| Name | Value | Description |
|------|-------|-------------|
|------|-------|-------------|

|                         |   |                                 |
|-------------------------|---|---------------------------------|
| <b>bleResponseError</b> | 0 | The Central is sending an error |
|-------------------------|---|---------------------------------|

|                                |   |                                       |
|--------------------------------|---|---------------------------------------|
| <b>bleResponseConfirmation</b> | 1 | The Central is sending a confirmation |
|--------------------------------|---|---------------------------------------|

|                           |   |                                  |
|---------------------------|---|----------------------------------|
| <b>bleResponseCommand</b> | 2 | The Central is sending a command |
|---------------------------|---|----------------------------------|

The Peripheral reads the first byte to determine the type of error or command.

For example, define the commands as:

#### Table 13-2. Command Values

| Name | Value | Description |
|------|-------|-------------|
|------|-------|-------------|

|                         |   |                               |
|-------------------------|---|-------------------------------|
| <b>bleCommandLedOff</b> | 1 | Turn off the Peripheral's LED |
|-------------------------|---|-------------------------------|

|                        |   |                              |
|------------------------|---|------------------------------|
| <b>bleCommandLedOn</b> | 2 | Turn on the Peripheral's LED |
|------------------------|---|------------------------------|

The Peripheral then responds to the Central with a status message regarding the success or failure to execute the command. This can also be expressed as two bytes (Figure 13-3).

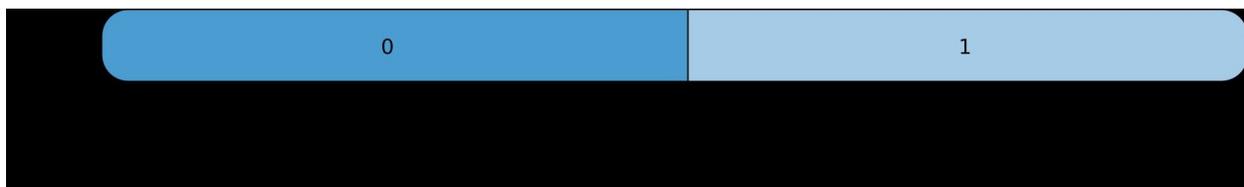


Figure 13-3. Packet structure for responses

If the Peripheral sends a confirmation that the LED state has changed, then the Central inspects the first byte of the message to determine what the current state of the Peripheral's LED is:

### **Table 13-3. Confirmation Values**

| <b>Name</b> | <b>Value</b> | <b>Description</b> |
|-------------|--------------|--------------------|
|-------------|--------------|--------------------|

|                    |   |                             |
|--------------------|---|-----------------------------|
| <b>ledStateOff</b> | 1 | The Peripheral's LED is off |
|--------------------|---|-----------------------------|

|                   |   |                            |
|-------------------|---|----------------------------|
| <b>ledStateOn</b> | 2 | The Peripheral's LED is on |
|-------------------|---|----------------------------|

In this way, a common language is established between the Central and the Peripheral.

## **Gatt Profile**

The Bluetooth Low Energy specification provides a special Service, the Automation IO Service (0x1815), specifically for remote control devices such as this. It is a best practice to use each Characteristic for a single purpose. For this reason, Characteristic 0x2a56 will be used for sending commands to the Peripheral and Characteristic 0x2a57 will be used for responses from the Peripheral:

### **Table 13-4. Characteristic Usages**

| <b>UUID</b> | <b>Use</b> |
|-------------|------------|
|-------------|------------|

|               |                                           |
|---------------|-------------------------------------------|
| <b>0x2a56</b> | Send commands from Central to Peripheral  |
| <b>0x2a57</b> | Send responses from Peripheral to Central |

# Programming the Central

This project shows how to send commands to a Peripheral from a Central.

## GATT Profile

The GATT Profile will be set up as a digital input/output under the Automation IO (0x1815) Service, with commands to and responses from the Peripheral on separate Characteristics.

```
// the Service UUID
static let serviceUuid = CBUUID(string: "1815")
// The Characteristic UUID used to write commands to the Peripheral static let
commandCharacteristicUuid = CBUUID(string: "2A56")
// The Characteristic UUID used to write commands to the Peripheral static let
responseCharacteristicUuid = CBUUID(string: "2A57")
```

## Data Formatting

In order to read and write binary commands to the Peripheral, it and the Central must understand the same messages and formatting.

```
// the size of the characteristic let characteristicLength = 2
```

```
// MARK: Command Data Format
```

```
let bleCommandFooterPosition:Int = 1 let bleCommandDataPosition:Int = 0 let
bleCommandFooter:UInt8 = 1 let bleCommandLedOn:UInt8 = 1 let
bleCommandLedOff:UInt8 = 2
```

```
// MARK: Response Data Format
```

```
let bleResponseFooterPosition:Int = 1 let bleResponseDataPosition:Int = 0 let
bleResponseErrorFooter = 0
let bleResponseConfirmationFooter:UInt8 = 1 static let
bleResponseLedError:UInt8 = 0 static let bleResponseLedOn:UInt8 = 1 static let
bleResponseLedOff:UInt8 = 2
```

## Enable Bluetooth

To turn on the Bluetooth radio programmatically, also enable Bluetooth admin permissions:

let centralManager = CBCentralManager(delegate: self, queue: nil) When the Bluetooth radio turns on, begin scanning:

```
func centralManagerDidUpdateState(_ central: CBCentralManager) {  
  
    switch (central.state) {  
    case .poweredOn:  
        central.scanForPeripherals(withServices: [serviceUuid], options: nil) default:  
    }  
    }  
}
```

Whenever a device is discovered, the centralManager didDiscover callback is triggered by the CBPeripheralManager. If the Advertised name matches the desired RemoteLed name, connect:

```
func getNameFromAdvertisementData(  
    advertisementData: [String : Any]) -> String?  
{  
    // grab the kCBAAdvDataLocalName from the advertisementData // to see if  
    // there's an alternate broadcast name  
    if advertisementData["kCBAAdvDataLocalName"] != nil { return  
        (advertisementData["kCBAAdvDataLocalName"] as! String) }  
    return nil  
}
```

```
func centralManager(  
    _ central: CBCentralManager,  
    didDiscover peripheral: CBPeripheral,  
    advertisementData: [String : Any],  
    rssi RSSI: NSNumber)  
  
    {  
    // find the advertised name  
    if let advertisedName = getNameFromAdvertisementData(  
  
        advertisementData: advertisementData) {  
        if advertisedName == "RemoteLed" {  
            centralManager.connect(peripheral, options: nil)        }  
    }  
}
```

```
}  
}  
}
```

## Connecting

Once a Peripheral is connected, the centralManager didConnect callback is triggered by the CBPeripheralManager. Use this to build a map of the GATT profile, starting with the Services:

```
var connectedPeripheral:CBPeripheral! func centralManager(  
_ central: CBCentralManager,  
didConnect peripheral: CBPeripheral) {  
// store a copy of the connected Peripheral so it isn't tossed  
connectedPeripheral = peripheral;  
connectedPeripheral.delegate = self;  
peripheral.discoverServices(serviceUuid)  
}
```

## Discovering GATT

Once services are discovered, the peripheral didDiscoverServices callback is triggered by the CBPeripheralDelegate. Since the Remote is known to have a Service with UUID 0x1815, search for that service and request discovery of the two known Characteristics, 0x2a56 and 0x2a57:

```
func peripheral(  
_ peripheral: CBPeripheral, didDiscoverServices error: Error?)  
  
{  
// update the local copy of the Peripheral connectedPeripheral = peripheral  
  
if error != nil {  
// error  
} else {  
for service in peripheral.services! {  
if service.uuid == serviceUuid {  
connectedPeripheral.discoverCharacteristics( [readCharacteristicUuid,  
writeCharacteristicUuid],
```

```

for: service)
}
}
}
}

```

A list of Characteristics for each Service will come back in the peripheral `didDiscoverCharacteristicsFor` callback in `CBPeripheralManagerDelegate`. Use this to subscribe to the Read Characteristics and to save references to each Characteristic:

```

func peripheral(
  _ peripheral: CBPeripheral,
  didDiscoverCharacteristicsFor service: CBService, error: Error?)

{
  // update local copy of the peripheral
  connectedPeripheral = peripheral
  connectedPeripheral.delegate = self
  // grab the service
  let serviceIdentifier = service.uuid.uuidString

  if let characteristics = service.characteristics {
    for characteristic in characteristics
    {
      if characteristic.uuid.uuidString == \commandCharacteristicUuid
      {
        commandCharacteristic = characteristic
      }
      else if characteristic.uuid.uuidString == \responseCharacteristicUuid
      {
        responseCharacteristic = characteristic
      }

      connectedPeripheral.setNotifyValue( true,
      for: characteristic)
    }
  }
}
}
}
}

```

## **Sending Commands**

To write the command, build a byte array based on the defined command data structure, and convert to a Data object. Then write the value using the `CBPeripheral.writeValue` method:

```
func writeCommand(ledCommandState: UInt8) {
    if peripheral != nil {
        // build byte array data structure

        var command = [UInt8](repeating: 0, count: characteristicLength)
        command[bleCommandDataPosition] = ledCommandState
        command[bleCommandFooterPosition] = bleCommandFooter

        // convert byte array into Data
        let value = Data(command)
        peripheral.writeValue(value, for: commandCharacteristic)

    }
}
```

## Subscribing to and Unsubscribing from Notifications

If the Characteristic can send notifications, the Central can subscribe to notifications or unsubscribe from them by setting the Notification value for that Characteristic

```
// Subscribe to Notifications on a Characteristic peripheral.setNotifyValue(true,
for: characteristic)
// Unsubscribe from Notification on a Characteristic
peripheral.setNotifyValue(false, for: characteristic)
```

## Receiving Responses

The peripheral `didUpdateValueFor` callback will be triggered by the `CBPeripheralDelegate` when the Peripheral has sent a response. This response can be decoded to determine if the LED turned on or off:

```
func peripheral(
    _ peripheral: CBPeripheral, d
    idUpdateValueFor characteristic: CBCharacteristic, error: Error?)
```

```

{
if let value = characteristic.value { let responseValue = [UInt8](value)

// decode message
let responseType = responseValue[bleResponseFooterPosition] switch
responseType {
case bleResponseConfirmationFooter:
let response = responseValue[bleResponseDataPosition]

if response == bleResponseLedOn { // LED was turned on
} else {
// LED was turned off
}
default:
}
}
}
}

```

## Putting It All Together

The following code will create a single Activity App with a toggle switch that connects to a Peripheral. Once connected the user can flip the toggle back and fourth, which issues a command to the Peripheral to turn an LED on or off. The switch changes state when the App receives confirmation that the

Create a new project called LedRemote.

Create folders and classes, and XML files to reproduce the following structure (Figure 13-4).

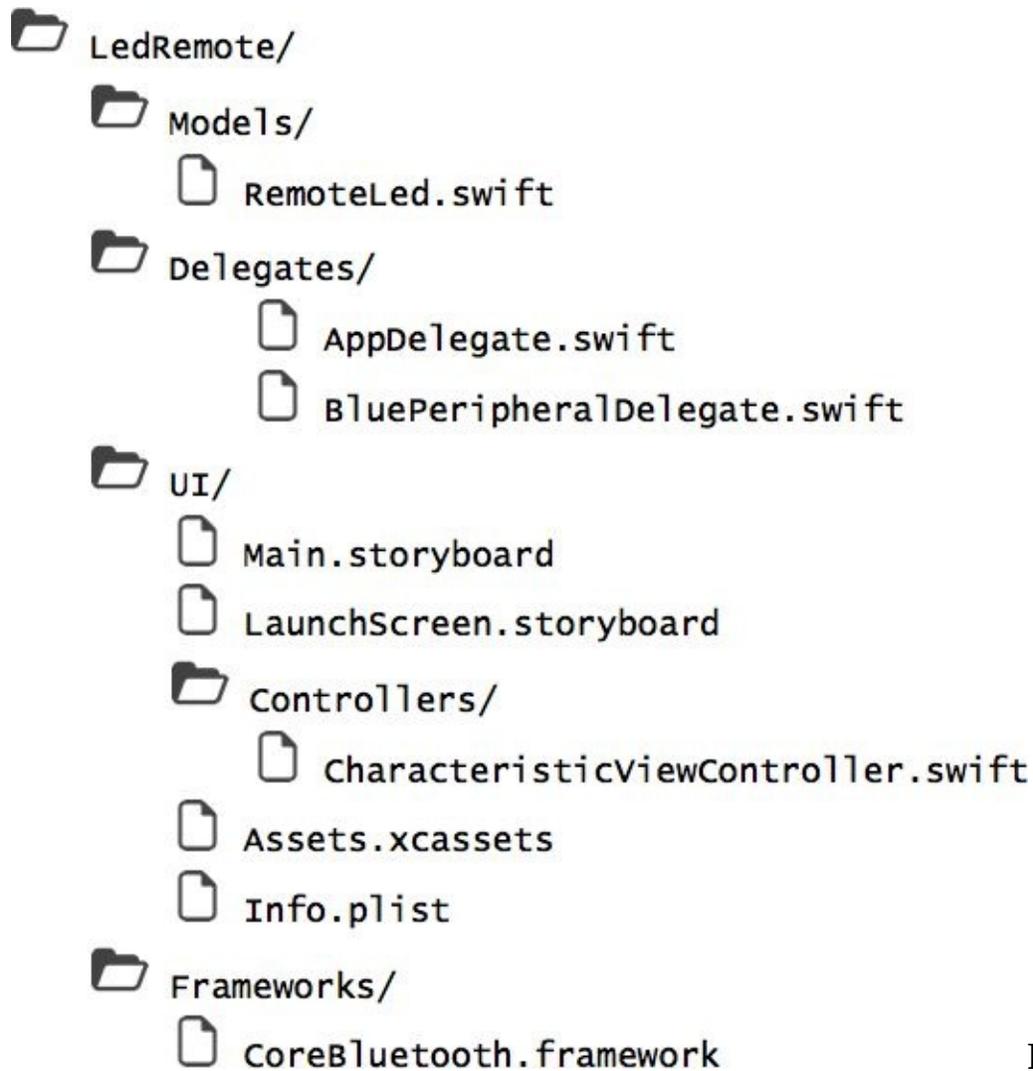


Figure 13-4.

