Introduction to Core Bluetooth: Building a Heart Rate Monitor

Learn how to use Core Bluetooth on a real-world device!

The Core Bluetooth framework lets your iOS and Mac apps communicate with Bluetooth low energy devices (Bluetooth LE for short). Bluetooth LE devices include heart rate monitors, digital thermostats, and more.

The Core Bluetooth framework is an abstraction of the Bluetooth 4.0 specification and defines a set of easy-to-use protocols for communicating with Bluetooth LE devices.

In this tutorial, you'll learn about the key concepts of the Core Bluetooth framework and how to leverage the framework to discover, connect, and retrieve data from compatible devices. You'll use these skills by building a heart rate monitoring application that communicates with a Bluetooth heart monitor.

The heart rate monitor we use in this tutorial is the Polar H7 Bluetooth Smart Heart Rate Sensor. If you don't have one of these devices, you can still follow along with the tutorial, but you'll need to tweak the code for whatever Bluetooth device that you need to work with.

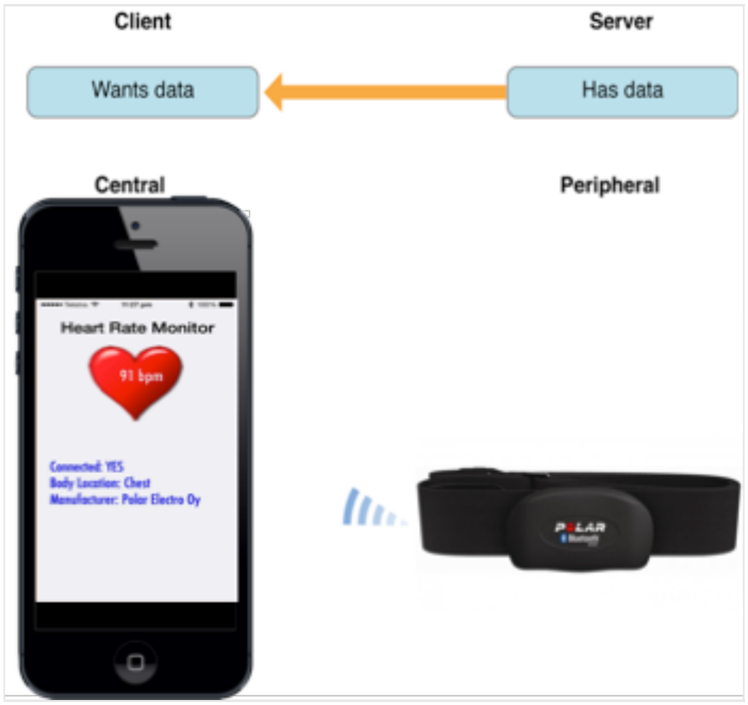
Allright, it's Bluetooth LE time!

Understanding Central and Peripheral Devices in Bluetooth

The two major players involved in all Bluetooth LE communication are known as the ***central*** and the ***peripheral***:

- A ***central*** is kinda like the "boss". It wants information from a bunch of its workers in order to accomplish a particular task.
- A ***peripheral*** is kinda like the "worker". It gathers and publishes data to that is consumed by other devices.

The following image illustrates this relationship:



In this scenario, an iOS device (the central) communicates with a Heart Rate Monitor (the peripheral) to retrieve and display heart rate information on the device in a user-friendly way.

How Centrals Communicate with Peripherals

Advertising is the primary way that peripherals make their presence known via Bluetooth LE.

In addition to advertising their existence, advertising packets can contain some data, such as the peripheral's name. It can also include some extra data related to what the peripheral collects. For example, in the case of a heart rate monitor, the packet also provides heartbeats per minute (BPM) data.

The job of a central is to scan for these advertising packets, identify any peripherals it finds relevant, and connect to individual devices for more information.

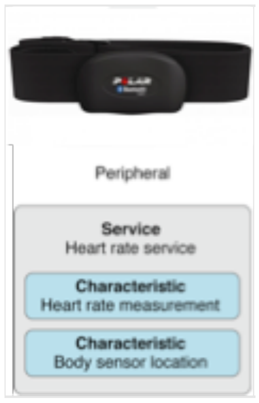
The Structure of Peripheral Data

Like I said, advertising packets are very small and cannot contain a great deal of information. So to get more, a central needs to connect to a peripheral to obtain all of the data available.

