Bluetooth Development for Communication with Epoc+ Devices.

Version 2
Author: CaptainSmiley

Disclaimer

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

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

2. 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 Microsoft[i], Apple[ii] and Adafruit[iii]. I would strongly recommend looking over their code as the basis for the bluetooth framework if you are developing for the Windows or Apple platforms.

I welcome all feedback and/or inquires you might have. You can find me on the project's Discord server under the name CaptainSmiley.

---

[i] https://msdn.microsoft.com/en-us/library/windows/hardware/jj159880(v=vs.85).aspx
[ii] Apple's development site: https://developer.apple.com/bluetooth/
[iii] Adafruit's Create a Bluetooth LE App for iOS: https://learn.adafruit.com/crack-the-code?view=all#overview

Table of Contents

Chapter 1.

Bluetooth Packet Format and Data Processing

OK… now on to the good stuff. 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.

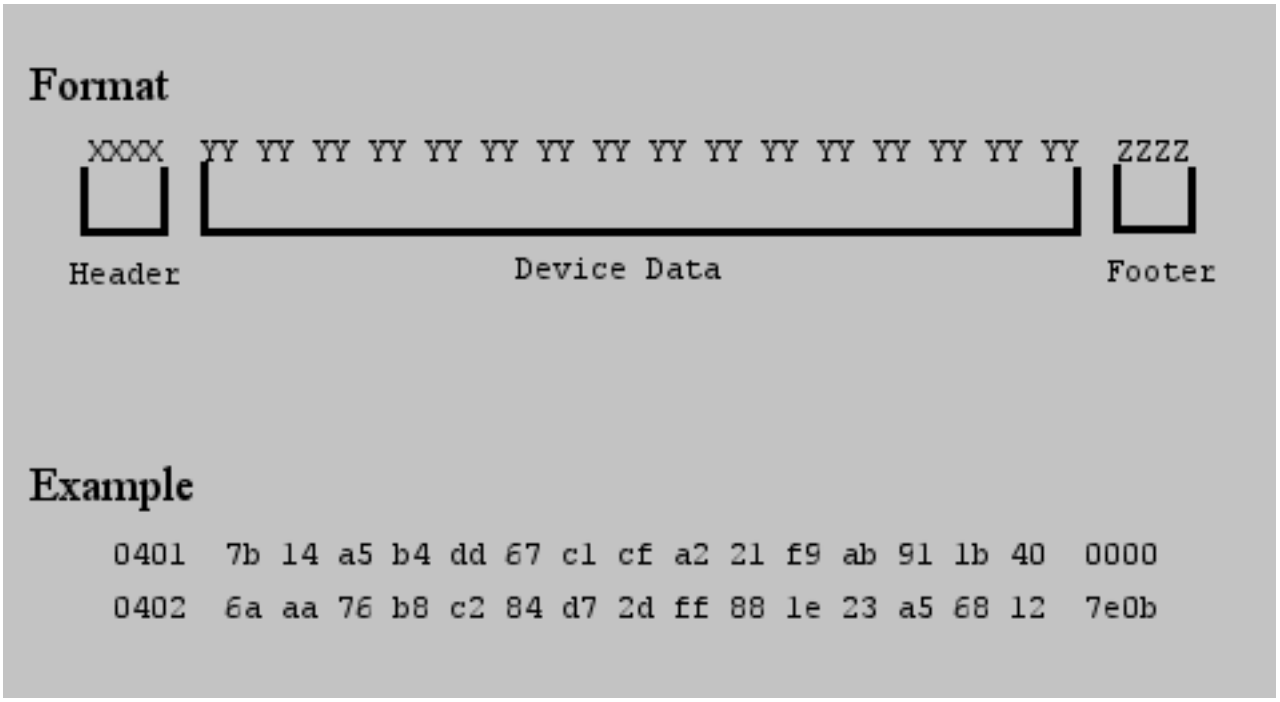

Figure 1.1

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))
```

Figure 1.2

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>
```

