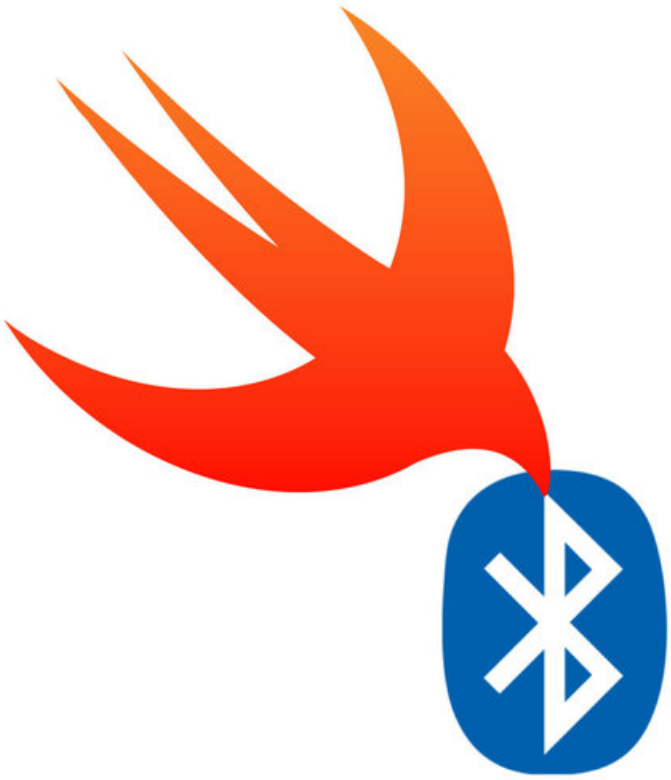
adafruit learning system

Create a Bluetooth LE App for iOS

Created by Trevor Beaton



Last updated on 2017-11-14 07:36:39 PM UTC

Guide Contents

Overview

Looking to learn how **Bluetooth Low Energy** works in iOS & **Swift 3**? Well look no further, we've got what you've been waiting for! This guide covers the basics on making your own Bluetooth Low Energy (BLE) app using the **Core Bluetooth Framework**.

You'll need a basic understanding of Swift and iOS development, but no prior experience with Bluetooth Low Energy is required. The example code can be applied to both iPhone and iPad users running the current version of IOS.

In this tutorial, I'll be using the Adafruit Feather 32u4 Bluefruit LE to send and receive data.

Before you begin, know that the **Simulator** in Xcode isn't capable of using Bluetooth LE services. You'll need to follow along using your iPhone or iPad running the most current version of iOS **(Version 10+).**

Since you'll need to test your app using an iOS device, you'll also need an **Apple Developer membership** (please note that **free membership enrollment** is fine for following along with this guide)**.** If you need help getting started as an iOS developer, check out our guide covering how to enroll to the **Apple Developer program** and gain membership:

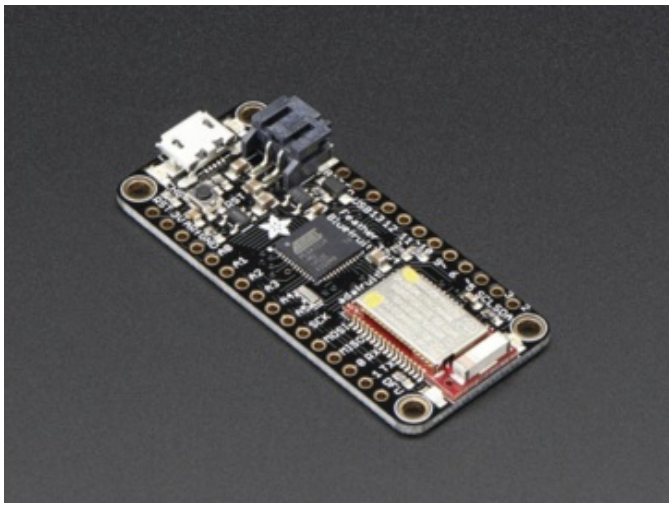
- Introduction to iOS Development

Before we start...

- Make sure your **Xcode IDE** is up-to-date to **version 8.2.1 or newer.**
- While in Xcode make sure the **development target** is **10.2. or higher.**
- You can download the **Arduino IDE** from the main website here if you haven't already.

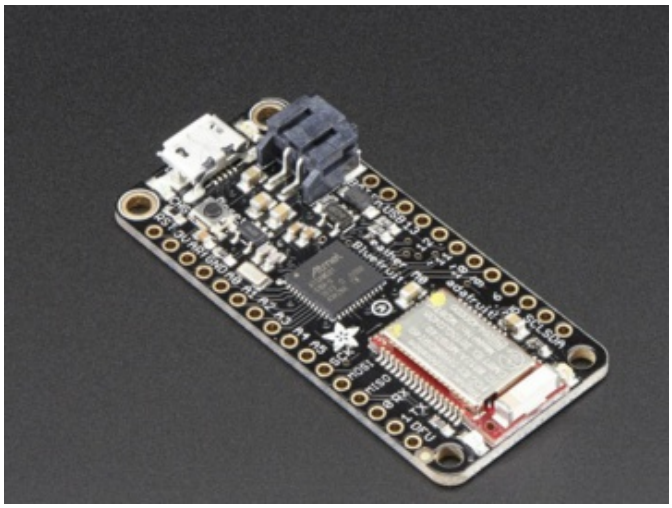
Parts Required

You don't need much to start this project, but you'll need these:



This is the **Adafruit Feather 32u4 Bluefruit** - our take on an 'all-in-one' Arduino-compatible + Bluetooth Low Energy with built in USB and battery charging. Its an Adafruit Feather 32u4 with a BTLE module, ready to rock!

Or



This is the **Adafruit Feather M0 with a Bluefruit module**. I didn't use this board for this project, but you can follow along with this as well.

