Bluetooth Development for Communication with Epoc+ Devices.

Version 1
Author: Captain Smiley

General Bluetooth Development

Before we begin there are a few disclaimers that need to be stated:

1. This code was developed using Mac OS X 10.x for iOS mainly using Swift 3 (although the code below does work with version 4). Bluetooth code in objective-C can be found on Apple's website.

2. The development is for Bluetooth Low Energy (BTLE) communication.

3. Before this project, I had never done any bluetooth development. So, the code represents the learning I did on-the-fly. Therefore, much of the general bluetooth core setup code is

based on the examples provided by Apple[i] and Adafruit[ii]. I would strongly recommend looking over their code as the basis for the bluetooth framework if you are developing for OS X or iOS.

Ok, on to the good stuff… The overall architecture of bluetooth communication is based on a client-server model. The "server" is known as the `CentralManager`. To begin, we need to scan for the headset itself. This requires a unique identifier so that the system can identify the correct device (and service, etc.) Below is a list of the unique identifers (UUIDs) we use:

```
//  UUID of the Epoc+ headset
let DEVICE_NAME_UUID = "81072F40-9F3D-11E3-A9DC-
     0002A5D5C51B"

//  UUID of the main data stream with ID 0x10
let TRANSFER_DATA_UUID = "81072F41-9F3D-11E3-A9DC-
     0002A5D5C51B"

//  UUID of the gyro/other? data stream with ID 0x20
let TRANSFER_MEMS_UUID = "81072F42-9F3D-11E3-A9DC-
     0002A5D5C51B"
…
let BLEDeviceName_UUID = CBUUID(string: DEVICE_NAME_UUID)
let BLE_Data_uuid_Rx = CBUUID(string:
     TRANSFER_DATA_UUID)
let BLE_Mems_uuid_Rx = CBUUID(string:
     TRANSFER_MEMS_UUID)
```

[i] Apple's development site: https://developer.apple.com/bluetooth/
[ii] Adafruit's Create a Bluetooth LE App for iOS: https://learn.adafruit.com/crack-the-code?view=all#overview

You could guess these UUIDs by dumping the strings of the EDK, but I wanted to know for sure what they were and how they were used so I used a bluetooth sniffer[iii]. The top three variables are the actual UUIDs of the hardware and specific services we will use. The bottom three variables are the Apple API converted UUIDs the function can actually use. The one we need to use to discover the headset itself is `BLEDeviceName_UUID`.

```
fileprivate var centralManager: CBCentralManager?
…
centralManager = CBCentralManager(delegate: self)
centralManager?.scanForPeripherals(withServices:
     [BLEDeviceName_UUID], options:
     [CBCentralManagerScanOptionAllowDuplicatesKey :
     NSNumber(value: true as Bool)])
```

Once we have have identified the device we can connect to it using
`centralManager?.connect(peripheral, options: nil)`

Again, refer to the Apple and Adafruit examples to see how these functions are laid out.

After connecting, we need to discover the services associated with the device. Each service has specific characteristics that layout what the service does. Again, we will need the UUIDs for the characteristics we are interested in – secifically the raw data associated with the specific eeg channel recordings and/or mems data.

[iii] Adafruit's Bluefrut LE Sniffer: https://www.adafruit.com/product/2269

```
let characteristics = service.characteristics
for characteristic in characteristics
{
     peripheral.readValue(for: characteristic)
     if((characteristic.uuid.isEqual(BLE_Data_uuid_Rx)))
     {
          var tmpInt = NSInteger(0x0001)
          let data = NSData(bytes: &tmpInt, length: 2)
          peripheral.setNotifyValue(true, for: characteristic)
          peripheral.writeValue(data as Data,
               for: descriptor.characteristic,
               type: CBCharacteristicWriteType.withResponse)
     }
     …
}
```

To get the data to stream to us, we need to tell the device to start streaming and to notify us every time the values are changed/updated. For this, we need to set the notification value to `true` and send `0x0001` to the peripheral's service.

