Unity Bluetooth LE Plugin for iOS

Introduction

This plugin provides basic access to the CoreBluetooth API provided in iOS. You can use this plugin from Unity to create both Centrals and Peripherals. Some of the more advanced features of CoreBluetooth are not supported with this plugin.

For information about CoreBluetooth you will need to log into your Apple developer account and locate the CoreBluetooth documentation.

Version Changes

2.9

Added a new parameter to the scan method that allows you to get RSSI values even if the scanned device doesn't have manufacturer specific data.

2.8

Added SimpleTest. This is an example of connecting to an RFduino at startup using callbacks, states and a timer.

2.7

Fixed bugs when getting characteristics from multiple devices at the same time.
Added automatic lower casing of all UUID values for Android since that is what the OS does.

2.6

Added support for subscriptions to different characteristics to have their own callback handler.
Added a new example based on the Adafruit Bluefruit LE with the Nordic BLE Chip.

2.5

Added support for both Notification and Indication subscriptions in the Android library.

2.4

Added DontDestroyOnLoad to the Bluetooth Receiver Script so it can survive loading a new level.

2.3

Added SubscribeCharacteristicWithDeviceAddress

2.2

Added disconnect callback from connect
Added callback for the RSSI and manufacturer specific data in the advertising packet

2.1

Added an all new example that connects to RFduino and TI SensorTag devices

1.3

Added the characteristic UUID to the callback when receiving characteristic value updates

1.2

Fix a bug when passing null in the service UUIDs
Updated the documentation explaining adding CoreBluetooth Framework

Setup Guide

Setting up this plugin involves installing the package and writing scripts to interact with the plugin. Follow these steps:

1. Import the package into your Unity project
2. Create a a game object that you can attach a script to
3. Create a script that makes calls into the BluetoothHardwareInterface class using static methods for the Bluetooth functionality you require

Unity 5

Unity version 5 adds some new features and requirements for iOS. The output from Unity for iOS does not support non-ARC built code. When you try to build the project in xCode after building the project from Unity 5 you will get errors about ARC.

To fix this you must select the UnityBluetoothLE.mm file in the plugins/ios folder. Once you select it you can enter a value in Compiler flags. Put -fno-objc-arc. Another new feature of Unity 5 is that you can now specify additional frameworks that are needed by the plugin. You do this by opening the Rarely used frameworks section and checking the CoreBluetooth framework.

Example Code

This package includes a sample scene with a simple GUI exposing the most common functions that you will need. The example is located under the Example folder.

The example scene just includes the Main Camera object that is created in a new project. The TestScript sample script has been attached to the main camera.

The TestScript builds a simple GUI. When each button is clicked some very basic code is executed to demonstrate using the basic features of this plugin.

The buttons on the left side of the screen are used to connect to Bluetooth LE peripherals that are near the device. This is known as being a Bluetooth LE Central.

The buttons on the right side of the screen are used to BE a Bluetooth LE Peripheral.

If the example project is built and deployed to 2 iOS devices the buttons on the right on one device can be used to become a peripheral and the buttons on the left of the second device can be used to connect to the first device.

All output showing the device states and connections and data transfer is sent to the debug output. In order to see this you must have an active debug session with Xcode or a console session using the Organizer tool in Xcode on a Mac.

There is also a second example called BluetoothLETest. This example can connect to RFduino and TI SensorTag devices.

First select whether you want to have the app perform the functions of a central or peripheral. Next start scanning. Any BLE devices will show up in the list. Nothing will happen if you select a device that is not an RFduino or TI SensorTag.

If you choose an RFduino or TI SensorTag device you will get a page specific to those devices.

The RFduino page shows an image of an RFduino device. If you select button1 on the RFduino that button on the image will have a green circle on it.

If you hit the LED button on the page the LED should show up on the RFduino device.

The TI SensorTag just shows temperature. It is left up to the user to implement other sensors on the TI SensorTag.

If you choose the app to be a peripheral then it will simulate an RFduino device. You can connect to it using another iOS or Android device running this app. The peripheral implementation acts just like the RFduino with the LEDButton code in it.

Included at the bottom of the CentralRFduinoScript.cs file is the source to the code that can be run on the RFduino hardware. It is the same code that is included in the RFduino Examples that come with the RFduino device installation.

The third example is SimpleTest. This example shows how to connect to an RFduino at startup and subscribe to the button press. It requires that you use the same RFduino code as the larger example.

SimpleTest shows how to use a basic state machine and timeout value to make sure operations don’t overlap. This is a very important aspect of BLE.

If you start and operation before the previous operation is finished then it cancels the first operation.

This is why you can't call the interface methods one after another in code. You must wait for the callback to an operation to be called before you initiate the next operation.

Plugin Layout

The plugin has several parts.