&

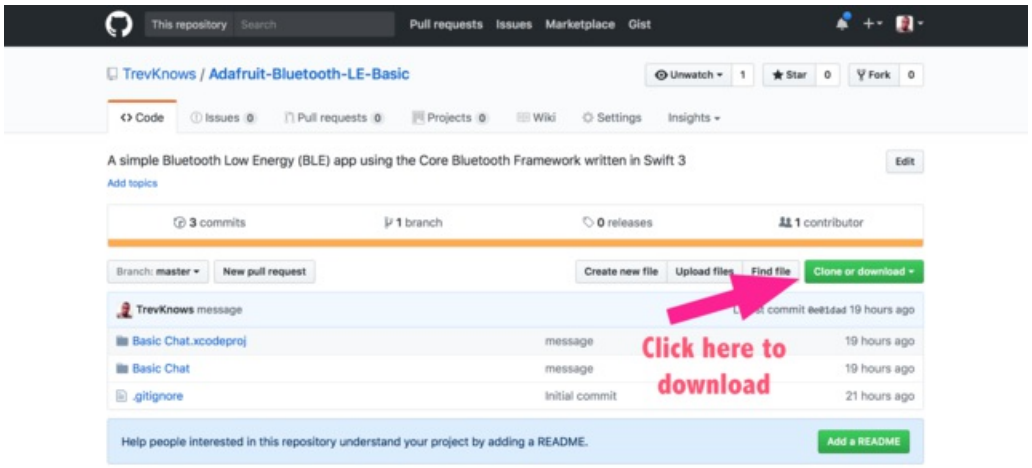
This here is your standard A to **micro-B USB cable**, for USB 1.1 or 2.0. You'll need this to connect your Adafruit Feather to your laptop.

You'll also need a **Mac** running the **Xcode** IDE and an **Apple Developer** account (check out the iOS Development Guide for more info on how to do this)

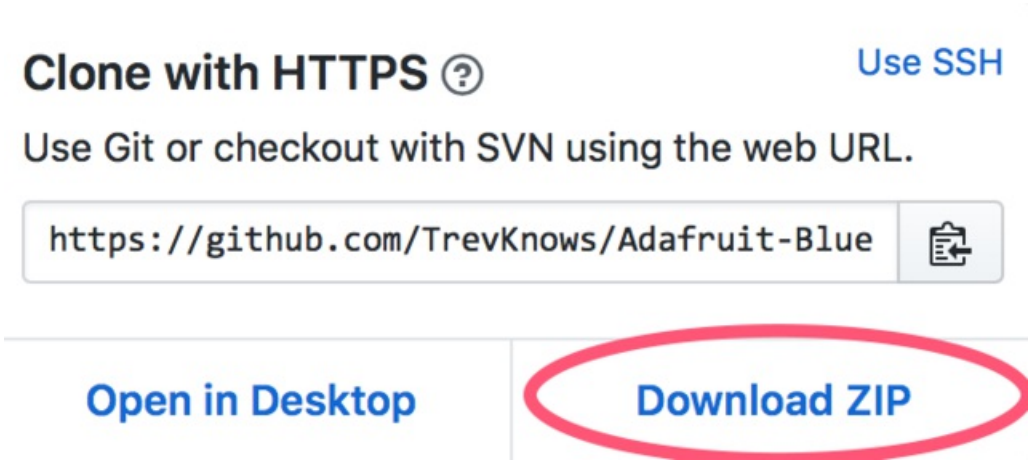
Download project from Github

To best visualize how things work, you'll want to follow along with the app provided in this guide. The **Basic Chat** app sends and receives data using the **Feather Bluefruit's UART** connection. You can learn more about UARTs here.

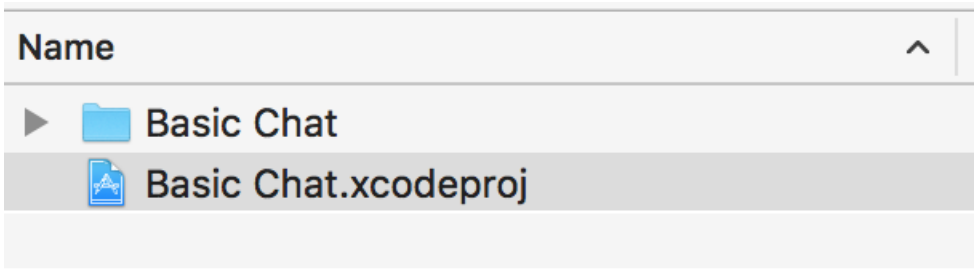
FIrst, go to the download page for the project. Once on the page, click on the **"Clone or download"** button.