## Project structure

### Frameworks

Import the CoreBluetooth Framework (Figure 13-5).

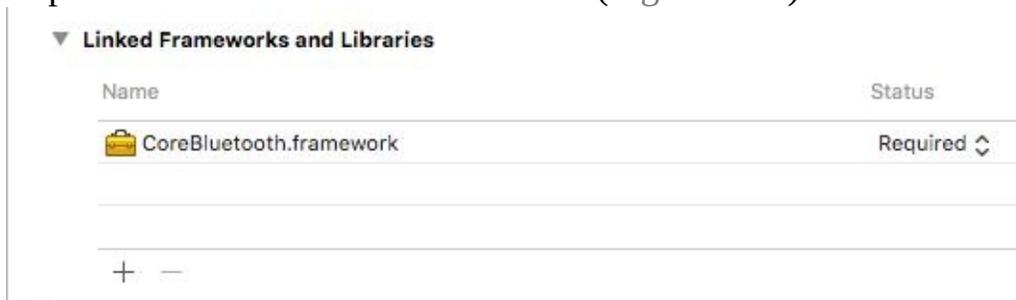


Figure 13-5.

### CoreBluetooth Framework linked into project

Import the CoreBluetooth APIs in the code header:

```
import CoreBluetooth
```

### Objects

The BleCommManager turns the Bluetooth radio on and scans for nearby Peripherals.

### **Example 13-1. Models/RemoteLed.swift**

```
import UIKit
import CoreBluetooth
class RemoteLed:NSObject, CBPeripheralDelegate {
// MARK: Peripheral properties

// The Broadcast name of the Perihperal
static let advertisedName = "LedRemote"
// the Service UUID
static let serviceUuid = CBUUID(string: "1815")
// The Characteristic UUID used to write commands to the Peripheral static let
commandCharacteristicUuid = CBUUID(string: "2A56") // The Characteristic
UUID used to write commands to the Peripheral static let
responseCharacteristicUuid = CBUUID(string: "2A57") // the size of the
characteristic
let characteristicLength = 2

// MARK: Command Data Format

// Footer data position
let bleCommandFooterPosition:Int = 1 // Message data position
let bleCommandDataPosition:Int = 0 // Command
let bleCommandFooter:UInt8 = 1
// Turn the LED on
let bleCommandLedOn:UInt8 = 1
// Turn the LED off
let bleCommandLedOff:UInt8 = 2
// MARK: Response Data Format
// Footer data position
let bleResponseFooterPosition:Int = 1 // Message data position
let bleResponseDataPosition:Int = 0 // Error response
let bleResponseErrorFooter = 0
// Confirmation response
let bleResponseConfirmationFooter:UInt8 = 1 // Error Response
static let bleResponseLedError:UInt8 = 0 // Confirmation response
static let bleResponseLedOn:UInt8 = 1 // Command
static let bleResponseLedOff:UInt8 = 2
```

```

// MARK: connected device

// RemateLedDelegate
var delegate:RemoteLedDelegate!
// connected Peripheral
var peripheral:CBPeripheral!
// connected Characteristic
var commandCharacteristic:CBCharacteristic! var
responseCharacteristic:CBCharacteristic!

/**
Initialize EchoServer with a corresponding Peripheral

- Parameters:
- delegate: The RemoteLEDPeripheral
- peripheral: The discovered Peripheral

*/
init(delegate: RemoteLedDelegate, peripheral: CBPeripheral) { super.init()
self.delegate = delegate
self.peripheral = peripheral
self.peripheral.delegate = self

}

/**
Notify the RemoteLed that the peripheral has been connected */

func connected(peripheral: CBPeripheral) {
self.peripheral = peripheral
self.peripheral.delegate = self
self.peripheral.discoverServices([RemoteLed.serviceUuid])

}

/**
Get a advertised name from an advertisementData packet. This may be different
than the actual Peripheral name */

static func getNameFromAdvertisementData(

```

```

advertisementData: [String : Any]) -> String?
{
// grab thekCBAAdvDataLocalName from the advertisementData // to see if
there's an alternate broadcast name
if advertisementData["kCBAAdvDataLocalName"] != nil { return
(advertisementData["kCBAAdvDataLocalName"] as! String) }
return nil
}

```

```
/**
```

```
Turn the remote LED on */
```

```
func turnLedOn() {
```

```
writeCommand(ledCommandState: bleCommandLedOn) }
```

```
/**
```

```
Turn the remote LED off
```

```
*/
```

```
func turnLedOff() {
```

```
writeCommand(ledCommandState: bleCommandLedOff) }
```

```
/**
```

```
Write a command to the remote
```

```
*/
```

```
func writeCommand(ledCommandState: UInt8) {
```

```
if peripheral != nil {
```

```
var command = [UInt8](repeating: 0, count: characteristicLength)
```

```
command[bleCommandDataPosition] = ledCommandState
```

```
command[bleCommandFooterPosition] = bleCommandFooter let value =
```

```
Data(command)
```

```
print("writing value: \(value)")
```

```
var writeType = CBCharacteristicWriteType.withResponse if
```

```
RemoteLed.isCharacteristic(
```

```
isWritableWithoutResponse: commandCharacteristic) {
```

```
writeType = CBCharacteristicWriteType.withoutResponse }
```

```
peripheral.writeValue(
```

```
value,  
for: commandCharacteristic,  
type: writeType)  
}  
}
```

```
/** Check if Characteristic is readable
```

```
- Parameters:
```

```
- characteristic: The Characteristic to test
```

```
- returns: True if characteristic is readable */
```

```
static func isCharacteristic(  
isReadable characteristic: CBCharacteristic) -> Bool {
```

```
if (characteristic.properties.rawValue & \  
  
CBCharacteristicProperties.read.rawValue) != 0 {  
print("readable")  
return true  
}  
return false  
}
```

```
CBCharacteristicProperties.read.rawValue) != 0 {
```

```
print("readable")
```

```
return true
```

```
}
```

```
return false
```

```
}
```

```
/**
```

```
Check if Characteristic is writeable
```

```
- Parameters:
```

```
- characteristic: The Characteristic to test
```

```
- returns: True if characteristic is writeable */
```

```
static func isCharacteristic(  
isWritable characteristic: CBCharacteristic) -> Bool
```

```
{  
print("testing if characteristic is writeable")  
if (characteristic.properties.rawValue & \  
  
CBCharacteristicProperties.write.rawValue) != 0 ||  
(characteristic.properties.rawValue & \  
CBCharacteristicProperties.writeWithoutResponse.rawValue) != 0  
  
{  
print("characteristic is writeable") return true
```

```
{  
print("testing if characteristic is writeable")
```

```
if (characteristic.properties.rawValue & \  
  
CBCharacteristicProperties.write.rawValue) != 0 ||  
(characteristic.properties.rawValue & \  
CBCharacteristicProperties.writeWithoutResponse.rawValue) != 0
```

```
{  
print("characteristic is writeable") return true
```

```
}  
print("characteristic is not writeable") return false
```

```
}  
/**
```

Check if Characteristic is writeable with response

- Parameters:

- characteristic: The Characteristic to test

- returns: True if characteristic is writeable with response \*/

```
static func isCharacteristic(  
isWritableWithResponse characteristic: CBCharacteristic) -> Bool {  
if (characteristic.properties.rawValue & \
```

```
CBCharacteristicProperties.write.rawValue) != 0 {  
return true }  
return false }
```

```
/**
```

Check if Characteristic is writeable without response

- Parameters:

- characteristic: The Characteristic to test

- returns: True if characteristic is writeable without response \*/

```
static func isCharacteristic(  
isWritableWithoutResponse characteristic: CBCharacteristic) -> Bool {  
if (characteristic.properties.rawValue & \
```

```
CBCharacteristicProperties.writeWithoutResponse.rawValue) != 0 {  
return true }  
return false }
```

```
/**
```

Check if Characteristic is notifiable

- Parameters:

- characteristic: The Characteristic to test

- returns: True if characteristic is notifiable \*/

```
static func isCharacteristic(  
isNotifiable characteristic: CBCharacteristic) -> Bool {  
if (characteristic.properties.rawValue & \
```

```

CBCharacteristicProperties.notify.rawValue) != 0 {
return true }
return false }

// MARK: CBPeripheralDelegate

/**
Characteristic has been subscribed to or unsubscribed from */

func peripheral(
    _ peripheral: CBPeripheral,
    didUpdateNotificationStateFor characteristic: CBCharacteristic, error: Error?)

{
    print("Notification state updated for: " + "\(characteristic.uuid.uuidString)")
    print("New state: \(characteristic.isNotifying)")

    if let errorMessage = error {
        print("error subscribing to notification: ")
        print(errorMessage.localizedDescription as String)
    }
}

/**
Value downloaded from Characteristic on connected Peripheral */

func peripheral(
    _ peripheral: CBPeripheral,
    didUpdateValueFor characteristic: CBCharacteristic, error: Error?)

{
    if let value = characteristic.value { print(value.debugDescription)

    print(value.description as String)
    let responseValue = [UInt8](value)

    // decode message
    let responseType = responseValue[bleResponseFooterPosition] switch
    responseType {

```

```

case bleResponseConfirmationFooter:

print("ble device responded")
let response = responseValue[bleResponseDataPosition]
delegate.remoteLed(confirmationReceived: response) default:
print("ble response unknown") }
}

/**
Services were discovered on the connected Peripheral */

func peripheral(
_ peripheral: CBPeripheral,
didDiscoverServices error: Error?)

{
print("services discovered")

if error != nil {
print("Discover service Error: \(error)")
} else {
print("Discovered Service")
for service in peripheral.services! {
if service.uuid == RemoteLed.serviceUuid {
self.peripheral.discoverCharacteristics( [
RemoteLed.commandCharacteristicUuid,
RemoteLed.responseCharacteristicUuid ],
for: service)
}
}
print(peripheral.services!)
print("DONE")
}
}
/**
Characteristics were discovered
for a Service on the connected Peripheral
*/

```

```

func peripheral(
    _peripheral: CBPeripheral,
    didDiscoverCharacteristicsFor service: CBService, error: Error?)
{
    print("characteristics discovered")

    // grab the service
    let serviceIdentifier = service.uuid.uuidString
    print("service: \(serviceIdentifier)")

    if let characteristics = service.characteristics { print("characteristics found: \(
        characteristics.count)") for characteristic in characteristics {

        if characteristic.uuid == \

        RemoteLed.commandCharacteristicUuid {
            commandCharacteristic = characteristic } else if characteristic.uuid == \
            RemoteLed.responseCharacteristicUuid {
                responseCharacteristic = characteristic
                print(" -> \(characteristic.uuid.uuidString): " + "\
                    (characteristic.properties.rawValue)")
                if RemoteLed.isCharacteristic(
                    isNotifiable: responseCharacteristic)
                {
                    self.peripheral.setNotifyValue( true,
                    for: responseCharacteristic)
                }
            }
        }
        delegate.remoteLed(connectedToCharacteristics: [ responseCharacteristic,
        commandCharacteristic
        ])
    }
}
}
}

```

## Delegates

Create an RemoteLedDelegate that relays important events from the RemoteLed.