Objective-C Code

The Plugins/iOS folder contains the Objective-C implementation. The full source code is contained in these files.

C# Scripts

The Plugins folder contains the C# scripts. These scripts provide the Unity – Plugin interface and some helper methods. The BluetoothHardwareInterface is a class that contains static methods to make calls into the Objective-C code. The BluetoothDeviceScript is used to receive messages passed back to Unity from the Obejctive-C code.

Example

The example code is detailed in the section above.

Editor Preprocessor

The Editor folder contains a preprocessor script that adds some required values to the plist.info file in the final Xcode project file.

CoreBluetooth Framework

In order to use Bluetooth Low Energy in an iOS app you must link your app with the CoreBluetooth framework library. This is done in Xcode and not in Unity. After you have built your app in Unity and opened the project in Xcode follow these steps to add the CoreBluetooth framework library:

1. Show the Project Navigator (upper left of tool panel the folder icon).
2. Select the Unity-iPhone Project at the top of the list. This will show the settings panel in the middle of the screen which has: General, Capabilities, Info, Build Settings, Build Phases and Build Rules.
3. Select the Build Phases
4. If it is not open already, open the Link Binary With Libraries section.
5. If CoreBluetooth is not in the list then click on the + symbol at the bottom of the list.
6. From the list that appears select the CoreBluetooth Framework and click on the Add button.

Every time you Replace the iOS project or build it from scratch you will need to go through these steps.

Support

For email support you can email support@shatalmic.com

Notes

As mentioned above all operations are asynchronous. This means you can't start a new operation until you have received the callback from the previous operation. If you do you will probably cancel the previous operation.

You can use timeouts and hope that you are done with the previous operation, but this is not as deterministic.

It is most reliable to scan for peripherals and store the address of the peripheral that you find during the scan and use that device address in all further API calls for that device.

Android is case sensitive. Keep this in mind when you are comparing UUID values. There are some methods in the plugin that convert everything to lower case. The examples also show doing this.

Some of the examples have a helper method called FullUUID. This method will take a 16 bit UUID or a 128 bit UUID. If it is 16 bit, then it is folded into the BLE Specification standard UUID to make it 128 bits.

Some of the examples also have a helper method called IsEqual. This method helps you compare 2 UUID values even if they are 16 bit, 128 bit, upper or lower case. It is recommended that you use these 2 helper methods in your apps.

API Reference

Initialization Errors

When Initialize is called there are several errors that can occur. You will receive the error text as the parameter to the errorAction callback. Here is a list of those errors:

Bluetooth LE Not Supported
Bluetooth LE Not Authorized
Bluetooth LE Powered Off
Bluetooth LE State Unknown

CoreBluetooth Enumeriations

```
public enum CBCharacteristicProperties
public  enum CBAttributePermissions
```

BluetoothHardwareInterface Methods

```
public static void Log (string message)
```

Log the string message to the Xcode console window

```
public static BluetoothDeviceScript Initialize (bool asCentral, bool asPeripheral, Action action, Action<string> errorAction)
```

Initialize the Bluetooth system as either a central, peripheral or both.
When completed the action callback will be executed.
If there is an error the errorAction callback will be executed.

```
public static void DeInitialize (Action action)
```

DeInitialize the Bluetooth system.
When completed the action callback will be executed.

```
public static void FinishDeInitialize ()
```

This method is automatically called by the BluetoothDeviceScript when it has been notified by the Objective-C code that everything else has been deinitialized.

```
public static void PauseMessages (bool isPaused)
```

This method notifies the bluetooth system that the app is going to be paused or unpaused. While the app is paused incoming Bluetooth messages are cached in an array. When the app unpauses all cached Bluetooth messages are sent to the app.

```
public static void ScanForPeripheralsWithServices (string[] serviceUUIDs, Action<string, string> action, Action<string, string, int, byte[]> actionAdvertisingInfo = null, rssiOnly = false)
```

This method puts the device into a scan mode looking for any peripherals that support the service UUIDs in the serviceUUIDs parameter array. If serviceUUIDs is NULL all Bluetooth LE peripherals will be discovered. As devices are discovered the action callback will be called with the ID and name of the peripheral.

The default value for the actionAdvertisingInfo callback is null for backwards compatibility. If you supply a callback for this parameter it will be called each time advertising data is received from a device. You will receive the ID and address of the device, the RSSI and the manufacturer specific data from the advertising packet.