Figure 1.3

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
```

Figure 1.4

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!)
```

Figure 1.5

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
```

Figure 1.6

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.

Chapter 2.

Bluetooth LE Development for 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)
```

Figure 2.1

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[iv]. 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)])
```

Figure 2.2

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.

[iv] 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)
     }
     …
}
```

Figure 2.3

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!)
```

Figure 2.4

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

Chapter 3.

Bluetooth Low Energy Development for Windows

Starting in Windows 8, Microsoft introduced native support for Bluetooth Low Energy (LE) devices. These are devices such as mobile phones, headphones, iWatch, and the Emotiv EEG headsets. Low energy devices have a different protocol stack then that of regular bluetooth so, unless you develop your own stack driver, you cannot use normal bluetooth development to communicate with low energy devices. To understand the formal structure of how Windows handles Bluetooth LE devices, check out their introduction to bluetooth development[v].

In Windows, connecting to a device requires we get a handle to said device. To do this we must either scan for the device or you can start by pairing the device to the operating system and we can then connect to it through the operating system itself. To make our lives easier, we will start by pairing the device to the operating system. On Windows 10, you can add a device by going to *Settings (right mouse clicking on the Windows icon on the bottom left of your screen) -> Devices -> and click on Add Bluetooth or other device*.

[v] http://msdn.microsoft.com/en-us/library/windows/hardware/jj159880(v=vs.85).aspx

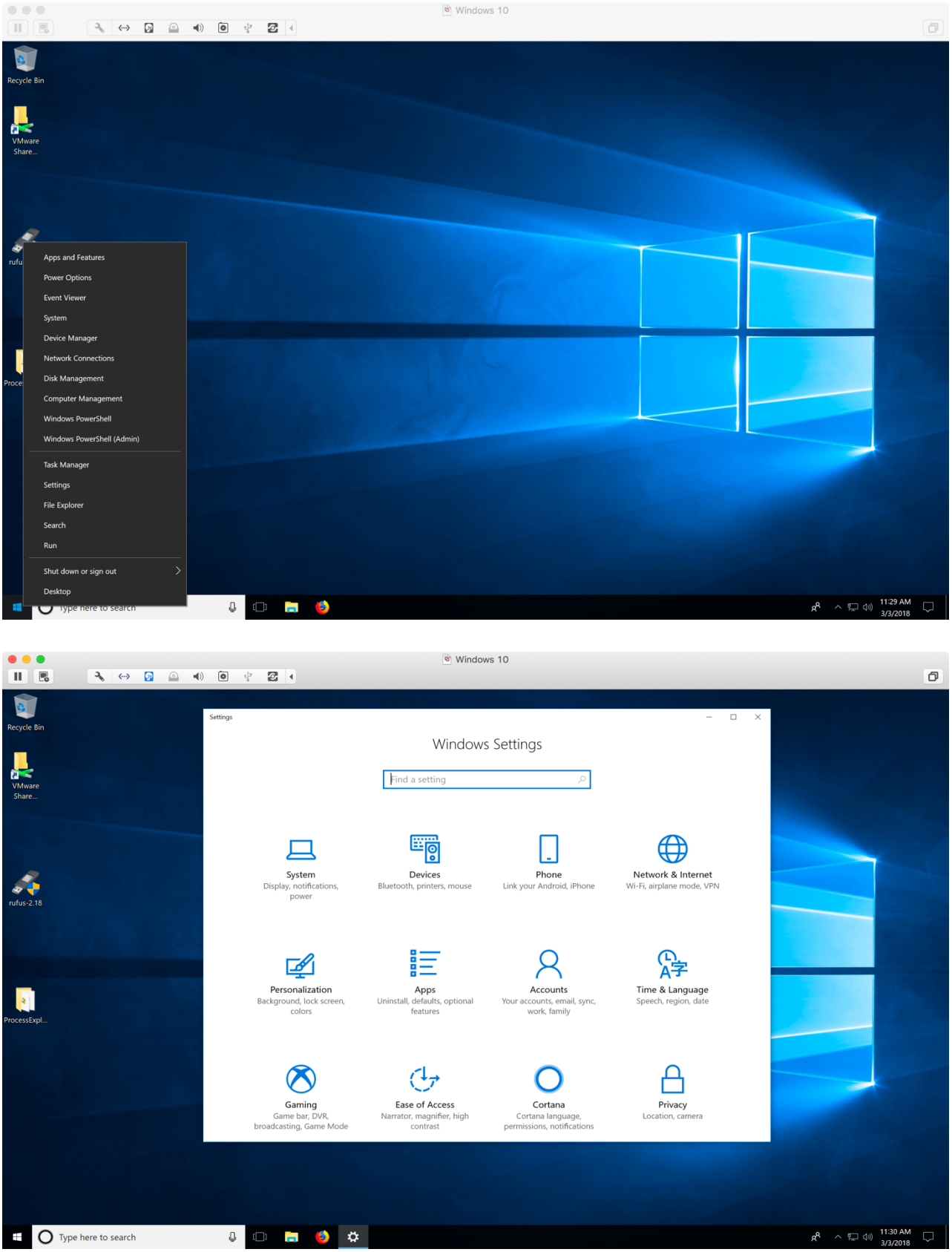
Windows 10
Recycle Bin
VMware Share...
rufu
Proce
Apps and Features
Power Options
Event Viewer
System
Device Manager
Network Connections
Disk Management
Computer Management
Windows PowerShell
Windows PowerShell (Admin)
Task Manager
Settings
File Explorer
Search
Run
Shut down or sign out
Desktop
Type here to search
11:29 AM
3/3/2018
Windows 10
Recycle Bin
VMware Share...
rufus-2.18
ProcessExpl...
Settings
Windows Settings
Find a setting
System
Display, notifications, power
Devices
Bluetooth, printers, mouse
Phone
Link your Android, iPhone
Network & Internet
Wi-Fi, airplane mode, VPN
Personalization
Background, lock screen, colors
Apps
Uninstall, defaults, optional features
Accounts
Your accounts, email, sync, work, family
Time & Language
Speech, region, date
Gaming
Game bar, DVR, broadcasting, Game Mode
Ease of Access
Narrator, magnifier, high contrast
Cortana
Cortana language, permissions, notifications
Privacy
Location, camera
Type here to search
11:30 AM
3/3/2018

Figure 3.1

Once you have paired the device, we can programatically connect to it (we will be using C++). We start by converting our device's UUID string to a GUID; then proceed to get a handle from the operating system that provices device information via the following process:

```
WCHAR *wchDevice = "{81072F40-9F3D-11E3-A9DC-0002A5D5C51B}";
GUID BluetoothInterfaceGUID;
...

CLSIDFromString(wchDevice, BluetoothInterfaceGUID);
...

HDEVINFO hDi = SetupDiGetClassDevs(&BluetoothInterfaceGUID,
                                   NULL,
                                   NULL,
                                   DIGCF_PRESENT | DIGCF_DEVICEINTERFACE);
```

Figure 3.2

hDI can now be used to enumerate through devices looking for the proper device interface. Since we know the specific GUID we are looking for we can quickly request the interface details such as the device's path so that we can get a handle to the specific device. The following process is used to accomplish this:

```
SP_DEVICE_INTERFACE_DATA did;
SP_DEVINFO_DATA dd;
DWORD i = 0;
DWORD size = 0;

did.cbSize  = sizeof(SP_DEVICE_INTERFACE_DATA);
dd.cbSize   = sizeof(SP_DEVINFO_DATA);
...

SetupDiEnumDeviceInterface(hDI, NULL, &BluetoothInterfaceGUID, i, &did);
...

SetupDiGetDeviceInterfaceDetail(hDI, &did, NULL, i, &size, 0);

PSP_DEVICE_INTERFACE_DETAIL_DATA pdIDD = (PSP_DEVICE_INTERFACE_DETAIL_DATA)
                                          GlobalAlloc(GPTR, size);

pdIDD->cbSize = sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);
SetupDiGetDeviceInterfaceDetail(hDI, &did, pdIDD, size, &size, &dd);
...

HANDLE hLEDevice = CreateFile(pdIDD->DevicePath, GENERIC_WRITE |
                              GENERIC_READ, FILE_SHARE_READ |
                              FILE_SHARE_WRITE, NULL, OPEN_EXISTING, 0,
                              NULL);
...

GlobalFree(pdIDD);
...

SetupDiDestroyDeviceInfoList(hDI);
```

Figure 3.3

The purpose of the code above is to obtain a handle to the headset device. To do this we need the system's path to the device. It will be similar to the following:

\\?\bthledevice#{81072f40-9f3d-11e3-a9dc-0002a5d5c51b}_ce08848cdee7#...#{81072f40-9f3d-11e3-a9dc-0002a5d5c51b}

Now that we have an open handle to the device, we can communicate with the device; this includes such things as getting the device's services, characteristics, and descriptors. Some good information on the concepts of services, characteristics, and descriptors can be found on the Bluetooth specification site[vi].

Windows provides straight forward functions to obtain the necessary information for proper GATT communication with the device. Based on the Bluetooth LE stack, we start by obtaining the services the device provides. Using the device's handle obtained above, we obtain the services as follows:

```
USHORT servicesBufferCount, numServices;
PBTH_LE_GATT_SERVICE pServiceBuffer;
...

HRESULT resSvc = BluetoothGATTGetServices(hLEDevice, 0, NULL,
                                          &serviceBufferCount,
                                          BLUETOOTH_GATT_FLAG_NONE);
pServiceBuffer = (PBTH_LE_GATT_SERVICE)malloc(sizeof(BTH_LE_GATT_SERVICE) *
                                              serviceBufferCount);

RtlZeroMemory(pServiceBuffer, sizeof(BTH_LE_GATT_SERVICE) *
              serviceBufferCount);

resSvc = BluetoothGATTGetServices(hLEDevice, serviceBufferCount,
                                  pServiceBuffer, &numServices,
                                  BLUETOOTH_GATT_FLAG_NONE);
```

Figure 3.4

Note the need to call `BluetoothGATTGetServices` twice. This is because we need to get the size of the buffer required to hold the list of services. The first call will get us the required size of the buffer; while in the second we actually specify the buffer count and the function will fill the service buffer as well as to provide the number of services that are in the service buffer.

[vi] https://blog.bluetooth.com/a-developers-guide-to-bluetooth

Now that we have a buffer that contains the services, we can get the characteristics for each service. The following code is used to obtain the characteristic data:

```
USHORT charBufferSize, numChars;
PBTH_LE_GATT_CHARACTERISTIC pCharBuffer = NULL;
...

HRESULT resCh = BluetoothGATTGetCharacteristics(hLEDevice, pServiceBuffer,
                                                0, NULL, &charBufferSize,
                                                BLUETOOTH_GATT_FLAG_NONE);

pCharBuffer = (PBTH_LE_GATT_CHARACTERISTIC)malloc(charBufferSize *
                                            sizeof(BTH_LE_GATT_SERVICE));

RtlZeroMemory(pCharBuffer, sizeof(BTH_LE_GATT_CHARACTERISTIC));

resSvc = BluetoothGATTGetCharacteristics(hLEDevice, pServiceBuffer,
                                         charBufferSize, pCharBuffer,
                                         &numChars,BLUETOOTH_GATT_FLAG_NONE);
```

Figure 3.5

As with the process of obtaining the device's services, we must also call `BluetoothGATTGetCharacteristics` twice. If the second call is successful (i.e. `resCh == S_OK`), the function will fill characteristics buffer (`pCharBuffer`) with the services characteristics and provide the number of characteristic via the `numChars` variable. `pCharBuffer` will contain the characteristics stored as a `BTH_LE_GATT_SERVICE` structure. This is an important structure because it will contain the UUIDs we are looking for to provide us the headsets raw mems and eeg data. The structure looks as follows:

```
typedef struct _BTH_LE_GATT_CHARACTERISTIC {
      USHORT      ServiceHandle;
      BTH_LE_UUID CharacteristicUuid;
      USHORT      AttributeHandle;
      USHORT      CharacteristicValueHandle;
      BOOLEAN     IsBroadcastable;
      BOOLEAN     IsReadable;
      BOOLEAN     IsWritable;
      BOOLEAN     IsWritableWithoutResponse;
      BOOLEAN     IsSignedWritable;
      BOOLEAN     IsNotifiable;
      BOOLEAN     IsIndicatable;
      BOOLEAN     HasExtendedProperties;
} BTH_LE_GATT_CHARACTERISTIC, *PBTH_LE_GATT_CHARACTERISTIC;
```

Figure 3.6

Of significant importance in this structure are items – `CharacteristicUuid` and `IsNotifiable`. The `CharacteristicUuid` variable will allow us to obtain the UUID that we can use to compare against the UUIDs for EEG data and mems. Recall from Chapter 2 that we have identified the unique UUIDs that will provide access to the data we are looking for. The `IsNotifiable` variable will tell us if the current characteristic will notify us if data values have been changed or updated – like we would see each time a raw data packet arrives from the headset.

To find the right characteristics that corrispond to the UUIDs we are looking for, we could do the following:

```
#define TRANSFER_DATA_UUID = L"81072F41-9F3D-11E3-A9DC-0002A5D5C51B"
#define TRANSFER_MEMS_UUID = L"81072F42-9F3D-11E3-A9DC-0002A5D5C51B"
// ... - elipse #1

PBTH_LE_GATT_CHARACTERISTIC currGattChar
USHORT descriptorBufferSize;
OLECHAR *pStr;
const WCHAR uuids[2] = {TRANSFER_DATA_UUID, TRANSFER_MEMS_UUID};

for(int ii = 0; ii < charBufferSize; ii++)
{
      currGattChar = &pCharBuffer[ii];
      StringFromCLSID(currGattChar->CharacteristicUuid.Value.LongUuid, &pStr);
      int z = 0;

      //  loop through all the UUIDs we are interested in
      while(uuids[z] != NULL)
      {
            if((_wcsicmp(pStr, uuids[z])) == 0)
            {
                  //  now get the descriptors for the characteristic
                  HRESULT resGD = BluetoothGATTGetDescriptors(hLEDevice,
                                                currGattChar, 0, NULL,
                                                BLUETOOTH_GATT_FLAG_NONE);

                  PBTH_LE_GATT_DESCRIPTOR pDescriptorBuffer;
                  if(descriptorBufferSize > 0)
                  {
                        pDescriptorBuffer = (PBTH_LE_GATT_DESCRIPTOR)malloc(
                                          descriptorBufferSize *
                                          sizeof(BTH_LE_GATT_DESCRIPTOR));

                        RtlZeroMemory(pDescriptorBuffer, descriptorBufferSize);

                        USHORT numDescriptors;
                        resGD = BluetoothGATTGetDescriptors(hLEDevice,
                                                currGattChar,
                                                descriptorBufferSize,
                                                pDescriptorBuffer,
                                                &numDescriptors,
                                                BLUETOOTH_GATT_FLAG_NONE);
                        // ... - continue after elipese (#2)

                  }
                  // ... - continue after elipese (#3)

            }

            z++;
      }
      // ... - continue after elipese (#4)
}
```

Figure 3.7

Our loop through each of the caracteristics starts by obtaining a string version of the UUID. This will allow us to use an easy string comparison function (`StringFromCLSID`) to find the UUIDs

we are looking for. Once we have found the characteristic of interest we proceed to get all of the descriptors for that specific characteristic. So, in our example the first set of descriptors we would get corrispond to `TRANSFER_DATA_UUID`. Continuing with the example code from Figure 3.7, after elipese #2 we include the following code:

```
for(int k = 0; k < numDescriptors; k++)
{
      ...
      PBTH_LE_GATT_DESCRIPTOR currGattDescriptor = &pDescriptorBuffer[k];
      BTH_LE_GATT_DESCRIPTOR_VALUE newValue;

      RtlZeroMemory(&newValue, sizeof(newValue);
      newValue.DescriptorType = ClientCharacteristicConfiguration;
      newValue.ClientCharacteristicConfiguration.IsSubscribeToNotification =
            TRUE;

      HRESULT resSDV = BluetoothGATTSetDescriptorValue(hLEDevice,
                                                currGattDescriptor,
                                                &newValue,
                                                BLUETOOTH_GATT_FLAG_NONE);
}
```

Figure 3.8

In the code snippet above, we tell the system that we want to be notified any time the value of the characteristic's descriptor changes – that is, any time we receive new data packets from the headset. This is accomplished by calling the `BluetoothGATTSetDescriptorValue` function and passing it a `BTH_LE_GATT_DESCRIPTOR_VALUE` structure. Of interest, we set the value `IsSubscribeToNotification` in the structure to `TRUE`. Upon successful completion, the system will now notify us when data arrives from the headset.

When data arrives from the headset we want to be alerted so we can process the data. The final step in setting up Bluetooth LE communication is to register an *event handler* that will be called anytime the descriptor's value changes (i.e. any time a data packet arrives from the headset).

```
HRESULT resRE, resSCV;
BLUETOOTH_GATT_EVENT_HANDLE eHandle;
BTH_LE_GATT_EVENT_TYPE eType = CharacteristicValueChangedEvent;
BLUETOOTH_GATT_VALUE_CHANGED_EVENT_REGIATRATION eParamIn;

if(currGattChar->IsNotifiable)
{
      eParamIn.Characteristics[0] = *currGattChar;
      eParamIn.NumCharacteristics = 1;

      resRE = BluetoothGATTRegisterEvent(hLEDevice,
                        eType,
                        &eParamIn,
                        (PFNBLUETOOTH_GATT_EVENT_CALLBACK)ProcessEvent,
                        NULL,
                        &eHandle,
                        BLUETOOTH_GATT_FLAG_NONE);

      BTH_LE_GATT_CHARACTERISTIC_VALUE newValue;
      RtlZeroMemory(&newValue, (sizeof(newValue)));

      newValue.DataSize = sizeof(ULONG);
      newValue.Data[0] = 0x100;

      resSCV = BluetoothGATTSetCharacteristicValue(hLEDevice,
                              currGattChar,
                              &newValue,
                              NULL,
                              BLUETOOTH_GATT_FLAG_WRITE_WITHOUT_RESPONSE);

}
```

Figure 3.9

There are three important things to note about the above code. The first is the check if the current characteristic is *notifiable* or not. This is important because if the characteristic is not, then we will not receive any information. The second thing to observe about the code in Figure 3.9 is the callback function `ProcessEvent`. That is the function that will be use to process headset data. Figure 3.10 shows the code we use to process the EEG packets.

```
void CALLBACK ProcessEvent(BTH_LE_GATT_EVENT_TYPE eType,
                           PVOID eOutParam,
                           PVOID Context)
{
      if(eOutParam->CharacteristicValue->DataSize != 0)
      {
            printf("%s\n", eChangedValue->CharacteristicValue->CharacteristicValue);
      }
}
```

Figure 3.10

The third observation about the code in Figure 3.9 is the use of the function `BluetoothGATTSetCharacteristicValue`. The EEG headset needs to be alerted that the application is ready to receive data. We do this by sending a value of *0x100* – either

`TRANSFER_DATA_UUID` or `TRANSFER_MEMS_UUID` from Figure 3.7 – to the headset. Once this value has been set, data will begin to stream from the headset to your application.