### Example 13-2. Delegates/RemoteLedDelegate.swift

```

import UIKit
import CoreBluetooth
protocol RemoteLedDelegate {
/**
Characteristic was connected on the Remote LED

- Parameters:
- characteristic: the connected Characteristic
*/
func remoteLed(
connectedToCharacteristics characteristics: [CBCharacteristic])

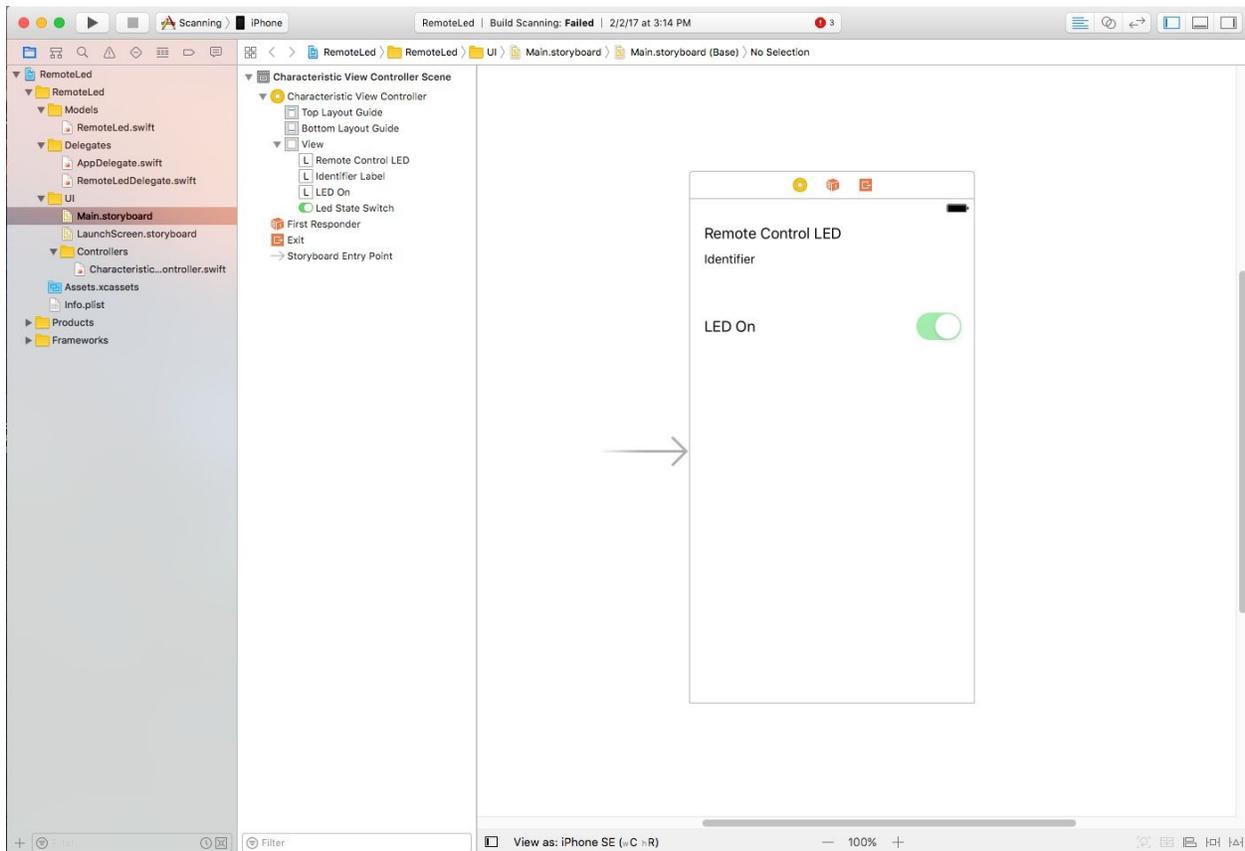
/**
Error received from Remote
- Parameters
- messageValue: an error response */
func remoteLed(errorReceived messageValue: String)
/**
Remote command was successful and a response was issued

- Parameters:
- ledState: one of RemoteLed.ledOn or RemoteLed.ledOff
*/
func remoteLed(confirmationReceived ledState: UInt8) }

```

## Storyboard

Create the UISwitches, UITextViews, and UILabels in the UIView in the Main.storyboard to create the App's user interface (Figure 13-6):



**Figure 13-6. Project Storyboard Controllers**

The Main activity will have a switch and a label describing what the switch does. When the user toggles the switch, the UIViewController will issue a command to the Peripheral to turn its LED on or off.

**Example 13-3. UI/Controllers/CharacteristicViewController.swift**

```
import UIKit
import CoreBluetooth
class CharacteristicViewController: UIViewController, \
    CBCentralManagerDelegate, RemoteLedDelegate {
    // MARK: UI Components
    @IBOutlet weak var identifierLabel: UILabel! @IBOutlet weak var
    ledStateSwitch: UISwitch!

    // MARK: Bluetooth stuff
    // Bluetooth Radio
    var centralManager:CBCentralManager!
```

```

// the remote Peripheral var remoteLed:RemoteLed!

/**
View loaded
*/

override func viewDidLoad() {
super.viewDidLoad()
centralManager = CBCentralManager(delegate: self, queue: nil)
}

/**
LED switch toggled
*/

@IBAction func onLedStateSwitchTouchUp(_ sender: UISwitch) { // prevent
user interaction during update
sender.isEnabled = false
if sender.isOn {

print("led switched on")
remoteLed.turnLedOn()

} else {
print("led switched off")
remoteLed.turnLedOff()

}
}
// MARK: RemoteLedDelegate

/**
Characteristic was connected on the Remote LED. Update UI */

func remoteLed(
connectedToCharacteristics characteristics: [CBCharacteristic])
{
ledStateSwitch.isEnabled = true remoteLed.turnLedOn()
}

```

```

/**
Error received from Remote. Update UI
*/

func remoteLed(errorReceived messageValue: String) { // There was a problem.
flip the switch back ledStateSwitch.isOn = !ledStateSwitch.isOn
ledStateSwitch.isEnabled = true

}

```

```

/**
Remote command was successful and a response was issued. Update UI */

func remoteLed(confirmationReceived ledState: UInt8) { if ledState ==
RemoteLed.bleResponseLedOn {
print("led turned on")
ledStateSwitch.isOn = true

} else {
print("led turned off")
ledStateSwitch.isOn = false

}
ledStateSwitch.isEnabled = true
}

```

```

// MARK: CBCentralManagerDelegate /**
centralManager is called each time a new Peripheral is discovered

```

- parameters
- central: the CentralManager for this UIView
- peripheral: A discovered Peripheral
- advertisementData: Bluetooth advertisement found with Peripheral
- rssi: the radio signal strength indicator for this Peripheral

```

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

```

```

//print("Discovered \(peripheral.name)")
print("Discovered \(peripheral.identifier.uuidString) " + "\(peripheral.name)")
remoteLed = RemoteLed(delegate: self, peripheral: peripheral)
// find the advertised name
if let advertisedName = RemoteLed.getNameFromAdvertisementData(
advertisementData: advertisementData)
{
if advertisedName == RemoteLed.advertisedName { print("connecting to
peripheral...") centralManager.connect(peripheral, options: nil) }
}
}

```

```
/**
```

Peripheral connected.

- Parameters:

- central: the reference to the central
- peripheral: the connected Peripheral \*/

```
func centralManager(
_ central: CBCentralManager, didConnect peripheral: CBPeripheral)
```

```

{
print("Connected Peripheral: \(peripheral.name)")
remoteLed.connected(peripheral: peripheral)
// Do any additional setup after loading the view.
identifierLabel.text = remoteLed.peripheral.identifier.uuidString

```

```
}
```

```
/**
```

Peripheral disconnected

- Parameters:

- central: the reference to the central
- peripheral: the connected Peripheral

```
*/
```

```
func centralManager(
_ central: CBCentralManager,
didDisconnectPeripheral peripheral: CBPeripheral, error: Error?)
```

```

{
// disconnected. Leave print("disconnected")

}
/**
Bluetooth radio state changed

- Parameters:
- central: the reference to the central
*/

func centralManagerDidUpdateState(_ central: CBCentralManager) {
print("Central Manager updated: checking state") switch (central.state) {

case .poweredOn:
print("bluetooth on")
centralManager.scanForPeripherals(

withServices: [RemoteLed.serviceUuid], options: nil)
default:
print("bluetooth unavailable")
}
}
}
}

```

The resulting App scan for and connect to a Bluetooth Low Energy Peripheral. Once connected, the user can turn the Peripheral's LED on or off using the switch. When the LED turns on. The switch fully moves when the App receives confirmation from the Peripheral that the LED state has changed (Figure 13-7).

●●●○○ Fido 3G

21:17

⌘ 100%  ⚡

# Remote Control LED

Identifier

LED On



●●●○○ Fido 3G

21:16

⌘ 100%  ⚡

# Remote Control LED

Identifier

LED On



**Figure 13-7. App screen showing LED switch in on and off states**

# Programming the Peripheral

This project shows how to process commands sent from a Central, and respond with status confirmations.

## Data Formatting

In order to read and write binary commands to the Peripheral, it and the Central must understand the same messages and formatting. To best emulate a typical Bluetooth device, UInt8 data types are used to represent the binary commands. Regular Integers are used for positions:

```
// MARK: Commands
```

```
// Data Positions
```

```
let bleCommandFooterPosition = 1; let bleCommandDataPosition = 0;
```

```
// Command flag
```

```
let bleCommandFooter:UInt8 = 1;
```

```
// LED State
```

```
let bleCommandLedOn:UInt8 = 1; let bleCommandLedOff:UInt8 = 2;
```

```
// MARK: Response
```

```
// Data Positions
```

```
let bleResponseFooterPosition = 1; let bleResponseDataPosition = 0;
```

```
// Response Types
```

```
let bleResponseErrorFooter:UInt8 = 0; let
```

```
bleResponseConfirmationFooter:UInt8 = 1;
```

```
// LED States
```

```
let bleResponseLedError:UInt8 = 0; let bleResponseLedOn:UInt8 = 1; let
```

```
bleResponseLedOff:UInt8 = 2;
```

## Advertising and GATT Profile

The GATT Profile will be set up as a digital input/output under the Automation IO (0x1815) Service, with commands to and responses from the Peripheral on separate Characteristics.

The Peripheral will host a writeable Characteristic on UUID 0x2a56 to receive commands and a read-only, notifiable Characteristic on UUID 0x2a57 to issue responses:

```
// Service UUID
let serviceUuid = CBUUID(string: "00001815-0000-1000-8000-00805f9b34fb")
// Characteristic UUIDs
let commandCharacteristicUuid =
    CBUUID(string: "00002a56-0000-1000-8000-00805f9b34fb") let
    responseCharacteristicUuid =
    CBUUID(string: "00002a57-0000-1000-8000-00805f9b34fb")

// Command Characteristic
var commandCharacteristic:CBMutableCharacteristic!
// Response Characteristic
var responseCharacteristic:CBMutableCharacteristic!
// create Service
let service = CBMutableService(type: serviceUuid, primary: true)

var rProperties = CBCharacteristicProperties.read
rProperties.formUnion(CBCharacteristicProperties.notify) var rPermissions =
    CBAttributePermissions.writeable
rPermissions.formUnion(CBAttributePermissions.readable)
responseCharacteristic = CBMutableCharacteristic(

type: responseCharacteristicUuid,
properties: rProperties,
value: nil,
permissions: rPermissions)

let cProperties = CBCharacteristicProperties.write let cPermissions =
    CBAttributePermissions.writeable commandCharacteristic =
    CBMutableCharacteristic(

type: commandCharacteristicUuid,
properties: cProperties,
```

```
value: nil,  
permissions: cPermissions)  
  
// add Characteristics to Service  
service.characteristics = [ responseCharacteristic, commandCharacteristic ] //  
add Service to Peripheral  
peripheralManager.add(service)
```

It will also advertise as "LedRemote" so as to be discoverable by the corresponding Central.

```
// Advertized name  
let advertisingName = "LedRemote"  
  
// build Advertising Data  
let serviceUuids = [serviceUuid]  
let advertisementData:[String: Any] = [  
  
    CBAvertisementDataLocalNameKey: advertisingName,  
    CBAvertisementDataServiceUUIDsKey: serviceUuids ]  
// start Advertising  
peripheralManager.startAdvertising(advertisementData)
```

## Handling Subscriptions and Writes

Once Central initiates a connection, it can subscribe to the response Characteristic (0x2a57), triggering the peripheralManager didSubscribeTo callback. This is used as an opportunity to store a reference to the connected Central:

```
func peripheralManager(  
    _ peripheral: CBPeripheralManager,  
    central: CBCentral,  
    didSubscribeTo characteristic: CBCharacteristic)  
  
{  
    self.central = central  
}
```

When a Central writes data to the command Characteristic (0x2a56), the

RemoteLed will process the command and change the LED state. It will change the value of the response Characteristic (0x2a56) to match the new LED state and send a notification of that change.

```
func peripheralManager(
  _ peripheral: CBPeripheralManager, didReceiveWrite requests:
  [CBATTRequest])

{
  for request in requests {
    peripheral.respond(to: request, withResult: CBATTError.success) if let value =
    request.value {

      let bleCommandValue = [UInt8](value)
      processCommand(bleCommandValue: bleCommandValue) }
    }
  }
}
```

The command is processed by reading the footer at position 1 to determine what type of message is being sent. If it is a command, position 0 is inspected to determine the LED state:

```
func processCommand(bleCommandValue: [UInt8]) {
  if bleCommandValue[bleCommandFooterPosition] == bleCommandFooter {
    print ("Command found")
    switch (bleCommandValue[bleCommandDataPosition]) {

      case bleCommandLedOn:
        print("Turning LED on") setLedState(ledState: true)

      case bleCommandLedOff:
        print("Turning LED off") setLedState(ledState: false)

      default:
        print("Unknown command value")
    }
  }
}
```

On iOS, it is possible to turn on the camera flash LED, like this:



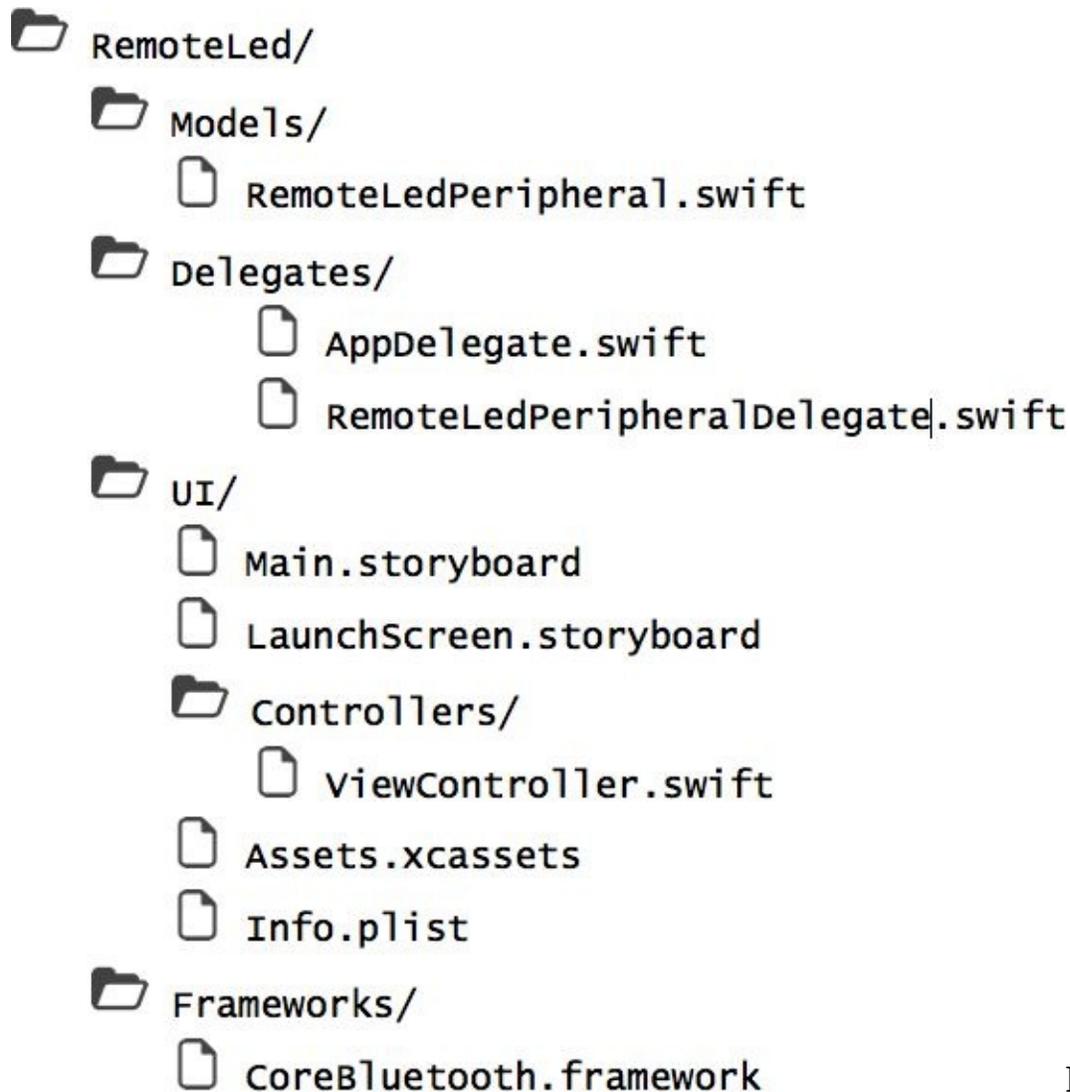


Figure 13-

## 8. Project structure

### Frameworks

Import the CoreBluetooth Framework (Figure 13-9):

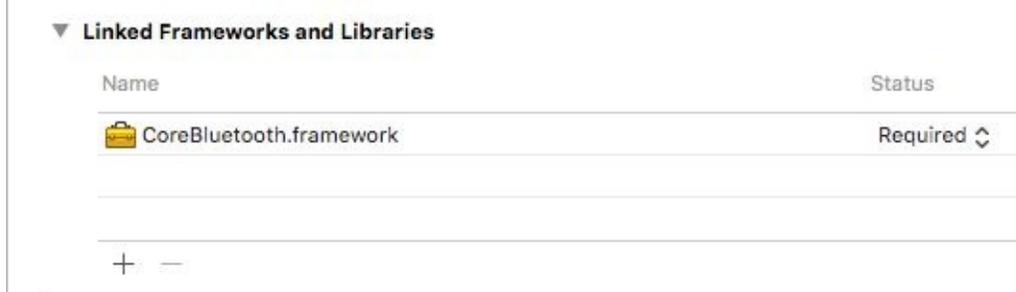


Figure 13-9.

### CoreBluetooth Framework linked into project

Import the CoreBluetooth APIs in the code header:

```
import CoreBluetooth
```

### Models

The RemoteLedPeripheral class will define the GATT Profile for the Peripheral, and will react to read, write, and subscription events:

**Example 13-4. Models/RemoteLedPeripheral.swift**

```
import UIKit
import CoreBluetooth
class RemoteLedPeripheral : NSObject, CBPeripheralManagerDelegate {
// MARK: Peripheral properties

// Advertized name
let advertisingName = "LedRemote"
// Device identifier
let peripheralIdentifier = "8f68d89b-448c-4b14-aa9a-f8de6d8a4753"

// MARK: GATT Profile

// Service UUID
let serviceUuid = CBUUID(string: "00001815-0000-1000-8000-00805f9b34fb")
// Characteristic UUIDs
let commandCharacteristicUuid =

CBUUID(string: "00002a56-0000-1000-8000-00805f9b34fb") let
responseCharacteristicUuid =
CBUUID(string: "00002a57-0000-1000-8000-00805f9b34fb") // Command
Characteristic
var commandCharacteristic:CBMutableCharacteristic!
// Response Characteristic
var responseCharacteristic:CBMutableCharacteristic!
// the size of a Characteristic
let commandCharacteristicLength = 2
let responseCharacteristicLength = 2

// MARK: Commands

// Data Positions
let bleCommandFooterPosition = 1; let bleCommandDataPosition = 0; //
Command flag
let bleCommandFooter:UInt8 = 1; // LED State
let bleCommandLedOn:UInt8 = 1; let bleCommandLedOff:UInt8 = 2;

// MARK: Response
```

```

// Data Positions
let bleResponseFooterPosition = 1; let bleResponseDataPosition = 0; // Response
Types
let bleResponseErrorFooter:UInt8 = 0; let
bleResponseConfirmationFooter:UInt8 = 1; // LED States
let bleResponseLedError:UInt8 = 0;
let bleResponseLedOn:UInt8 = 1;
let bleResponseLedOff:UInt8 = 2

// MARK: Peripheral State

// Peripheral Manager
var peripheralManager:CBPeripheralManager! // Connected Central
var central:CBCentral!
// delegate
var delegate:RemoteLedPeripheralDelegate!

/**
Initialize BlePeripheral with a corresponding Peripheral

- Parameters:
- delegate: The BlePeripheralDelegate
- peripheral: The discovered Peripheral

*/
init(delegate: RemoteLedPeripheralDelegate?) { super.init()
// empty dispatch queue
let dispatchQueue:DispatchQueue! = nil
// Build Advertising options
let options:[String : Any] = [

CBPeripheralManagerOptionShowPowerAlertKey: true, // Peripheral unique
identifier
CBPeripheralManagerOptionRestoreIdentifierKey:

peripheralIdentifier
]
peripheralManager = CBPeripheralManager(

delegate: self,

```

```

queue: dispatchQueue, options: options)

self.delegate = delegate }

/**
Stop advertising, shut down the Peripheral */

func stop() {
peripheralManager.stopAdvertising()
}

/**
Start Bluetooth Advertising.
This must be after building the GATT profile
*/

func startAdvertising() {
let serviceUuids = [serviceUuid]
let advertisementData:[String: Any] = [

CBAdvertisementDataLocalNameKey: advertisingName,
CBAdvertisementDataServiceUUIDsKey: serviceUuids ]
peripheralManager.startAdvertising(advertisementData) }

/**
Build Gatt Profile.
This must be done after Bluetooth Radio has turned on
*/

func buildGattProfile() {
let service = CBMutableService(type: serviceUuid, primary: true) var rProperties
= CBCharacteristicProperties.read
rProperties.formUnion(CBCharacteristicProperties.notify) var rPermissions =
CBAttributePermissions.writable
rPermissions.formUnion(CBAttributePermissions.readable)
responseCharacteristic = CBMutableCharacteristic(
type: responseCharacteristicUuid,
properties: rProperties,
value: nil,
permissions: rPermissions)

```

```

let cProperties = CBCharacteristicProperties.write let cPermissions =
CBAAttributePermissions.writable commandCharacteristic =
CBMutableCharacteristic(

type: commandCharacteristicUuid,
properties: cProperties,
value: nil,
permissions: cPermissions)

service.characteristics = [
responseCharacteristic,
commandCharacteristic

]
peripheralManager.add(service)
}

/**
Make sense of the incoming byte array as a command
*/

func processCommand(bleCommandValue: [UInt8]) {
if bleCommandValue[bleCommandFooterPosition] == bleCommandFooter {
print ("Command found")
switch (bleCommandValue[bleCommandDataPosition]) { case
bleCommandLedOn:

print("Turning LED on")
setLedState(ledState: true)
case bleCommandLedOff:
print("Turning LED off")
setLedState(ledState: false)
default:
print("Unknown command value")
}
}
}
/**
Turn Camera Flash on as an LED

```

- Parameters:

```
- ledState: *true* for on, *false* for off
*/
```

```
func setLedState(ledState: Bool) {
if ledState {
sendBleResponse(ledState: bleResponseLedOn)

} else {
sendBleResponse(ledState: bleResponseLedOff)
}
delegate?.remoteLedPeripheral?(ledStateChangedTo: ledState) }
```

```
/**
```

Send a formatted response out via a Bluetooth Characteristic

- Parameters

- ledState: one of bleResponseLedOn or bleResponseLedOff \*/

```
func sendBleResponse(ledState: UInt8) {
var responseArray = [UInt8](
repeating: 0,
count: responseCharacteristicLength)

responseArray[bleResponseFooterPosition] = \ bleResponseConfirmationFooter
responseArray[bleResponseDataPosition] = ledState
let value = Data(bytes: responseArray)
responseCharacteristic.value = value
peripheralManager.updateValue(
value,
for: responseCharacteristic,
onSubscribedCentrals: nil)
// MARK: CBPeripheralManagerDelegate
```

```
/**
```

Peripheral will become active

```
*/
```

```
func peripheralManager(
_ peripheral: CBPeripheralManager, willRestoreState dict: [String : Any])
```

```

{
print("restoring peripheral state")
}

/**
Peripheral added a new Service
*/

func peripheralManager(
 _ peripheral: CBPeripheralManager, didAdd service: CBService,
error: Error?)

{
print("added service to peripheral") if error != nil {

print(error.debugDescription) }
}

/**
Peripheral started advertising
*/

func peripheralManagerDidStartAdvertising( _ peripheral:
CBPeripheralManager, error: Error?)

{
if error != nil {
print ("Error advertising peripheral")

print(error.debugDescription) }
self.peripheralManager = peripheral

delegate?.remoteLedPeripheral?(startedAdvertising: error) }

/**
Connected Central requested to read from a Characteristic */

func peripheralManager(
 _ peripheral: CBPeripheralManager,
didReceiveRead request: CBATTRequest)

```

```

{
let characteristic = request.characteristic
if (characteristic.uuid == responseCharacteristic.uuid) {

if let value = characteristic.value {
//let stringValue = String(data: value, encoding: .utf8)! if request.offset >
value.count {

peripheralManager.respond(
to: request,
withResult: CBALError.invalidOffset)

return
}
let range = Range(uncheckedBounds: (

lower: request.offset,
upper: value.count - request.offset))
request.value = value.subdata(in: range)
peripheral.respond(
to: request,
withResult: CBALError.success)
}
}
}

/** Connected Central requested to write to a Characteristic */

func peripheralManager(
_peripheral: CBPeripheralManager,
didReceiveWrite requests: [CBATTRequest])

{
for request in requests {
peripheral.respond(to: request, withResult: CBALError.success) print("new
request")
if let value = request.value {

let bleCommandValue = [UInt8](value)
processCommand(bleCommandValue: bleCommandValue) }
}
}

```

```

}
}

/**
Connected Central subscribed to a Characteristic */

func peripheralManager(
    _ peripheral: CBPeripheralManager,
    central: CBCentral,
    didSubscribeTo characteristic: CBCharacteristic)

{
    self.central = central
}

/**
Connected Central unsubscribed from a Characteristic */

func peripheralManager(
    _ peripheral: CBPeripheralManager,
    central: CBCentral,
    didUnsubscribeFrom characteristic: CBCharacteristic)

{
    self.central = central }
/**
Peripheral is about to notify subscribers of changes to a Characteristic
*/
func peripheralManagerIsReady(
toUpdateSubscribers peripheral: CBPeripheralManager)

{
    print("Peripheral about to update subscribers")
}

/**
Bluetooth Radio state changed
*/

func peripheralManagerDidUpdateState( _ peripheral: CBPeripheralManager)

```

```

{
peripheralManager = peripheral
switch peripheral.state {
case CBManagerState.poweredOn:
buildGattProfile()
startAdvertising()
default: break
}
delegate?.remoteLedPeripheral?(stateChanged: peripheral.state)
} }