Once the central connects to a peripheral, it needs to choose the data it is interested in. In Bluetooth LE, data is organized into concepts called ***services*** and ***characteristics***:

- A ***service*** is a collection of data and associated behaviors describing a specific function or feature of a device. An example of a service is a heart rate monitor exposing heart rate data from the monitor's heart rate sensor. A device can have more than one service.
- A ***characteristic*** provides further details about a peripheral's service. For example, the heart rate service just described may contain a characteristic that describes the intended body location of the device's heart rate sensor and another characteristic that transmits heart rate measurement data. A service can have more than one characteristic.

The diagram below further describes the relationship between services and characteristics:



Once a central has established a connection to a peripheral, it's free to discover the full range of services and characteristics of the peripheral and to read or write the characteristic values of the available services.

CBPeripheral, CBService and CBCharacteristic

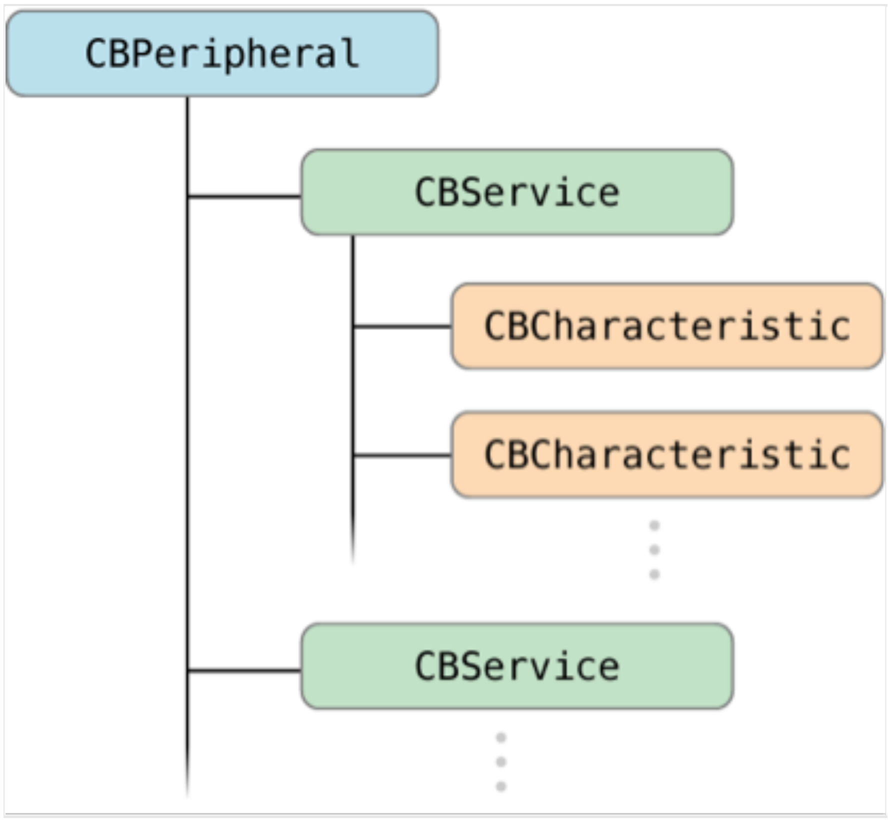
In the CoreBluetooth framework, a peripheral is represented by the ***CBPeripheral*** object, while the services relating to a specific peripheral are represented by the ***CBService*** object.

The characteristics of a peripheral's service are represented by ***CBCharacteristic*** objects which are defined as attribute types containing a single logical value.

Note: If you're interested in learning more about the Bluetooth standard, feel free to check out http://developer.bluetooth.org where you can find a list of the standardized services and characteristics of each.

Centrals are represented by the ***CBCentralManager*** object and are used to manage discovered or connected peripheral devices.

The following diagram illustrates the basic structure of a peripheral's services and characteristics object hierarchy:



Each service and characteristic you create must be identified by a unique identifier, or UUID. UUIDs can be 16- or 128-bit values, but if you are building your client-server (central-peripheral) application, then you'll need to create your own 128-bit UUIDs. You'll also need to make sure the UUIDs don't collide with other potential services in close proximity to your device.

In the next section, you'll learn how to reference the ***CoreBluetooth*** and ***QuartzCore*** header files and conform to their delegates so you can communicate and retrieve information about the heart rate monitor.

Getting Started

Enough background, time to code!

Start by downloading the starter project for this tutorial. This is a bare bones View-based application that simply has an image you need added.

Next, you need to import CoreBluetooth and QuartzCore into your project. To do this, open ***HRMViewController.h*** and add the following lines:

```
@import CoreBluetooth;
@import QuartzCore;
```

This uses the new @import keyword introduced in Xcode 5. To learn more about this, check out our What's New in Objective-C and Foundation in iOS 7 tutorial.