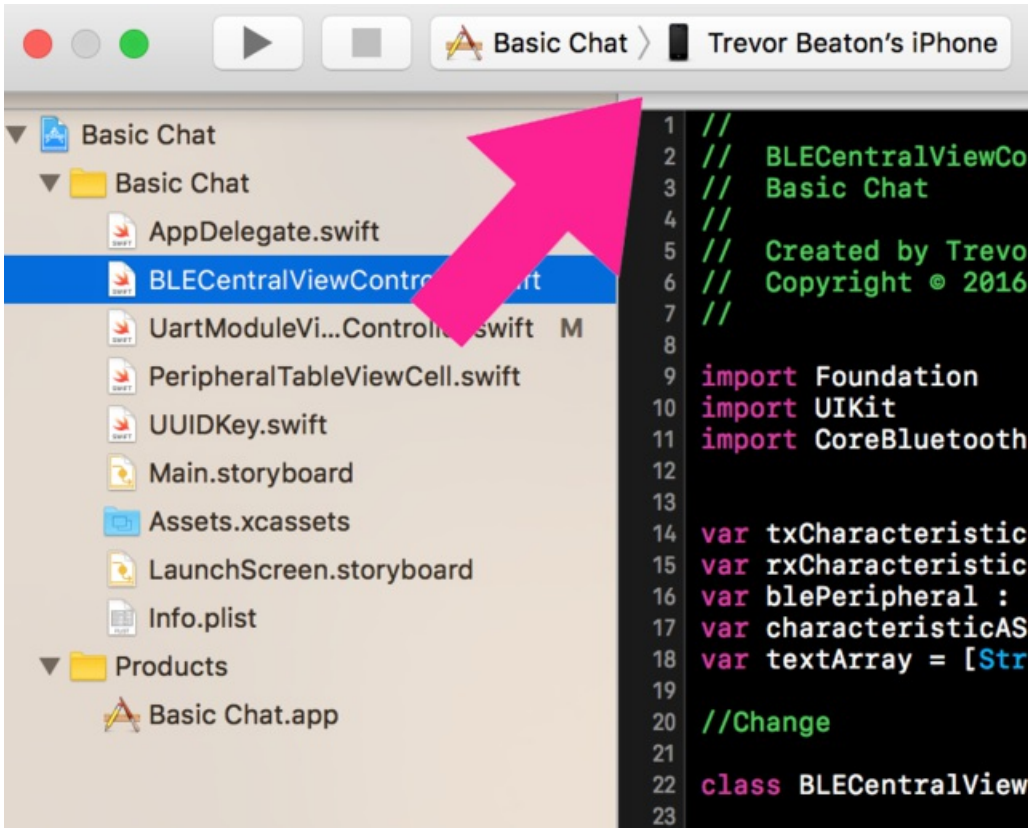
A small window will pop up beneath the **"Clone or Download"** button. Click the **Download Zip** button in the pop-up.



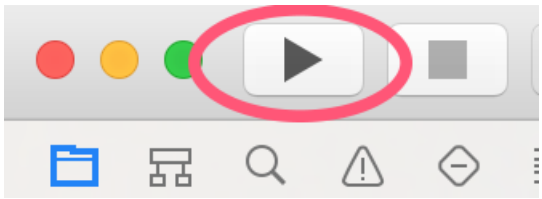
Once the file is downloaded, click on the **Basic Chat.xcodeproj** app and it will open up the project in **Xcode**.



Once we have the project open, select your iOS device's **scheme** in **Xcode**.



Now just press the **Run** button to test the app on your iOS device.



If all goes well, the **Basic Chat** app will run on your iOS device and you'll be able to use the app as a reference while you explore the guide.

Make sure your Bluetooth is enabled while using Basic Chat

These are the steps we'll make during the development process:

- Create an **instance** of a CBCentralManager.
- Once Bluetooth LE is powered on, we'll start scanning for **CBPeripherals** that hold services we are interested in.
- Once we've found matching peripherals, we'll display them in a table view and allow the user to choose a peripheral to connect to.
- Once connected, we'll look for the CBServices we're interested on the CBPeripheral.
- Once we find and connect to the desired CBService, we'll start looking for it's CBCharacteristics then subscribe to the one we are interested in.

- FInally, we'll send and receive data with the device using the CBCharacteristics found.

Before we begin, let's get a basic understanding of how some the app works.

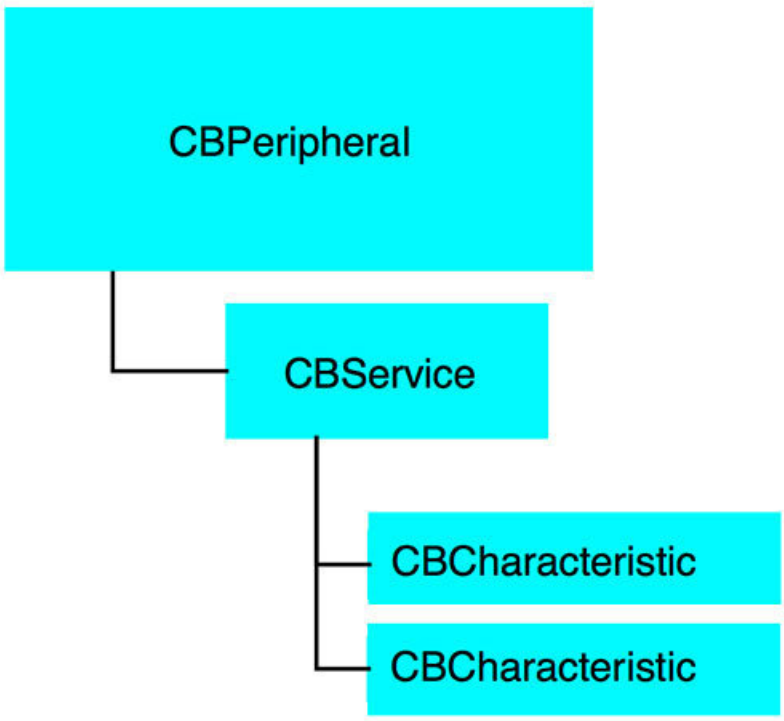
Understanding CBCentralManager and Peripheral

The **CBCentralManager** plays the most important role in this application, it represents the local device. Think of it as the "key player" in this operation. The CBCentralManager is responsible for scanning and connecting to peripherals.

If the CBCentralManager is looked at as the key player, then the **CBPeripheral** can be looked at as the "supporting player".

A CBPeripheral holds services (each defined as a **CBService**), and each CBService holds **CBCharacteristics**.