```

## Delegates

The RemoteLedPeripheralDelegate will relay important events and state changes from the RemoteLedPeripheral, such as changes in the Advertising state and changes in the LED state:

### Example 13-5. Delegates/RemoteLedPeripheralDelegate.swift

```

import UIKit
import CoreBluetooth
@objc protocol RemoteLedPeripheralDelegate : class {
/**
RemoteLed State Changed

- Parameters:
- rssi: the RSSI
- blePeripheral: the BlePeripheral

*/
@objc optional func remoteLedPeripheral( stateChanged state:
CBManagerState)
/**
RemoteLed started advertising

- Parameters:
- error: the error message, if any */
@objc optional func remoteLedPeripheral( startedAdvertising error: Error?)
/**

```

LED turned on or off

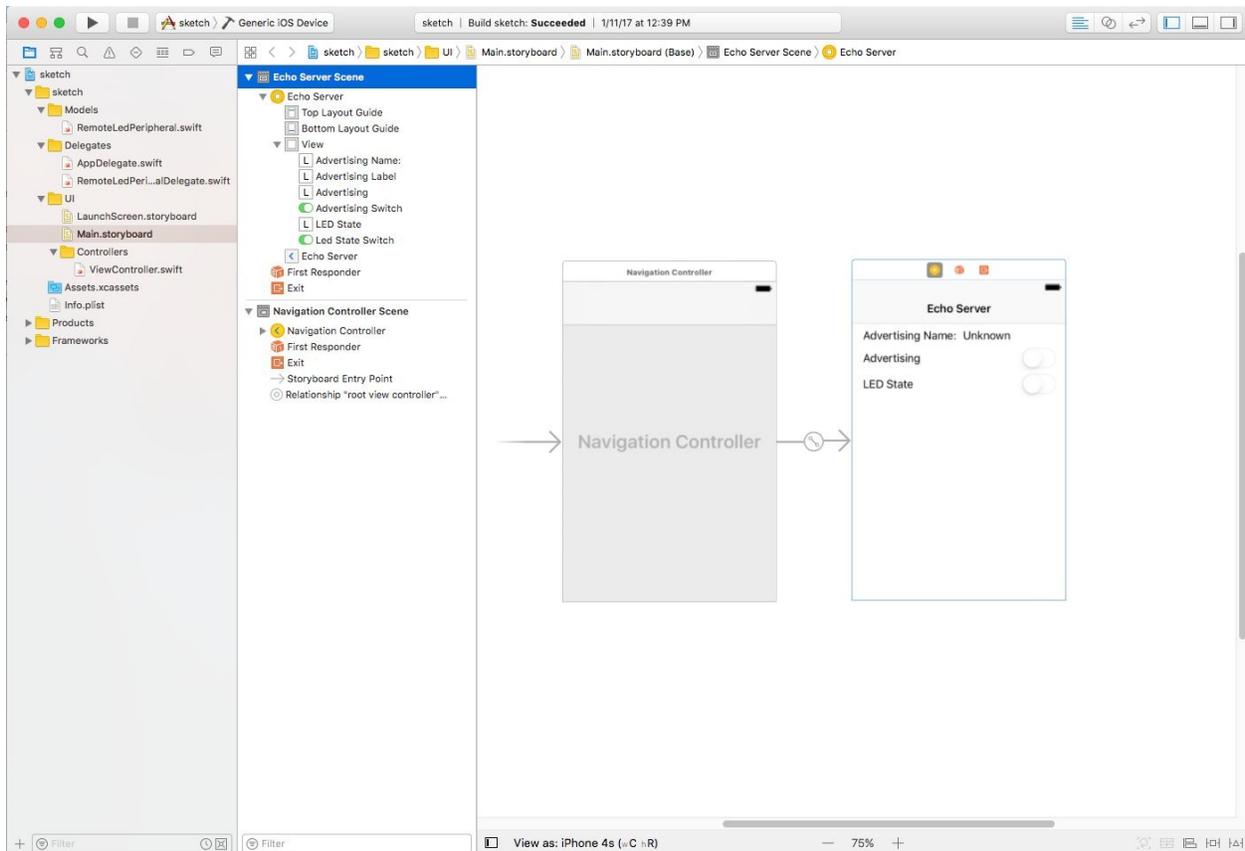
- Parameters:

- ledState: *\*true\** for on, *\*false\** for off *\*/*

@objc optional func remoteLedPeripheral( ledStateChangedTo ledState: Bool)

### **Storyboard**

Create the UISwitches, UITextViews, and UILabels in the UIView in the Main.storyboard to create the App's user interface (Figure 13-10).



**Figure 13-10. Project Storyboard Controllers**

The View will display the advertising state and LED state with UISwitches, and will turn the camera flash on and off in response to the incoming LED state:

**Example 13-6. UI/Controllers/ViewController.swift**

```
import UIKit
import CoreBluetooth import AVFoundation

class ViewController: UIViewController, RemoteLedPeripheralDelegate {

// MARK: UI Elements
@IBOutlet weak var advertisingLabel: UILabel! @IBOutlet weak var
advertisingSwitch: UISwitch! @IBOutlet weak var ledStateSwitch: UISwitch!

// MARK: BlePeripheral
// BlePeripheral
var remoteLed:RemoteLedPeripheral!

/**
```

```

UIView loaded
*/

override func viewDidLoad() { super.viewDidLoad()
}

/**
View appeared. Start the Peripheral
*/

override func viewWillAppear(_ animated: Bool) { remoteLed =
RemoteLedPeripheral(delegate: self) advertisingLabel.text =
remoteLed.advertisingName

}

/**
View will appear. Stop transmitting random data */

override func viewWillDisappear(_ animated: Bool) { remoteLed.stop()

}

/**
View disappeared. Stop advertising
*/

override func viewDidDisappear(_ animated: Bool) {
advertisingSwitch.setOn(false, animated: true)
}

// MARK: BlePeripheralDelegate
/**
RemoteLed state changed

- Parameters:
- state: the CBManagerState representing the new state */

func remoteLedPeripheral(stateChanged state: CBManagerState) { switch (state)
{
case CBManagerState.poweredOn:

```

```
print("Bluetooth on")
case CBManagerState.poweredOff:
print("Bluetooth off")
default:
print("Bluetooth not ready yet...")
}
}
```

```
/**
```

```
RemoteLed started advertising
```

```
- Parameters:
```

```
- error: the error message, if any
```

```
*/
```

```
func remoteLedPeripheral(startedAdvertising error: Error?) { if error != nil {
print("Problem starting advertising: " + error.debugDescription)
```

```
} else {
```

```
print("advertising started")
```

```
advertisingSwitch.setOn(true, animated: true)
```

```
}
```

```
}
```

```
/**
```

```
Led State Changed
```

```
- Parameters:
```

```
- stringValue: the value read from the Characteristic
```

```
- characteristic: the Characteristic that was written to
```

```
*/
```

```
func remoteLedPeripheral(ledStateChangedTo ledState: Bool) { if let device =
AVCaptureDevice.defaultDevice(
```

```
withMediaType: AVMediaTypeVideo)
```

```
{
```

```
if (device.hasTorch) { do {
```

```
try device.lockForConfiguration()
```

```
try device.setTorchModeOnWithLevel(1.0)
```

```
if ledState {
print("Led turned on")
device.torchMode = AVCaptureTorchMode.on ledStateSwitch.setOn(true,
animated: true) } else {
print("Led turned off")
device.torchMode = AVCaptureTorchMode.off ledStateSwitch.setOn(false,
animated: true) }
device.unlockForConfiguration()
} catch let error as NSError {
print("problem locking camera: "+error.debugDescription) }
}
} }
```

The resulting App can turn the camera flash on and off in response to commands from a connected Central. When the command is processed, a response is sent through the response Characteristic (Figure 13-11).

●●●● Fido 3G

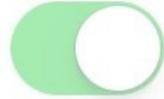
14:20

100%  ⚡

## Bluetooth Peripheral

Advertising Name: LedRemote

Advertising



LED State



●●●●○ Fido 3G

14:21

⌘ 100%  ⚡

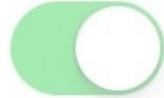
## Bluetooth Peripheral

Advertising Name: LedRemote

Advertising



LED State



**Figure 13-11. App screen showing LED in on and off states**

## Example code

The code for this chapter is available online  
at: <https://github.com/BluetoothLowEnergyIniOSSwift/Chapter13>

# Appendix

For reference, the following are properties of the Bluetooth Low Energy network and hardware.

**Range** 100 m (330 ft)

**Data Rate** 1M bit/s

**Application Throughput** 0.27 Mbit/s

**Security**

128-bit AES with Counter Mode CBC-MAC and application layer user defined (BEWARE: this encryption has vulnerabilities)

**Robustness**

Adaptive Frequency Hopping, Lazy Acknowledgement, 24-bit CRC, 32-bit Message Integrity Check

**Range** 100 m (330 ft)

**Data Rate** 1M bit/s

**Application Throughput** 0.27 Mbit/s

**Security**

**Peak Current Consumption**

**Byte-Order in Broadcast**

**Range**

**Data Rate**

**Application Throughput**

**Security**

128-bit AES with Counter Mode CBC-MAC and application layer user defined (BEWARE: this encryption has vulnerabilities)

< 15 mA

Big Endian (most significant bit at end)

100 m (330 ft)

1M bit/s

0.27 Mbit/s

128-bit AES with Counter Mode CBC-MAC and application layer user defined (BEWARE: this encryption has vulnerabilities)

**Source:** Wikipedia: Bluetooth\_Low\_Energy Retrieved from

[https://en.wikipedia.org/wiki/Bluetooth\\_low\\_energy](https://en.wikipedia.org/wiki/Bluetooth_low_energy)

## Appendix II: UUID Format

Bluetooth Low Energy has tight space requirements. Therefore it is preferred to transmit 16-bit UUIDs instead of 32-bit UUIDs. UUIDs can be converted between 16-bit and 32-bit with the standard Bluetooth Low Energy UUID format:

### Table II-1. 16-bit to 32-bit UUID Conversion Standard

| UUID Format | uuid16                               | Resulting uuid32                            |
|-------------|--------------------------------------|---------------------------------------------|
|             | 00000000-0000-1000-8000-00805f9b34fb | 0x2A56 00002A56-0000-1000-8000-00805f9b34fb |

## Appendix III: Minimal Recommended GATT

As a best practice, it is good to host a standard set of Services and Characteristics in a Peripheral's GATT Profile. These Characteristics allow connected Centrals to get the make and model number of the device, and the battery level if the Peripheral is battery-powered:

**Table III-1. Minimal GATT Profile**

| <b>GATT Type</b> | <b>Name</b>        | <b>Data Type</b> | <b>UUID</b> |
|------------------|--------------------|------------------|-------------|
| Service          | Device Information | Service          | 0x180a      |
| Characteristic   | Device Name        | char array       | 0x2a00      |
| Characteristic   | Model Number       | char array       | 0x2a24      |
| Characteristic   | Serial Number      | char array       | 0x2a04      |
| Service          | Battery Level      | 0x180f           |             |
| Characteristic   | Battery Level      | integer          | 0x2a19      |

## Appendix IV: Reserved GATT Services

Services act as a container for Characteristics or other Services, providing a tree-like structure for organizing Bluetooth I/O.

These Services UUIDs have been reserved for special contexts, such as Device Information (0x180A) Which may contain Characteristics that communicate information about the Peripheral's name, version number, or settings.

*Note: All Bluetooth Peripherals should have a Battery Service (0x180F) Service containing a Battery Level (0x2A19) Characteristic.*

**Table IV-1. Reserved GATT Services**

### **Specification Name UUID Specification Type**

Alert Notification Service 0x1811 org.bluetooth.service.alert\_notification

Automation IO 0x1815 org.bluetooth.service.automation\_io

Battery Service 0x180F org.bluetooth.service.battery\_service

Blood Pressure 0x1810 org.bluetooth.service.blood\_pressure

Body Composition 0x181B org.bluetooth.service.body\_composition

Bond Management 0x181E org.bluetooth.service.bond\_management

Continuous Glucose Monitoring

0x181F org.bluetooth.service.continuous\_glucose\_monitoring Current Time Service

0x1805 org.bluetooth.service.current\_time

Cycling Power 0x1818 org.bluetooth.service.cycling\_power

Cycling Speed and Cadence 0x1816

org.bluetooth.service.cycling\_speed\_and\_cadence

Device Information 0x180A org.bluetooth.service.device\_information

Environmental Sensing 0x181A org.bluetooth.service.environmental\_sensing

Generic Access 0x1800 org.bluetooth.service.generic\_access

Generic Attribute 0x1801 org.bluetooth.service.generic\_attribute

Glucose 0x1808 org.bluetooth.service.glucose

Health Thermometer 0x1809 org.bluetooth.service.health\_thermometer

Heart Rate 0x180D org.bluetooth.service.heart\_rate

HTTP Proxy 0x1823 org.bluetooth.service.http\_proxy

Human Interface Device 0x1812 org.bluetooth.service.human\_interface\_device

Immediate Alert 0x1802 org.bluetooth.service.immediate\_alert  
Indoor Positioning 0x1821 org.bluetooth.service.indoor\_positioning  
Internet Protocol Support 0x1820  
org.bluetooth.service.internet\_protocol\_support Link Loss 0x1803  
org.bluetooth.service.link\_loss  
Location and Navigation 0x1819 org.bluetooth.service.location\_and\_navigation  
Next DST Change Service 0x1807 org.bluetooth.service.next\_dst\_change  
Object Transfer 0x1825 org.bluetooth.service.object\_transfer  
Phone Alert Status Service 0x180E org.bluetooth.service.phone\_alert\_status  
Pulse Oximeter 0x1822 org.bluetooth.service.pulse\_oximeter  
Reference Time Update Service  
0x1806 org.bluetooth.service.reference\_time\_update  
Running Speed and Cadence 0x1814  
org.bluetooth.service.running\_speed\_and\_cadence  
Scan Parameters 0x1813 org.bluetooth.service.scan\_parameters  
Transport Discovery 0x1824 org.bluetooth.service.transport\_discovery  
Tx Power 0x1804 org.bluetooth.service.tx\_power  
User Data 0x181C org.bluetooth.service.user\_data  
Weight Scale 0x181D org.bluetooth.service.weight\_scale

**Source:** Bluetooth SIG: GATT Services Retrieved from  
<https://www.bluetooth.com/specifications/gatt/services>

## Appendix V: Reserved GATT Characteristics

Characteristics act a data port that can be read from or written to.