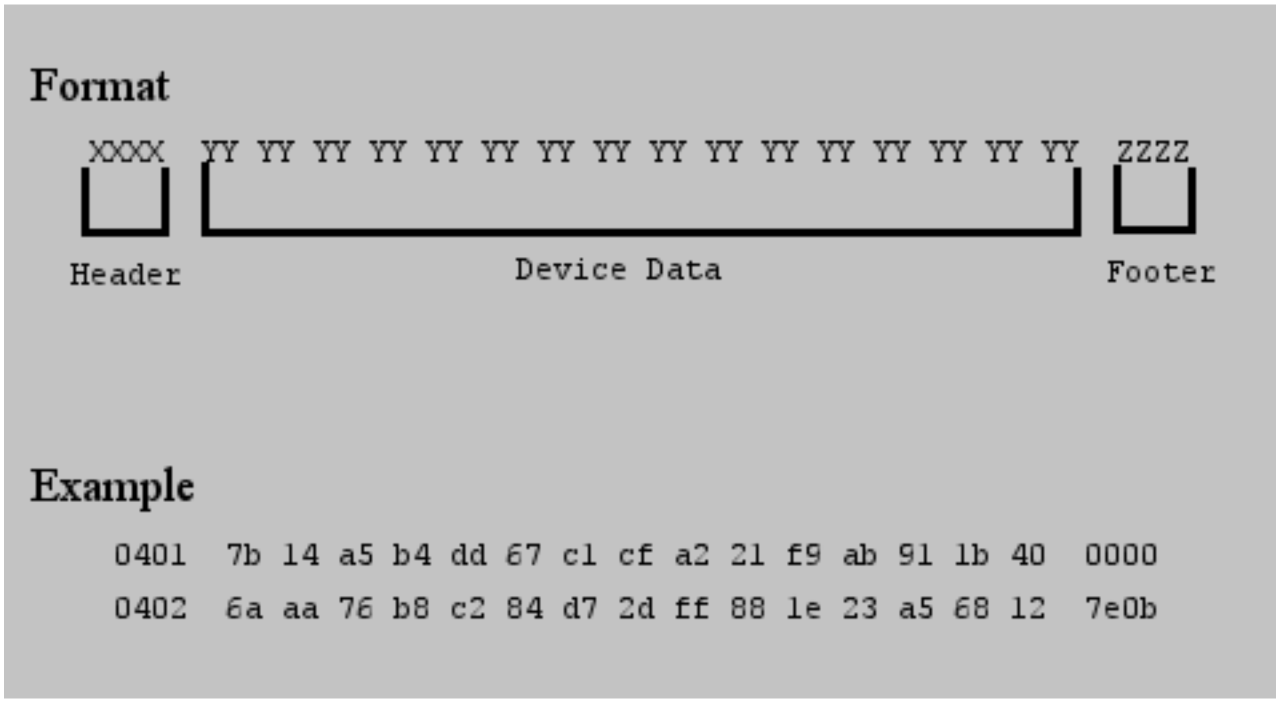
After the notification has been set, the `didUpdateValueFor` callback function will be called each time the device sends out a packet of data. To handle the data, we do the following:

```
fileprivate let data = NSMutableData()
...
data.setData(characteristic.value!)
```

The `characteristic.value` will contain the packet data which we assign it to the variable `data` so we can process the information.

Bluetooth Packet Format and Data Processing

The Bluetooth data stream packet format is different then that of the WiFi packet. In Bluetooth, the data is broken into two separate packets. Below is the format of a single data packet.



```
0401  7b 14 a5 b4 dd 67 c1 cf a2 21 f9 ab 91 1b 40  0000
0402  6a aa 76 b8 c2 84 d7 2d ff 88 1e 23 a5 68 12  7e0b
```

The packet begins with a header that consists of a counter and which half of the data is contained. In the example above, the packet counter is `04` and the `01` contains the first half (16 bytes) of the headset data while the `02` contains the second half (16 bytes) of the data. If your headset is set to sample at 128Hz, the packet counter will run from 00 to 7f. For our purposes of getting the raw values of the headset, we ignore the footer.