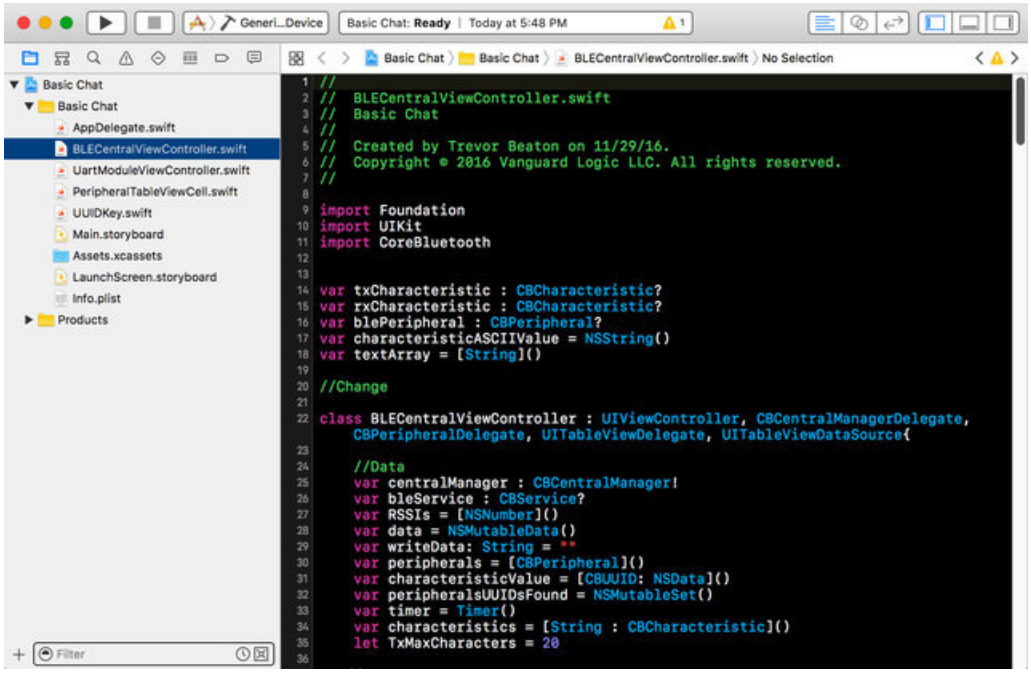
The following diagram shows the hierarchy of the CBPeripheral:



OK - Let's start looking at how the app works!

Getting Started

In Xcode, take a look at the project's file hierarchy in the left hand side of the window. Select the file named "**BLECentralViewController.swift**" and check out its contents in the main window. This file handles all of the BLE functions we'll be focussing on in this guide. Let's take a look at some of the code ...



At the top of the file, you'll notice the following line of code:

```
import CoreBluetooth
```

Importing the **Core Bluetooth framework** gives you the ability to communicate with BLE devices. With this you'll be able to make your own digital thermostat, game controller, heart rate monitor, and much more.

Next, notice that this object conforms to the **CBCentralManagerDelegate** protocol in order to discover, scan, and connect to peripheral devices in your area. In addition, it also conforms to the **CBPeripheralDelegate** protocol to analyze and explore peripheral properties.

```
class BLECentralViewController : UIViewController, CBCentralManagerDelegate, CBPeripheralDelegate, UITabl
```

Likely the most important variable of the **BLECentralManager** is the **CBCentralManager** object.
The **CBCentralManager** object works behind the scenes and performs all the direct communication with CBPeripherals. The code below creates the **CBCentralManager** and assigns its **delegate** variable to be the **BLECentralManager.**

Next we'll look at scanning for peripherals ...

```
var centralManager : CBCentralManager!

//You'll want to add the line below into the your viewDidLoad
centralManager = CBCentralManager(delegate: self, queue: nil)
```

Scanning for Peripherals

Once we've created a central manager and the iOS device's Bluetooth is powered on, the CBCentralManager will automatically call the `centralManagerDidUpdateState` function on its delegate (which is the **BLECentralViewController** in our app)

```
func centralManagerDidUpdateState(_ central: CBCentralManager) {
        if central.state == CBManagerState.poweredOn {
            // We will just handle it the easy way here: if Bluetooth is on, proceed...start scan!
            print("Bluetooth Enabled")
            startScan()

        } else {
            //If Bluetooth is off, display a UI alert message saying "Bluetooth is not enable" and "Make
            print("Bluetooth Disabled- Make sure your Bluetooth is turned on")

            let alertVC = UIAlertController(title: "Bluetooth is not enabled", message: "Make sure that y
            let action = UIAlertAction(title: "ok", style: UIAlertActionStyle.default, handler: { (actio
                self.dismiss(animated: true, completion: nil)
            })
            alertVC.addAction(action)
            self.present(alertVC, animated: true, completion: nil)
        }
    }
```

The `centralManagerDidUpdateState` function lets the device know when bluetooth is enabled or disabled. We can use this function to respond to bluetooth being powered on or off. In our example code, we do this by comparing `central.state` with the desired value of `CBManagerState.poweredOn`.

If these states match, we call the `startScan` function to begin looking for BLE peripherals nearby. If they don't match, the app presents an alert view notifying the user that they need to turn Bluetooth on.

Scanning for Peripherals

Once the CBCentralManager is up and powered on, we call the `startScan()` function to look for peripherals are around us.

```
func startScan() {
        print("Now Scanning...")
        self.timer.invalidate()
        centralManager?.scanForPeripherals(withServices: [BLEService_UUID] , options: [CBCentralManagerS
        Timer.scheduledTimer(timeInterval: 17, target: self, selector: #selector(self.cancelScan), userI
  }
```

The `startScan()` function calls *`scanForPeripherals(withServices)`* and creates a timer which stops scanning after 17 seconds by calling `self.cancelScan()`.

This `Timer` object is not required but allows for a reasonable amount of time to discover all nearby peripherals.

Using CBUUIDs

```
centralManager?.scanForPeripherals(withServices: [BLEService_UUID] , options: nil)
```

Notice that when we called *scanForPeripherals(withServices)*, we included [BLEService_UUID] to specify what services we're looking for. The BLEService_UUID variable (defined in the **UUIDKey.swift** file) is a **CBUUID** object. CBUUIDs are unique identifiers used throughout CoreBluetooth and are essential to BLE communication. Each peripheral, service, and characteristic has its own CBUUID.