These Characteristic UUIDs have been reserved for specific types of data, such as Device Name (0x2A00) which may read the Peripheral's current battery level.

*Note: All Bluetooth Peripherals should have a Battery Level (0x2A19)*

*Characteristic, contained inside a Battery Service (0x180F) Service.*

### Table V-1. Reserved GATT Characteristics

#### Specification Name UUID Specification Type

Aerobic Heart Rate Lower Limit

0x2A7E org.bluetooth.characteristic.aerobic\_heart\_rate\_lower\_limit

Aerobic Heart Rate Upper Limit

0x2A84 org.bluetooth.characteristic.aerobic\_heart\_rate\_upper\_limit

Aerobic Threshold 0x2A7F org.bluetooth.characteristic.aerobic\_threshold

Age 0x2A80 org.bluetooth.characteristic.age

Aggregate 0x2A5A org.bluetooth.characteristic.aggregate

Alert Category ID 0x2A43 org.bluetooth.characteristic.alert\_category\_id

Alert Category ID Bit Mask 0x2A42

org.bluetooth.characteristic.alert\_category\_id\_bit\_mask

Alert Level 0x2A06 org.bluetooth.characteristic.alert\_level

Alert Notification Control Point

0x2A44 org.bluetooth.characteristic.alert\_notification\_control\_point Alert Status

0x2A3F org.bluetooth.characteristic.alert\_status

Altitude 0x2AB3 org.bluetooth.characteristic.altitude

Anaerobic Heart Rate Lower Limit

0x2A81 org.bluetooth.characteristic.anaerobic\_heart\_rate\_lower\_limit

Anaerobic Heart Rate Upper Limit

0x2A82 org.bluetooth.characteristic.anaerobic\_heart\_rate\_upper\_limit

Anaerobic Threshold 0x2A83 org.bluetooth.characteristic.anaerobic\_threshold

Analog 0x2A58 org.bluetooth.characteristic.analog

Apparent Wind Direction 0x2A73

org.bluetooth.characteristic.apparent\_wind\_direction

Apparent Wind Speed 0x2A72 org.bluetooth.characteristic.apparent\_wind\_speed

Appearance 0x2A01 org.bluetooth.characteristic.gap.appearance

Barometric Pressure Trend

0x2AA3 org.bluetooth.characteristic.barometric\_pressure\_trend

Battery Level 0x2A19 org.bluetooth.characteristic.battery\_level

Blood Pressure Feature 0x2A49  
org.bluetooth.characteristic.blood\_pressure\_feature  
Blood Pressure Measurement 0x2A35  
org.bluetooth.characteristic.blood\_pressure\_measurement  
Body Composition Feature  
0x2A9B org.bluetooth.characteristic.body\_composition\_feature  
Body Composition Measurement 0x2A9C  
org.bluetooth.characteristic.body\_composition\_measurement  
Body Sensor Location 0x2A38  
org.bluetooth.characteristic.body\_sensor\_location Bond Management Control  
Point 0x2AA4 org.bluetooth.characteristic.bond\_management\_control\_point  
Bond Management Feature  
0x2AA5 org.bluetooth.characteristic.bond\_management\_feature  
Boot Keyboard Input Report  
0x2A22 org.bluetooth.characteristic.boot\_keyboard\_input\_report  
Boot Keyboard Output Report  
Boot Mouse Input Report  
Central Address Resolution  
0x2A32 org.bluetooth.characteristic.boot\_keyboard\_output\_report  
0x2A33 org.bluetooth.characteristic.boot\_mouse\_input\_report  
  
0x2AA6  
org.bluetooth.characteristic.gap.central\_address\_resolution\_su pport  
  
CGM Feature 0x2AA8 org.bluetooth.characteristic.cgm\_feature  
CGM Measurement 0x2AA7 org.bluetooth.characteristic.cgm\_measurement  
CGM Session Run Time  
0x2AAB org.bluetooth.characteristic.cgm\_session\_run\_time  
CGM Session Start Time  
0x2AAA org.bluetooth.characteristic.cgm\_session\_start\_time  
CGM Specific Ops Control Point 0x2AAC  
org.bluetooth.characteristic.cgm\_specific\_ops\_control\_point  
CGM Status 0x2AA9 org.bluetooth.characteristic.cgm\_status  
CSC Feature 0x2A5C org.bluetooth.characteristic.csc\_feature  
CSC Measurement 0x2A5B org.bluetooth.characteristic.csc\_measurement  
Current Time 0x2A2B org.bluetooth.characteristic.current\_time  
Cycling Power Control Point  
  
0x2A66 org.bluetooth.characteristic.cycling\_power\_control\_point Cycling

Power Feature

0x2A65 org.bluetooth.characteristic.cycling\_power\_feature

Cycling Power Measurement 0x2A63

org.bluetooth.characteristic.cycling\_power\_measurement

Cycling Power Vector 0x2A64

org.bluetooth.characteristic.cycling\_power\_vector

Database Change Increment

0x2A99 org.bluetooth.characteristic.database\_change\_increment

Date of Birth 0x2A85 org.bluetooth.characteristic.date\_of\_birth

Date of Threshold Assessment 0x2A86

org.bluetooth.characteristic.date\_of\_threshold\_assessment

Date Time 0x2A08 org.bluetooth.characteristic.date\_time

Day Date Time 0x2A0A org.bluetooth.characteristic.day\_date\_time

Day of Week 0x2A09 org.bluetooth.characteristic.day\_of\_week

Descriptor Value Changed

0x2A7D org.bluetooth.characteristic.descriptor\_value\_changed

Device Name 0x2A00 org.bluetooth.characteristic.gap.device\_name

Dew Point 0x2A7B org.bluetooth.characteristic.dew\_point

Digital 0x2A56 org.bluetooth.characteristic.digital

DST Offset 0x2A0D org.bluetooth.characteristic.dst\_offset

Elevation 0x2A6C org.bluetooth.characteristic.elevation

Email Address 0x2A87 org.bluetooth.characteristic.email\_address Exact Time

256 0x2A0C org.bluetooth.characteristic.exact\_time\_256

Fat Burn Heart Rate Lower Limit

0x2A88 org.bluetooth.characteristic.fat\_burn\_heart\_rate\_lower\_limit

Fat Burn Heart Rate Upper Limit

0x2A89 org.bluetooth.characteristic.fat\_burn\_heart\_rate\_upper\_limit

Firmware Revision String

0x2A26 org.bluetooth.characteristic.firmware\_revision\_string

First Name 0x2A8A org.bluetooth.characteristic.first\_name

Five Zone Heart Rate Limits

0x2A8B org.bluetooth.characteristic.five\_zone\_heart\_rate\_limits

Floor Number 0x2AB2 org.bluetooth.characteristic.floor\_number

Gender 0x2A8C org.bluetooth.characteristic.gender

Glucose Feature 0x2A51 org.bluetooth.characteristic.glucose\_feature

Glucose Measurement 0x2A18

org.bluetooth.characteristic.glucose\_measurement

Glucose Measurement Context

0x2A34 org.bluetooth.characteristic.glucose\_measurement\_context  
Gust Factor 0x2A74 org.bluetooth.characteristic.gust\_factor  
Hardware Revision String  
0x2A27 org.bluetooth.characteristic.hardware\_revision\_string  
Heart Rate Control Point  
0x2A39 org.bluetooth.characteristic.heart\_rate\_control\_point  
Heart Rate Max 0x2A8D org.bluetooth.characteristic.heart\_rate\_max  
Heart Rate Measurement

0x2A37 org.bluetooth.characteristic.heart\_rate\_measurement Heat Index  
0x2A7A org.bluetooth.characteristic.heat\_index

Height 0x2A8E org.bluetooth.characteristic.height  
HID Control Point 0x2A4C org.bluetooth.characteristic.hid\_control\_point  
HID Information 0x2A4A org.bluetooth.characteristic.hid\_information  
Hip Circumference 0x2A8F org.bluetooth.characteristic.hip\_circumference  
HTTP Control Point 0x2ABA org.bluetooth.characteristic.http\_control\_point  
HTTP Entity Body 0x2AB9 org.bluetooth.characteristic.http\_entity\_body  
HTTP Headers 0x2AB7 org.bluetooth.characteristic.http\_headers  
HTTP Status Code 0x2AB8 org.bluetooth.characteristic.http\_status\_code  
HTTPS Security 0x2ABB org.bluetooth.characteristic.https\_security  
Humidity 0x2A6F org.bluetooth.characteristic.humidity  
IEEE 11073-20601 Regulatory  
Certification Data List  
Indoor Positioning Configuration  
Intermediate Cuff Pressure  
Intermediate  
Temperature

0x2A2A  
org.bluetooth.characteristic.ieee\_11073-20601\_regulatory\_cert  
ification\_data\_list

0x2AAD org.bluetooth.characteristic.indoor\_positioning\_configuration  
0x2A36 org.bluetooth.characteristic.intermediate\_cuff\_pressure  
0x2A1E org.bluetooth.characteristic.intermediate\_temperature  
Irradiance 0x2A77 org.bluetooth.characteristic.irradiance Language 0x2AA2  
org.bluetooth.characteristic.language  
Last Name 0x2A90 org.bluetooth.characteristic.last\_name

Latitude 0x2AAE org.bluetooth.characteristic.latitude  
LN Control Point 0x2A6B org.bluetooth.characteristic.ln\_control\_point  
LN Feature 0x2A6A org.bluetooth.characteristic.ln\_feature  
Local East Coordinate 0x2AB1  
org.bluetooth.characteristic.local\_east\_coordinate  
Local North 0x2AB0 org.bluetooth.characteristic.local\_north\_coordinateCoordinate  
Local Time Information 0x2A0F  
org.bluetooth.characteristic.local\_time\_information  
Location and Speed 0x2A67 org.bluetooth.characteristic.location\_and\_speed  
Location Name 0x2AB5 org.bluetooth.characteristic.location\_name  
Longitude 0x2AAF org.bluetooth.characteristic.longitude  
Magnetic Declination 0x2A2C org.bluetooth.characteristic.magnetic\_declination  
Magnetic Flux Density 0x2AA0  
org.bluetooth.characteristic.magnetic\_flux\_density\_2D\_2D  
Magnetic Flux Density 0x2AA1  
org.bluetooth.characteristic.magnetic\_flux\_density\_3D\_3D  
Manufacturer Name 0x2A29  
org.bluetooth.characteristic.manufacturer\_name\_stringString  
Maximum  
org.bluetooth.characteristic.maximum\_recommended\_heart\_rateRecommended Heart  
0x2A91 teRate  
Measurement Interval 0x2A21  
org.bluetooth.characteristic.measurement\_interval  
Model Number String 0x2A24  
org.bluetooth.characteristic.model\_number\_string  
Navigation 0x2A68 org.bluetooth.characteristic.navigation  
New Alert 0x2A46 org.bluetooth.characteristic.new\_alert  
Object Action Control Point  
0x2AC5 org.bluetooth.characteristic.object\_action\_control\_point  
Object Changed 0x2AC8 org.bluetooth.characteristic.object\_changed  
Object First-Created 0x2AC1 org.bluetooth.characteristic.object\_first\_created  
Object ID 0x2AC3 org.bluetooth.characteristic.object\_id  
Object Last-Modified 0x2AC2 org.bluetooth.characteristic.object\_last\_modified  
Object List Control Point  
0x2AC6 org.bluetooth.characteristic.object\_list\_control\_point  
Object List Filter 0x2AC7 org.bluetooth.characteristic.object\_list\_filter  
Object Name 0x2ABE org.bluetooth.characteristic.object\_name  
Object Properties 0x2AC4 org.bluetooth.characteristic.object\_properties

Object Size 0x2AC0 org.bluetooth.characteristic.object\_size  
Object Type 0x2ABF org.bluetooth.characteristic.object\_type  
OTS Feature 0x2ABD org.bluetooth.characteristic.ots\_feature

Peripheral Preferred Connection  
Parameters

Peripheral Privacy Flag  
PLX Continuous Measurement

0x2A04  
org.bluetooth.characteristic.gap.peripheral\_preferred\_connection\_parameters