Next, add some #defines for the service UUIDs for the Polar H7 heart rate monitor you'll be working with in this tutorial. These come from the services section of the Bluetooth specification:

```
#define POLARH7_HRM_DEVICE_INFO_SERVICE_UUID @"180A"
#define POLARH7_HRM_HEART_RATE_SERVICE_UUID @"180D"
```

You are interested in two services here: one for the device info, and one for the heart rate service.

Similarly, add some defines for the characteristics you're interested in. These come from the characteristics section of the Bluetooth specification:

```
#define POLARH7_HRM_MEASUREMENT_CHARACTERISTIC_UUID @"2A37"
#define POLARH7_HRM_BODY_LOCATION_CHARACTERISTIC_UUID @"2A38"
#define POLARH7_HRM_MANUFACTURER_NAME_CHARACTERISTIC_UUID @"2A29"
```

Here you list out the three characteristics from the heart rate service that you are interested in.

Note that if you are working with a different type of device, you can add the appropriate services/characteristics for your device here according to your device and the specification.

Conforming to the Delegate

HRMViewController needs to implement the **`CBCentralManagerDelegate`** protocol to allow the delegate to monitor the discovery, connectivity, and retrieval of peripheral devices. It also needs to implement the **`CBPeripheralDelegate`** protocol so it can monitor the discovery, exploration, and interaction of a remote peripheral's services and properties.

Open ***HRMViewController.h*** and update the interface declaration as follows:

```
@interface HRMViewController : UIViewController
<CBCentralManagerDelegate, CBPeripheralDelegate>
```

Next, add the following properties between the **`@interface`** and **`@end`** lines to represent your CentralManager and your peripheral device:

```
@property (nonatomic, strong) CBCentralManager *centralManager;
@property (nonatomic, strong) CBPeripheral     *polarH7HRMPeripheral;
```

Next let's add some stub implementations for the delegate methods. Switch to ***HRMViewController.m*** and add this code:

```
#pragma mark - CBCentralManagerDelegate

// method called whenever you have successfully connected to the BLE
peripheral
- (void)centralManager:(CBCentralManager *)central
didConnectPeripheral:(CBPeripheral *)peripheral
{
}

// CBCentralManagerDelegate - This is called with the CBPeripheral
class as its main input parameter. This contains most of the
information there is to know about a BLE peripheral.
- (void)centralManager:(CBCentralManager *)central
didDiscoverPeripheral:(CBPeripheral *)peripheral advertisementData:
(NSDictionary *)advertisementData RSSI:(NSNumber *)RSSI
{
}

// method called whenever the device state changes.
- (void)centralManagerDidUpdateState:(CBCentralManager *)central
{
}
```

That takes care of the CentralManager — now add the following empty stubs for your delegate callback methods for your **`CBPeripheralDelegate`** protocol:

```
#pragma mark - CBPeripheralDelegate

// CBPeripheralDelegate - Invoked when you discover the peripheral's
available services.
- (void)peripheral:(CBPeripheral *)peripheral didDiscoverServices:
(NSError *)error
{
}

// Invoked when you discover the characteristics of a specified
service.
- (void)peripheral:(CBPeripheral *)peripheral
didDiscoverCharacteristicsForService:(CBService *)service error:
(NSError *)error
{
}

// Invoked when you retrieve a specified characteristic's value, or
when the peripheral device notifies your app that the characteristic's
value has changed.
- (void)peripheral:(CBPeripheral *)peripheral
didUpdateValueForCharacteristic:(CBCharacteristic *)characteristic
error:(NSError *)error
{
}
```

Finally, create the following empty stubs for retrieving **`CBCharacteristic`** information for Heart Rate, Manufacturer Name, and Body Location:

```
#pragma mark - CBCharacteristic helpers

// Instance method to get the heart rate BPM information
- (void) getHeartBPMData:(CBCharacteristic *)characteristic error:
(NSError *)error
{
}
// Instance method to get the manufacturer name of the device
```

```
- (void) getManufacturerName:(CBCharacteristic *)characteristic
{
}
// Instance method to get the body location of the device
- (void) getBodyLocation:(CBCharacteristic *)characteristic
{
}
// Helper method to perform a heartbeat animation
- (void)doHeartBeat {
}
```

Note: As you progress through this tutorial, you'll flesh out these methods as required.