To process the device data, we first strip away the header and footer. The following will strip away bytes 00 and 01 as well as 18 and 19 leaving us with the 16 bytes of headset data for decryption.

```
let tmpData = data.subdata(with: NSMakeRange(2, 16))
```

We then pass the tmpData to our decryption function. I do the decryption in objectiv-C because the Emotiv EDK uses OpenSSL, so using a dirivitive of C/C++ made my life a bit easier. As there is good documentation of the various OpenSSL functions it is bit easer to decrypt the data (as well as find out the decryption key). However, to use objectiv-C in Swift code we must use a "bridge".

To do this, we start by creating a header file in our project name `<Project Name>-Bridging-Header.h`. Since we are only using the bridge for OpenSSL code, the contents of the header file will look as follows:

```
//
//  Use this file to import your target's public headers that
//  you would like to expose to Swift.
//

#import "MyOpenSSL.h"
#include <openssl/opensslv.h>
```

The `#import` line is the name of the header file that contains my code declarations and the `#include` is same include we would do in a objectiv-C/C/C++ app to use OpenSSL functions. The name of the function I use for processing our data is `decrypt_raw_packet`, so `MyOpenSSL.h` will look like this:

```
//
//  MyOpenSSL.h
//

#import <Foundation/Foundation.h>

@interface MyOpenSSL : NSObject

- (NSString *)decrypt_raw_packet : (NSData *)data;

@end
```

Now we can call decrypt_raw_packet from our Swift code. We will start by declaring a variable that contains the class we are importing and then we can call the function.

```
...
var crypto: MyOpenSSL = MyOpenSSL()

...

let tmpLine = crypto.decrypt_raw_packet((tmpData) as Data!)
```

Our imported OpenSSL code is contained an an object-C style name `MyOpenSSL.m` file. We use OpenSSL's ECB decryption method. The code is shown below.

```
//
//  MyOpenSSL.m
//

#import “MyOpenSSL.h”
#import <openssl/evp.h>
#import <openssl/aes.h>
#import <openssl/rand.h>


@implementation MyOpenSSL

#define HID_DATA_LEN     16
#define MULTIPLIER       0.5128205128205129

- (NSString *)decrypt_raw_packet:(NSData *)data
{
     AES_KEY key;
     unsigned char aes_[] = <your hex string key here>;
     unsigned char dec_out[((HID_DATA_LEN+AES_BLOCK_SIZE /
               AES_BLOCK_SIZE) * AES_BLOCK_SIZE)] = {0};
     unsigned char *cipher = (unsigned char *)
               malloc([data length]);

     memcpy(cipher, [data bytes], [data length]);

     AES_set_decrypt_key(aes_key, 128, &key);

     int c = 0;
     while (c < HID_DATA_LEN)
     {
          AES_ecb_encrypt(cipher+c, dec_out+c, &key,
               AES_DECRYPT);
          c += 16;
     }

     NSMutableString *strResult = [NSMutableString string];

     for (int i = 0; I < HID_DATA_LEN; i += 2)
     {
          int tmpVal = (dec_out[i+1] << 8) | dec_out[i];
          float rawVal = (tmpVal * MULTIPLIER) * 0.25;
          [strResult appendFormat:@”%lf”, rawVal];
     }

     return strResult;
}

@end
```

In the code example above `strResult` will contain the decrypted raw data float value as a string. The defined multiplier above was based on research of the iOS EDK. However, research done for the CyKit project may have a more accurate multiplier to use – therefor the values can be substitued with values found in that python code.

Conclusion

This document was created to assist development of Bluetooth Low Energy projects (especially for those developing for OS X and iOS). As time allows, I will release a version 2 that will include a tutorial for cross platform development using QT and possibly the Windows SDK. I hope this has been benifical – happy coding!