0x2A02 org.bluetooth.characteristic.gap.peripheral\_privacy\_flag  
0x2A5F org.bluetooth.characteristic.plx\_continuous\_measurement  
PLX Features 0x2A60 org.bluetooth.characteristic.plx\_features  
PLX Spot-Check Measurement 0x2A5E  
org.bluetooth.characteristic.plx\_spot\_check\_measurement  
PnP ID 0x2A50 org.bluetooth.characteristic.pnp\_id  
Pollen Concentration 0x2A75 org.bluetooth.characteristic.pollen\_concentration  
Position Quality 0x2A69 org.bluetooth.characteristic.position\_quality  
Pressure 0x2A6D org.bluetooth.characteristic.pressure  
Protocol Mode 0x2A4E org.bluetooth.characteristic.protocol\_mode  
Rainfall 0x2A78 org.bluetooth.characteristic.rainfall  
Reconnection Address 0x2A03  
org.bluetooth.characteristic.gap.reconnection\_address  
Record Access Control Point  
0x2A52 org.bluetooth.characteristic.record\_access\_control\_point  
Reference Time Information 0x2A14  
org.bluetooth.characteristic.reference\_time\_information  
Report 0x2A4D org.bluetooth.characteristic.report  
Report Map 0x2A4B org.bluetooth.characteristic.report\_map <sup>Resolvable Private</sup>  
0x2AC9 org.bluetooth.characteristic.resolvable\_private\_address\_only <sup>Address Only</sup>  
Resting Heart Rate 0x2A92 org.bluetooth.characteristic.resting\_heart\_rate  
Ringer Control Point 0x2A40 org.bluetooth.characteristic.ringer\_control\_point  
Ringer Setting 0x2A41 org.bluetooth.characteristic.ringer\_setting  
RSC Feature 0x2A54 org.bluetooth.characteristic.rsc\_feature  
RSC Measurement 0x2A53 org.bluetooth.characteristic.rsc\_measurement  
SC Control Point 0x2A55 org.bluetooth.characteristic.sc\_control\_point

Scan Interval Window 0x2A4F  
org.bluetooth.characteristic.scan\_interval\_window  
Scan Refresh 0x2A31 org.bluetooth.characteristic.scan\_refresh  
Sensor Location 0x2A5D org.bluetooth.characteristic.sensor\_location  
Serial Number String 0x2A25 org.bluetooth.characteristic.serial\_number\_string  
Service Changed 0x2A05 org.bluetooth.characteristic.gatt.service\_changed  
Software Revision 0x2A28 org.bluetooth.characteristic.software\_revision\_stringString  
Sport Type for Aerobic  
org.bluetooth.characteristic.sport\_type\_for\_aerobic\_and\_anaerobic\_and Anaerobic 0x2A93  
obic\_thresholdsThresholds  
Supported New Alert 0x2A47  
org.bluetooth.characteristic.supported\_new\_alert\_categoryCategory  
Supported Unread 0x2A48  
org.bluetooth.characteristic.supported\_unread\_alert\_categoryAlert Category  
System ID 0x2A23 org.bluetooth.characteristic.system\_id  
TDS Control Point 0x2ABC org.bluetooth.characteristic.tds\_control\_point  
Temperature 0x2A6E org.bluetooth.characteristic.temperature  
Temperature Measurement 0x2A1C  
org.bluetooth.characteristic.temperature\_measurement  
Temperature Type 0x2A1D org.bluetooth.characteristic.temperature\_type  
Three Zone Heart Rate Limits 0x2A94  
org.bluetooth.characteristic.three\_zone\_heart\_rate\_limits  
Time Accuracy 0x2A12 org.bluetooth.characteristic.time\_accuracy  
Time Source 0x2A13 org.bluetooth.characteristic.time\_source  
Time Update Control Point  
0x2A16 org.bluetooth.characteristic.time\_update\_control\_point  
Time Update State 0x2A17 org.bluetooth.characteristic.time\_update\_state  
Time with DST 0x2A11 org.bluetooth.characteristic.time\_with\_dst  
Time Zone 0x2A0E org.bluetooth.characteristic.time\_zone  
True Wind Direction 0x2A71 org.bluetooth.characteristic.true\_wind\_direction  
True Wind Speed 0x2A70 org.bluetooth.characteristic.true\_wind\_speed  
Two Zone Heart Rate Limit  
0x2A95 org.bluetooth.characteristic.two\_zone\_heart\_rate\_limit  
Tx Power Level 0x2A07 org.bluetooth.characteristic.tx\_power\_level  
Uncertainty 0x2AB4 org.bluetooth.characteristic.uncertainty  
Unread Alert Status 0x2A45 org.bluetooth.characteristic.unread\_alert\_status  
URI 0x2AB6 org.bluetooth.characteristic.uri  
User Control Point 0x2A9F org.bluetooth.characteristic.user\_control\_point

User Index 0x2A9A org.bluetooth.characteristic.user\_index  
UV Index 0x2A76 org.bluetooth.characteristic.uv\_index  
VO2 Max 0x2A96 org.bluetooth.characteristic.vo2\_max  
Waist Circumference 0x2A97 org.bluetooth.characteristic.waist\_circumference  
Weight 0x2A98 org.bluetooth.characteristic.weight  
Weight Measurement 0x2A9D org.bluetooth.characteristic.weight\_measurement  
Weight Scale Feature 0x2A9E org.bluetooth.characteristic.weight\_scale\_feature  
Wind Chill 0x2A79 org.bluetooth.characteristic.wind\_chill

**Source:** Bluetooth SIG: GATT Characteristics Retrieved from  
<https://www.bluetooth.com/specifications/gatt/characteristics>

## Appendix VI: GATT Descriptors

The following GATT Descriptor UUIDs have been reserved for specific uses.

GATT Descriptors describe features within a Characteristic that can be altered, for instance, the Client Characteristic Configuration (0x2902) which can be flagged to allow a connected Central to subscribe to notifications on a Characteristic.

**Table VI-1. Reserved GATT Descriptors**

| <b>Specification Name</b>                                         | <b>UUID</b>                                                      | <b>Specification Type</b>                     |
|-------------------------------------------------------------------|------------------------------------------------------------------|-----------------------------------------------|
| Characteristic Aggregate Format                                   | 0x2905                                                           |                                               |
| org.bluetooth.descriptor.gatt.characteristic_aggregate_format     |                                                                  |                                               |
| Characteristic Extended Properties                                |                                                                  |                                               |
| 0x2900                                                            | org.bluetooth.descriptor.gatt.characteristic_extended_properties |                                               |
| Characteristic Presentation Format                                | 0x2904                                                           |                                               |
| org.bluetooth.descriptor.gatt.characteristic_presentation_format  |                                                                  |                                               |
| Characteristic User Description                                   |                                                                  |                                               |
| 0x2901                                                            | org.bluetooth.descriptor.gatt.characteristic_user_description    |                                               |
| Client Characteristic Configuration                               | 0x2902                                                           |                                               |
| org.bluetooth.descriptor.gatt.client_characteristic_configuration |                                                                  |                                               |
| Environmental Sensing Configuration                               |                                                                  |                                               |
| 0x290B                                                            | org.bluetooth.descriptor.es_configuration                        |                                               |
| Environmental Sensing Measurement                                 |                                                                  |                                               |
| 0x290C                                                            | org.bluetooth.descriptor.es_measurement                          |                                               |
| Environmental Sensing Trigger Setting                             | 0x290D                                                           |                                               |
| org.bluetooth.descriptor.es_trigger_setting                       |                                                                  |                                               |
| External Report Reference                                         |                                                                  |                                               |
| 0x2907                                                            | org.bluetooth.descriptor.external_report_reference               | Number of Digitals                            |
| 0x2909                                                            | org.bluetooth.descriptor.number_of_digitals                      |                                               |
| Report Reference                                                  | 0x2908                                                           | org.bluetooth.descriptor.report_reference     |
| Server Characteristic Configuration                               | 0x2903                                                           |                                               |
| org.bluetooth.descriptor.gatt.server_characteristic_configuration |                                                                  |                                               |
| Time Trigger Setting                                              | 0x290E                                                           | org.bluetooth.descriptor.time_trigger_setting |
| Valid Range                                                       | 0x2906                                                           | org.bluetooth.descriptor.valid_range          |

Value Trigger Setting 0x290A org.bluetooth.descriptor.value\_trigger\_setting

**Source:** Bluetooth SIG: GATT Descriptors Retrieved from  
<https://www.bluetooth.com/specifications/gatt/descriptors>

## Appendix VII: Company Identifiers

The following companies have specific Manufacturer Identifiers, which identify Bluetooth devices in the Generic Access Profile (GAP). Peripherals with no specific manufacturer use ID 65535 (0xffff). All other IDs are reserved, even if not yet assigned.

This is a non-exhaustive list of companies. A full list and updated can be found on the Bluetooth SIG website.

### Table VII-1. Company Identifiers

#### Decimal Hexadecimal Company

0 0x0000 Ericsson Technology Licensing

1 0x0001 Nokia Mobile Phones

2 0x0002 Intel Corp.

3 0x0003 IBM Corp.

4 0x0004 Toshiba Corp.

5 0x0005 3Com

6 0x0006 Microsoft

7 0x0007 Lucent

8 0x0008 Motorola

13 0x000D Texas Instruments Inc.

19 0x0013 Atmel Corporation

29 0x001D Qualcomm

36 0x0024 Alcatel

37 0x0025 NXP Semiconductors (formerly Philips Semiconductors)

60 0x003C BlackBerry Limited (formerly Research In Motion)

76 0x004C Apple, Inc.

86 0x0056 Sony Ericsson Mobile Communications

89 0x0059 Nordic Semiconductor ASA

92 0x005C Belkin International, Inc.

93 0x005D Realtek Semiconductor Corporation

101 0x0065 Hewlett-Packard Company

104 0x0068 General Motors

117 0x0075 Samsung Electronics Co. Ltd. 120 0x0078 Nike, Inc.

135 0x0087 Garmin International, Inc.

138 0x008A Jawbone  
184 0x00B8 Qualcomm Innovation Center, Inc. (QuIC)  
215 0x00D7 Qualcomm Technologies, Inc.  
216 0x00D8 Qualcomm Connected Experiences, Inc.  
220 0x00DC Procter & Gamble  
224 0x00E0 Google  
359 0x0167 Bayer HealthCare  
367 0x016F Podo Labs, Inc  
369 0x0171 Amazon Fulfillment Service  
387 0x0183 Walt Disney  
398 0x018E Fitbit, Inc.  
425 0x01A9 Canon Inc.  
427 0x01AB Facebook, Inc. 474 0x01DA Logitech International SA  
558 0x022E Siemens AG  
605 0x025D Lexmark International Inc.  
637 0x027D HUAWEI Technologies Co., Ltd. ( )  
720 0x02D0 3M  
876 0x036C Zipcar  
897 0x0381 Sharp Corporation  
921 0x0399 Nikon Corporation  
1117 0x045D Boston Scientific Corporation  
65535 0xFFFF No Device ID

**Source :** Bluetooth SIG: Company Identifiers Retrieved from  
<https://www.bluetooth.com/specifications/assigned-numbers/company-identifiers>

# Glossary

The following is a list of Bluetooth Low Energy terms and their meanings.

**Attribute** - An unit of a GATT Profile which can be accessed by a Central, such as a Service or a Characteristic.

**Beacon** - A Bluetooth Low Energy Peripheral which continually Broadcasts so that Centrals can discern their location from information gleaned from the properties of the broadcast.

**Bluetooth Low Energy (BLE)** - A low power, short range wireless protocol used on micro electronics.

**Broadcast** - A feature of Bluetooth Low Energy where a Peripheral outputs a name and other specific data about a itself

**Central** - A Bluetooth Low Energy device that can connect to several Peripherals.

**Channel** - A finely-tuned radio frequency used for Broadcasting or data transmission.

**Characteristic** - A port or data endpoint where data can be read or written.

**Descriptor** - A feature of a Characteristic that allows for some sort of data interaction, such as Read, Write, or Notify.

**E0** - The encryption algorithm built into Bluetooth Low Energy.

**Generic Attribute (GATT) Profile** - A list of Services and Characteristics which are unique to a Peripheral and describe how data is served from the Peripheral. GATT profiles are hosted by a Peripheral

**iBeacon** - An Apple compatible Beacon which allows a Central to download a specific packet of data to inform the Central of its absolute location and other properties.

**Notify** - An operation where a Peripheral alerts a Central of a change in data.

**Peripheral** - A Bluetooth Low Energy device that can connect to a single Central. Peripherals host a Generic Attribute (GATT) profile.

**Read** - An operation where a Central downloads data from a Characteristic.

**Scan** - The process of a Central searching for Broadcasting Peripherals.

**Scan Response** - A feature of Bluetooth Low Energy which allows Centrals to download a small packet of data without connecting.

**Service** - A container structure used to organize data endpoints. Services are

hosted by a Peripheral.

**Universally Unique Identifier (UUID)** - A long, randomly generated alphanumeric string that is unique regardless of where it's used. UUIDs are designed to avoid name collisions that may happen when countless programs are interacting with each other.

**Write** - An operation where a Central alters data on a Characteristic. (This page intentionally left blank)



## About the Author



essential truth that defines a company. Tony's infinite curiosity compels him to want to open up and learn about everything he touches, and his excitement compels him to share what he learns with others.