By specifying what service we're looking for, we've told the **CBCentralManager** to ignore all peripherals which don't advertise that specific service, and return a list of **only** the peripherals which do offer that service.

Discovering Peripherals

Now that we've started scanning, what happens when we discover a peripheral?

Every time a peripheral is discovered, the **CBCentralManager** will notify us by calling the centralManager(_:didDiscover:advertisementData:rssi:) function on its delegate.

This function provides the following information about the newly discovered peripheral:

- The **CBCentralManager** providing the update.
- The discovered peripheral as a **CBPeripheral** object
- A Dictionary containing the peripheral's advertisement data.
- The current received signal strength indicator (**RSSI**) of the peripheral, in decibels.

```
func centralManager(_ central: CBCentralManager, didDiscover peripheral: CBPeripheral,advertisementData:
        stopScan()
   self.peripherals.append(peripheral)
        self.RSSIs.append(RSSI)
        peripheral.delegate = self
        peripheral.discoverServices([BLEService_UUID])
        self.baseTableView.reloadData()
        if blePeripheral == nil {
            print("We found a new pheripheral devices with services")
            print("Peripheral name: \(peripheral.name)")
            print("***********************************")
            print ("Advertisement Data : \(advertisementData)")
            blePeripheral = peripheral
        }
   }
```

In our implementation of this function we perform the following actions:

1. Stop scanning for peripherals (for our example app, we're only interested in the first one which appears)
2. Add the newly discovered peripheral to an array of peripherals.
3. Add the new peripheral's **RSSI** to an array of RSSIs.
4. Set the peripheral's **delegate** to self (**BLECentralViewController**)
5. Tell the Central Manager to discover more details about the peripheral's service
6. Reload the table view which uses our peripherals array as a data source
7. Set the blePeripheral variable to the new peripheral and print relevant information in the debug window

Next up - we'll actually connect to that peripheral.

Connecting to a Peripheral

Once the user taps a row in the found devices table view, the *connectToDevice()* function is called.

```
func connectToDevice () {
   centralManager?.connect(blePeripheral!, options: nil)
}
```

The *connectToDevice()* function calls *centralManager?.connect* to establishes a local connection to the desired peripheral.

Once the connection is made, the central manager calls the *centralManager(_:didConnect)* delegate function to provide incoming information about the newly connected peripheral.

```
func centralManager(_ central: CBCentralManager, didConnect peripheral: CBPeripheral) {
        print("******************************")
        print("Connection complete")
        print("Peripheral info: \(blePeripheral)")

        //Stop Scan- We don't need to scan once we've connected to a peripheral. We got what we came for.
        centralManager?.stopScan()
        print("Scan Stopped")

        //Erase data that we might have
        data.length = 0

        //Discovery callback
        peripheral.delegate = self
        //Only look for services that matches transmit uuid
        peripheral.discoverServices([BLEService_UUID])
}
```

Within this function, we perform a couple of actions:

- Display the selected peripheral's info in the console
- Stop scanning
- Erase any data from any previous scans
- Set the peripheral's delegate
- Discover the peripheral's services

Ok - Now, we'll need to handle the possible incoming services from the peripheral.

Discovering Services

Once the peripheral's services are successfully discovered, the central manager will call the *didDiscoverServices()* delegate function. didDiscoverService() handles and filters services, so that we can use whichever service we are interested in right away.

```
func peripheral(_ peripheral: CBPeripheral, didDiscoverServices error: Error?) {
        print("*******************************************************")

        if ((error) != nil) {
            print("Error discovering services: \(error!.localizedDescription)")
            return
        }

        guard let services = peripheral.services else {
            return
        }
        //We need to discover the all characteristic
        for service in services {

            peripheral.discoverCharacteristics(nil, for: service)
        }
        print("Discovered Services: \(services)")
    }
```

First, we handle any possible errors returned by the central manager, then we request characteristics for each service returned by calling `discoverCharacteristics(_:)`

Discovering Characteristics

Now that we've called the `discoverCharacteristics(_:)` function, the central manager will call the `didDiscoverCharacteristicsFor()` delegate function and provide the discovered characteristics of the specified service.

```
    func peripheral(_ peripheral: CBPeripheral, didDiscoverCharacteristicsFor service: CBService, error

        print("**************************************************")

        if ((error) != nil) {
            print("Error discovering services: \(error!.localizedDescription)")
            return
        }

        guard let characteristics = service.characteristics else {
            return
        }

        print("Found \(characteristics.count) characteristics!")

        for characteristic in characteristics {
            //looks for the right characteristic

            if characteristic.uuid.isEqual(BLE_Characteristic_uuid_Rx)  {
               rxCharacteristic = characteristic

                //Once found, subscribe to the this particular characteristic...
                peripheral.setNotifyValue(true, for: rxCharacteristic!)
                // We can return after calling CBPeripheral.setNotifyValue because CBPeripheralDelegate's
                // didUpdateNotificationStateForCharacteristic method will be called automatically
                peripheral.readValue(for: characteristic)
                print("Rx Characteristic: \(characteristic.uuid)")
            }
            if characteristic.uuid.isEqual(BLE_Characteristic_uuid_Tx){
                txCharacteristic = characteristic
                print("Tx Characteristic: \(characteristic.uuid)")
            }
        peripheral.discoverDescriptors(for: characteristic)
    }
}
```

A couple of things are happening in this function:

1. Handle errors and print characteristic info to the debug console
2. Look through the array of characteristics for a match to our desired UUIDs.
3. Perform any necessary actions for the matching characteristics
4. Discover descriptors for each characteristic

In this case, the specific UUIDs we're looking for are stored in the `BLE_Characteristic_uuid_Rx` and `BLE_Characteristic_uuid_Tx` variables.