The rssiOnly parameter will allow scanned devices that don't have manufacturer specific data to still send the RSSI value. The reason this defaults to false is for backwards compatibility.

```
public static void RetrieveListOfPeripheralsWithServices (string[] serviceUUIDs, Action<string, string> action)
```

This method will retrieve a list of all currently connected peripherals with the UUIDs listed in the serviceUUIDs parameter. If serviceUUIDs is NULL all Bluetooth LE peripherals will be discovered. As devices are discovered the action callback will be called with the ID and name of the peripheral.

```
public static void StopScan ()
```

This method stops the scanning mode initiated using the ScanForPeripheralsWithServices method call.

```
public static void ConnectToPeripheral (string name, Action<string> connectAction, Action<string, string> serviceAction, Action<string, string, string> characteristicAction, Action<string> disconnectAction)
```

This method attempts to connect to the named peripheral. If the connection is successful the connectAction will be called with the name of the peripheral connected to. Once connected the serviceAction is called for each service the peripheral supports. Each service is enumerated and the characteristics supported by each service are indicated by calling the characteristicAction callback.

The default value for the disconnectAction is null for backwards compatibility. If you supply a callback for this parameter it will be called whenever the connected device disconnects. Keep in mind that if you also supply a callback for the DisconnectPeripheral command below both callbacks will be called.

```
public static void DisconnectPeripheral (string name, Action<string> action)
```

This method will disconnect a peripheral by name. When the disconnection is complete the action callback is called with the ID of the peripheral.

```
public static void ReadCharacteristic (string name, string
service, string characteristic, Action<string,
byte[]> action)
```

This method will initiate a read of a characteristic using the name of the peripheral, the service and characteristic to be read. If the read is successful the action callback is called with the UUID of the characteristic and the raw bytes of the read.

```
public static void WriteCharacteristic (string name, string
 service, string characteristic, byte[] data, int length, b
ool withResponse, Action<string> action)
```

This method will initiate a write of a characteristic by the name of the peripheral and the service and characteristic to be written. The value to write is a byte buffer with the length indicated in the data and length parameters. The withResponse parameter indicates when the user wants a response after the write is completed. If a response is requested then the action callback is called with the message from the Bluetooth system on the result of the write operation.

```
public static void SubscribeCharacteristic (string name, st
ring service, string characteristic, Action<string> notific
ationAction, Action< string, byte[]> action)
```

This method will subscribe to a characteristic by peripheral name and the service and characteristic. The notificationAction callback is called when the notification occurs and the action callback is called whenever the characteristic value is updated by the peripheral. The first parameter is the characteristic UUID. The second is the raw data bytes that have been updated for the characteristic. This method is for backwards compatibility. A new method with the device address was added in version 2.3 (see below).

```
public static void SubscribeCharacteristicWithDeviceAddress
 (string name, string service, string characteristic, Actio
n<string, string> notificationAction, Action<string,
string, byte[]> action)
```

This method will subscribe to a characteristic by peripheral name and the service and characteristic. The notificationAction callback is called when the notification occurs and the action callback is called

whenever the characteristic value is updated by the peripheral. The first parameter is the device address. The second parameter is the characteristic UUID. The third is the raw data bytes that have been updated for the characteristic.

public static void UnSubscribeCharacteristic (string name, string service, string characteristic, Action<string> action)
This method is unsubscribe from a characteristic by name, service and characteristic. When complete the action callback is called.

public static void PeripheralName (string newName)
This method sets the peripheral name to use when advertising as a peripheral.

public static void CreateService (string uuid, bool primary, Action<string> action)
This method will create a service using the uuid parameter. It will be marked as primary or not by the primary parameter. When complete the action callback will be called. Note that the uuid can be a full UUID or a16 bit UUID. If the 16 bit UUID is used it will be converted into a full UUID according to the Bluetooth LE specification on using 16 bit UUID values.

public static void RemoveService (string uuid)
This method will remove a previously created peripheral service.

public static void RemoveServices ()
This method will remove all previously created peripheral services.

public static void CreateCharacteristic (string uuid, CBCharacteristicProperties properties, CBAttributePermissions permissions, byte[] data, int length, Action<string, byte[]> action)
This method will create a characteristic using the uuid parameter. It can have any of the CoreBluetooth properties and permissions contained in the enums. Data and length are the initial value to set the characteristic to. Action is a callback that will receive notification whenever a characteristic has been written from a central. Note that the uuid can be

a full UUID or a16 bit UUID. If the 16 bit UUID is used it will be converted into a full UUID according to the Bluetooth LE specification on using 16 bit UUID values.

```
public static void RemoveCharacteristic (string uuid)
```

This method will remove a previously created peripheral characteristic.

```
public static void RemoveCharacteristics ()
```

This method will remove all previously created peripheral characteristic.

```
public static void StartAdvertising (Action action)
```

This method will start advertising as a Bluetooth LE peripheral using the services and characteristics created above.

```
public static void StopAdvertising (Action action)
```

This method will stop advertising as a Bluetooth LE peripheral.

```
public static void UpdateCharacteristicValue (string uuid,
byte[] data, int length)
```

This method will update a characteristic's value. If a central has subscribed for notifications then it will receive a notification and value update.