He has two true passions: branding and inventing.

His passion for branding led him to start a company that did branding and marketing in 4 countries for firms such as Apple, Intel, and Sony BMG. He loves weaving the elements of design, writing, product, and strategy into an

His passion for inventing led him to start a company that uses brain imaging to quantify meditation and to predict seizures, a company acquired \$1.5m in funding and was incubated in San Francisco where he currently resides.

Those same passions have led him on some adventures as well, including living in a Greek monastery with orthodox monks and to tagging along with a gypsy in Spain to learn to play flamenco guitar.

371

(This page intentionally left blank)



## **About this Book**

This book is a practical guide to programming Bluetooth Low Energy for iPhone and iPad.

In this book, you will learn the basics of how to program an iOS device to communicate with any Central or Peripheral device over Bluetooth Low Energy. Each chapter of the book builds on the previous one, culminating in three projects:

- A Beacon and Scanner
- A Echo Server and Client
- A Remote Controlled Device

Through the course of the book you will learn important concepts that relate to:

- How Bluetooth Low Energy works
- How data is sent and received
- Common paradigms for handling data

### **Skill Level**

This book is excellent for anyone who has basic or advanced knowledge of iOS programming in SWIFT.

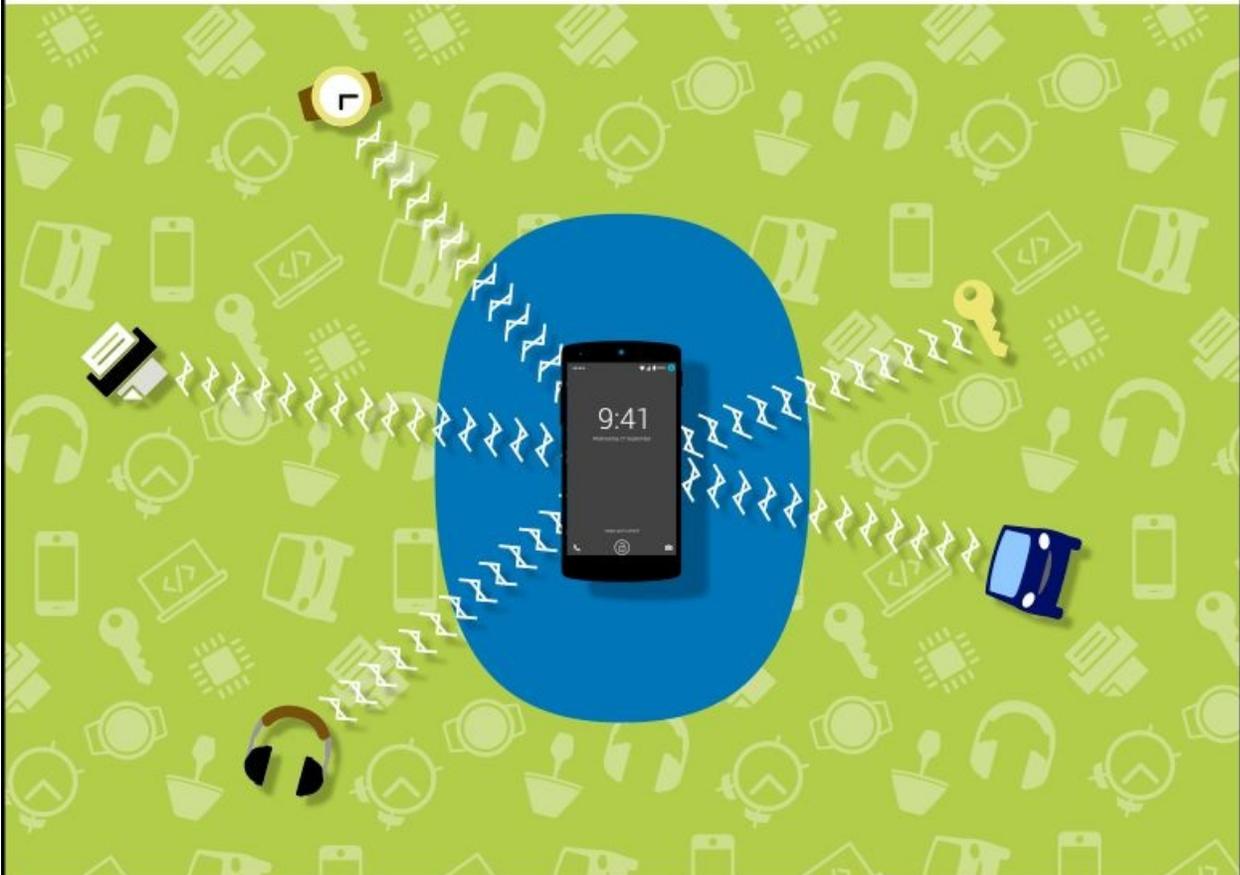
## **Other Books in this Series**

If you are interested in programming other Bluetooth Low Energy Devices, please check out the other books in this series or visit [bluetoothlowenergybooks.com](http://bluetoothlowenergybooks.com):

TURN YOUR ANDROID PHONE OR TABLET INTO A  
SENSOR • REMOTE CONTROL • BEACON • TRANSMITTER  
USING

# Bluetooth Low Energy

IN ANDROID JAVA



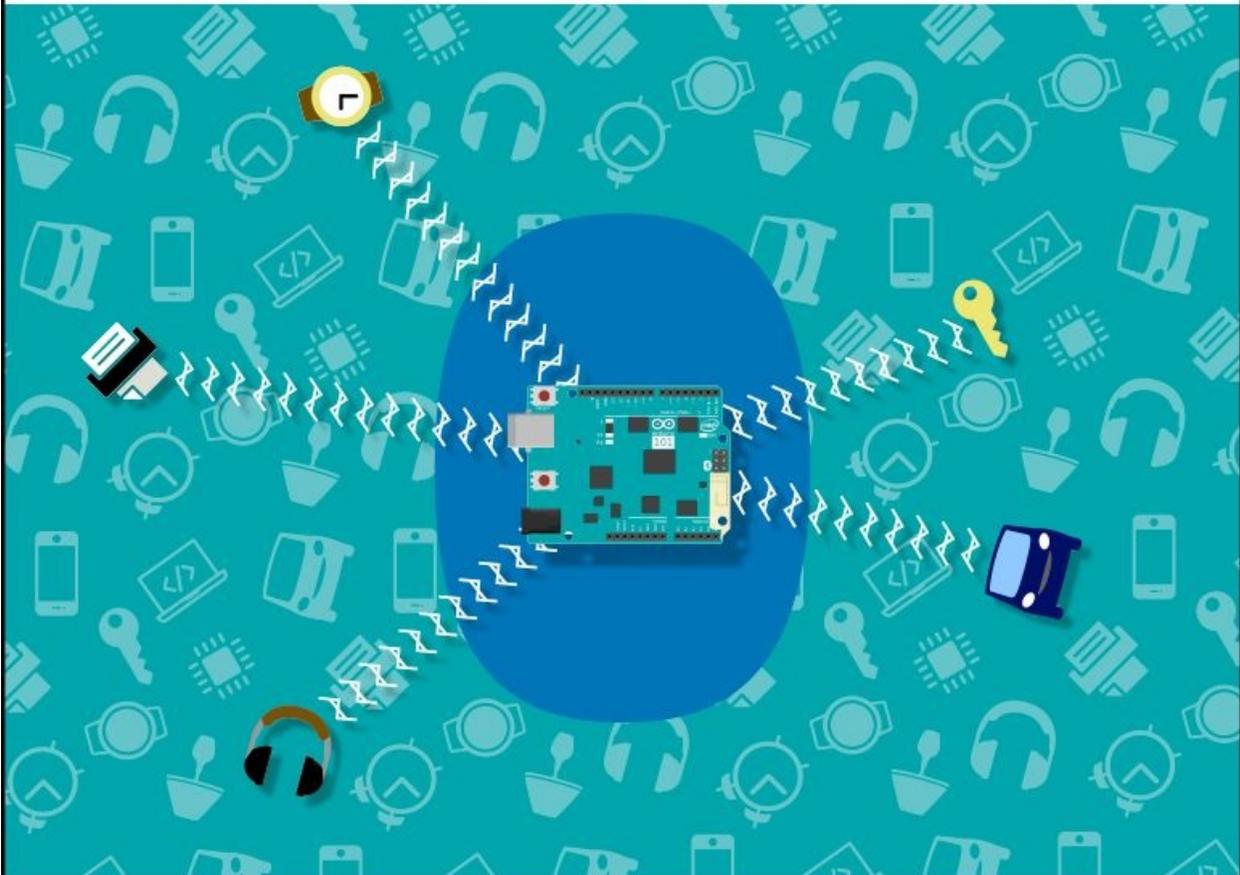
**Tony Gaitatzis**

2017

TURN YOUR ARDUINO 101 INTO A  
SENSOR • REMOTE CONTROL • BEACON • TRANSMITTER  
USING

# Bluetooth Low Energy

IN ARDUINO C++



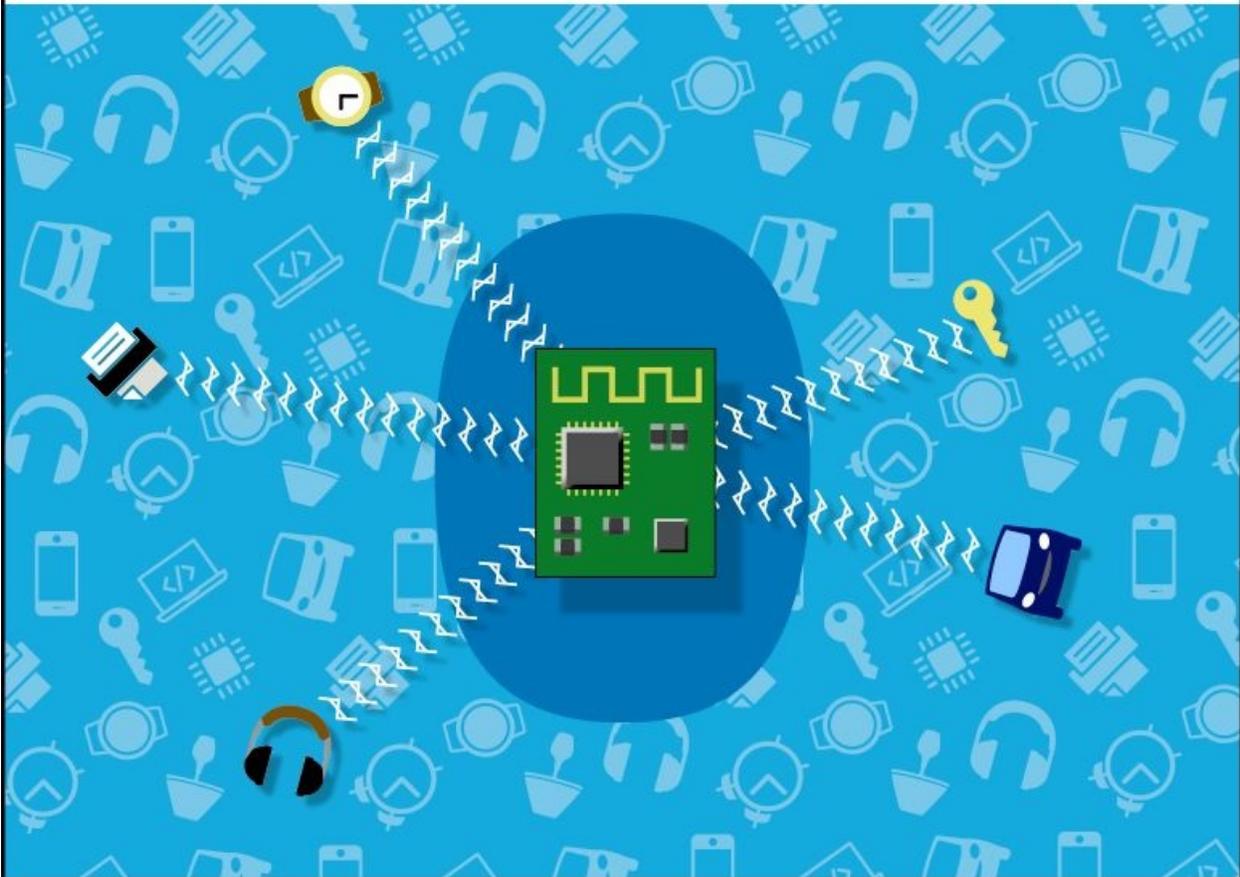
**Tony Gaitatzis**

2017

TURN YOUR NRF51822 / NRF52832 INTO A  
SENSOR • REMOTE CONTROL • BEACON • TRANSMITTER  
USING

# Bluetooth Low Energy

IN C++



**Tony Gaitatzis**

2017

**Bluetooth Low Energy Programming in Android Java**

Tony Gaitatzis, 2017

ISBN: 978-1-7751280-1-4

**Bluetooth Low Energy Programming in Arduino C++**

Tony Gaitatzis, 2017

ISBN: 978-1-7751280-2-1

**Bluetooth Low Energy Programming in C++ for nRF51822 and nRF52832**

[Tony Gaitatzis, 2017](#)

ISBN: 978-1-7751280-3-8 (This page intentionally left blank)