When we find the RX characteristic, we subscribe to updates to its value by calling `setNotifyValue()` - this is how we receive data from the peripheral. Additionally, we read the current value from the characteristic and print its info to the console.

When we find the TX characteristic, we save a reference to it so we can write values to it later - this is how we send data to the peripheral.

Disconnecting from Peripheral

When the user taps the **Disconnect** button, the `disconnectFromDevice()` function is called.

```
func disconnectFromDevice () {
 if blePeripheral != nil {
 centralManager?.cancelPeripheralConnection(blePeripheral!)
    }
 }
```

The `disconnectFromDevice()` function first checks if there is a current peripheral set - if there is, it calls `cancelPeripheralConnection()` which cancels an active or pending local connection to the peripheral.

Now, we can proceed to reading and writing using CBCharacteristics.

Communication

Reading the Value of a Characteristic

Because we called `peripheral.setNotify()` in the previous step, whenever new data is available on the RX characteristic, two functions will be called on the peripheral's delegate – `peripheral(_:didUpdateNotificationStateFor:error:)` & `peripheral(_:didUpdateValueFor:error:)`.

In our implementation, `peripheral(_:didUpdateNotificationStateFor:error:)` simply prints info to the console. `peripheral(_:didUpdateValueFor:error:)` is where we read and handle the new incoming data from the peripheral.

```
func peripheral(_ peripheral: CBPeripheral, didUpdateValueFor characteristic: CBCharacteristic, error: Er

   if characteristic == rxCharacteristic {
     if let ASCIIstring = NSString(data: characteristic.value!, encoding: String.Encoding.utf8.rawValue)
        characteristicASCIIValue = ASCIIstring
        print("Value Recieved: \((characteristicASCIIValue as String))")
        NotificationCenter.default.post(name:NSNotification.Name(rawValue: "Notify"), object: nil)
 }
  }
}
```

In `didUpdateValueFor()`, we check to make sure the characteristic is the RX characteristic. If it is, we read the value and convert it to an ASCII string. We print the converted ASCII value to the console and we post a notification which can be used to update UI elements with the newly received data.

Writing to a Characteristic

Before we can write data to our external peripheral, we need to know how we want to write that data. There are two types of CBCharacteristic write types that we need to be aware of. The CBCharacteristic write type can be either `.withResponse` or `.withoutResponse`.

The `.withResponse` property type gets a response from the peripheral to indicate whether the write was successful. The `.withoutResponse` doesn't send any response back from the peripheral.

To write to a characteristic we'll need to write a value with an instance **NSData** and we'll do that by calling `writeValue(for: , type: CBCharacteristicWriteType.withResponse)` method:

```
blePeripheral.writeValue(data!, for: txCharacteristic, type: CBCharacteristicWriteType.withResponse)
```

Now you'll be able to write to a characteristic with a writeable attribute. You can find this example code in the **UartModuleViewController**.

```
// Write functions
    func writeValue(data: String){
      let valueString = (data as NSString).data(using: String.Encoding.utf8.rawValue)
        if let blePeripheral = blePeripheral{
           if let txCharacteristic = txCharacteristic {
              blePeripheral.writeValue(valueString!, for: txCharacteristic, type: CBCharacteristicWriteTy
            }
        }
    }
```

In the `writeValue(data)` function above, we format the outgoing string as **NSData** and then check to see if the `bleperipheral` and `txCharacteristic` varaiables are set. If they are, we then call the the `writeValue()` function as explained previously.

Because we wrote the value with a type of `CBCharacteristicWriteType.withResponse`, we can be notified of the response by implementing the following function in the delegate:

```
func peripheral(_ peripheral: CBPeripheral, didWriteValueFor characteristic: CBCharacteristic, error: Err
    guard error == nil else {
        print("Error discovering services: error")
      return
    }
     print("Message sent")
}
```

In this case, we simply use the write response to print relevant info to the debug console.

Communicating with Arduino IDE

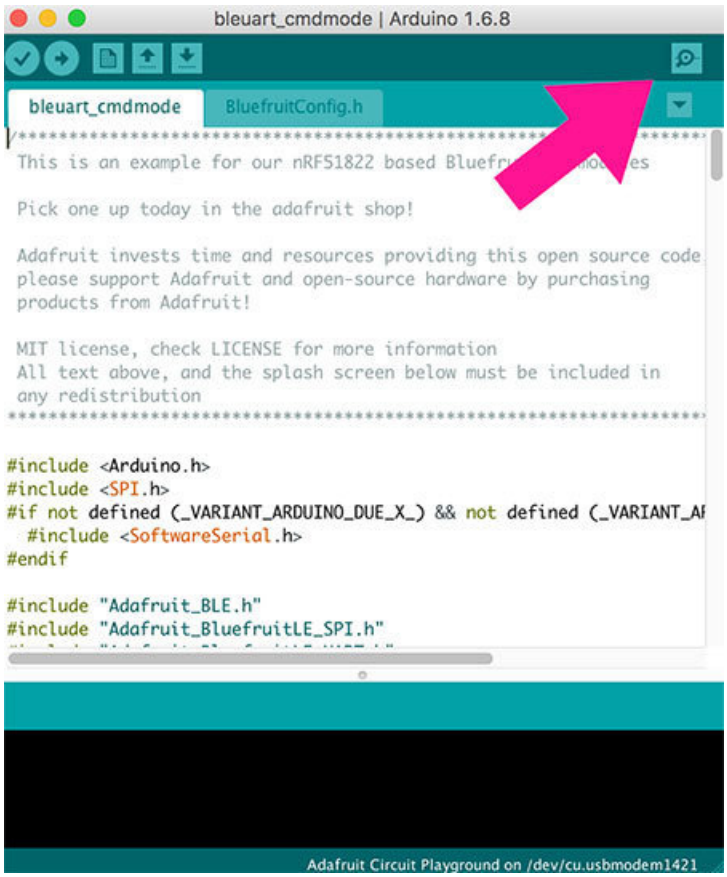
Sending Data

We can demonstrate the functionality of the Basic Chat app using an Adafruit Feather Bluefruit 32u4 board connected to a computer running the Arduino IDE. If you don't already have your Arduino IDE setup or if it's not recognizing your board, take a look at this learn guide on how to do so:

- Adafruit Feather 32u4 Bluefruit LE Learn Guide

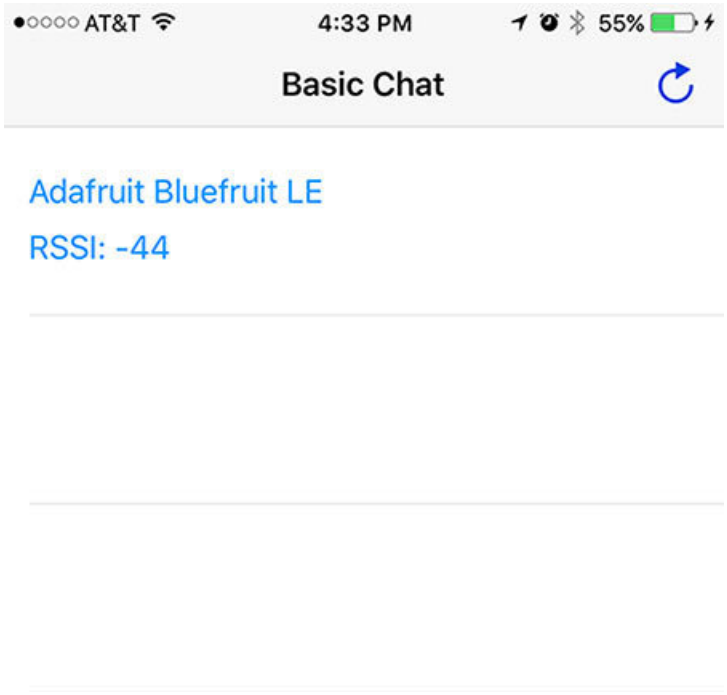
We'll use the serial monitor to read and write messages back and forth with the Basic Chat app.

Within the Arduino IDE, go to **File->Examples->Adafruit BluefruitLEnRF51** then select **bleart_cmdmode**. Once the sketch is opened, **upload** it to your Feather and click on the **Serial Monitor** button in the upper right of the Arduino IDE window. A new serial monitor window will open.

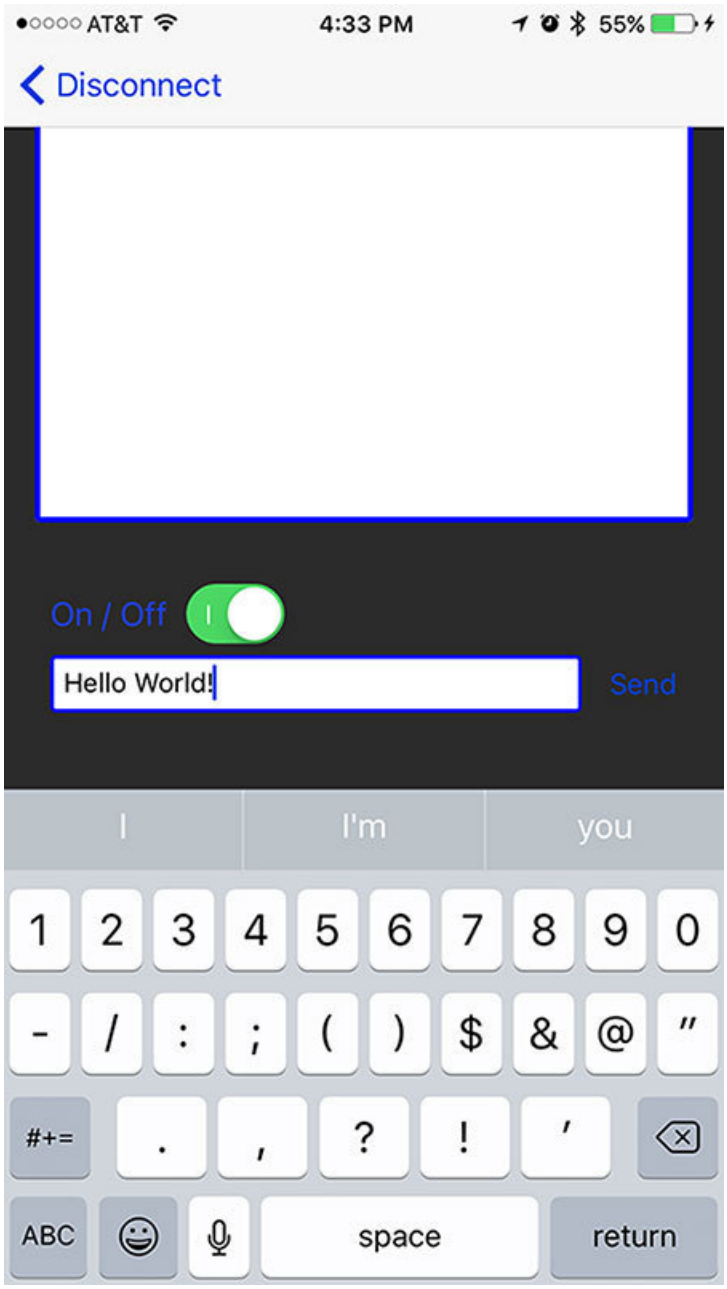


This new serial monitor window is where you'll be able to communicate between the **Feather** and the **Basic Chat** app.

Let's start by sending data to the Feather from the Basic Chat app. First, open up the Basic Chat app, then select **Adafruit Bluefruit LE**.

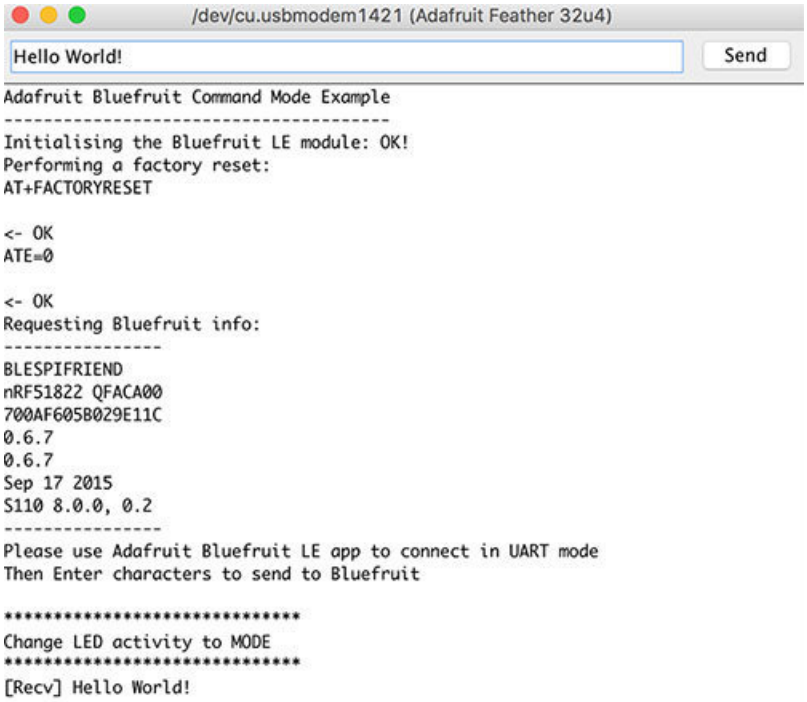


When we're in the **UARTModule** view we'll send a simple..."Hello World!" string in the text field below. Hit **Send** when you're done.

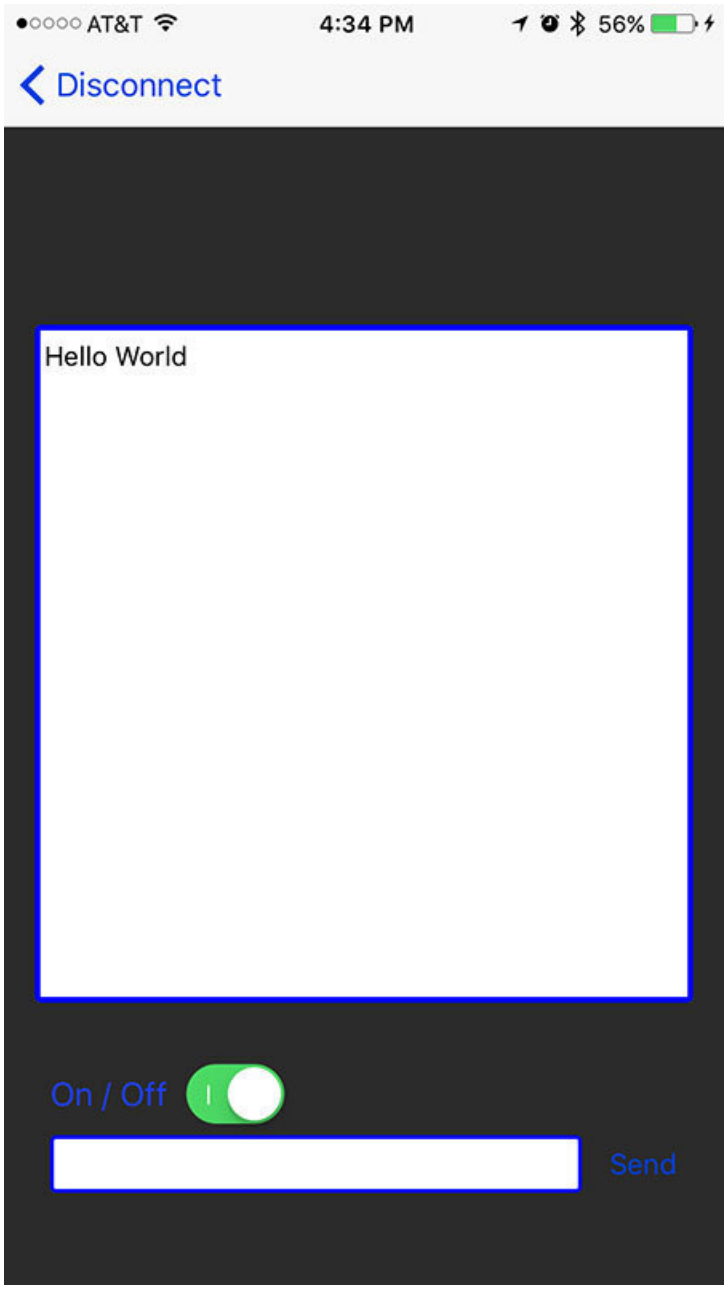

AT&T
4:33 PM
55%
Disconnect
On / Off
Hello World!
Send
I
I'm
you
1 2 3 4 5 6 7 8 9 0
- / : ; ( ) $ & @ "
#+= . , ? ! '
ABC
space
return

Now you've successfully sent the string "Hello World!" to the feather and it displayed in the Serial monitor view.

Now let's send "Hello World!" to the Basic Chat app. Type "Hello World!" in the text field of the Arduino IDE's serial monitor window and then hit **Send**.



Your "Hello World!" string should show up in the Basic Chat app's console.



Congrats!

You've successfully demonstrated the Basic Chat app's functionality and learned the basics of BLE communication using CoreBluetooth. With this knowledge and the Basic Chat app as a reference, you can start building your own BLE-capable apps for iOS!