Creating the User Interface

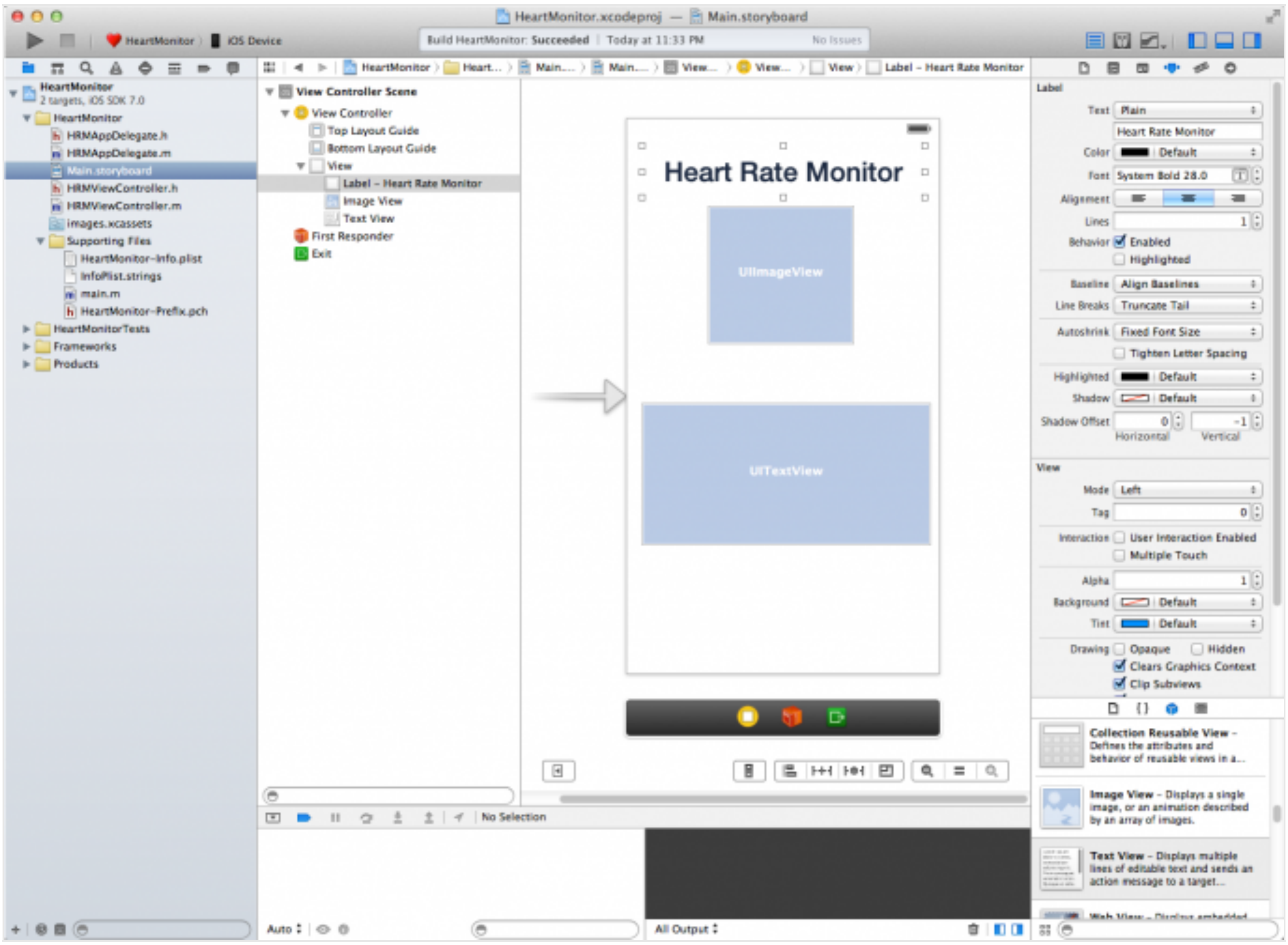
Let's create a rough user interface to display the data from the heart rate monitor.

Open ***HRMViewController.h*** and add the following properties between the **`@interface`** and **`@end`** lines, underneath the other property methods you just created:

```
// Properties for your Object controls
@property (nonatomic, strong) IBOutlet UIImageView *heartImage;
@property (nonatomic, strong) IBOutlet UITextView  *deviceInfo;

// Properties to hold data characteristics for the peripheral device
@property (nonatomic, strong) NSString   *connected;
@property (nonatomic, strong) NSString   *bodyData;
@property (nonatomic, strong) NSString   *manufacturer;
@property (nonatomic, strong) NSString   *polarH7DeviceData;
@property (assign) uint16_t heartRate;

// Properties to handle storing the BPM and heart beat
@property (nonatomic, strong) UILabel    *heartRateBPM;
@property (nonatomic, retain) NSTimer    *pulseTimer;

// Instance method to get the heart rate BPM information
- (void) getHeartBPMData:(CBCharacteristic *)characteristic error:
(NSError *)error;

// Instance methods to grab device Manufacturer Name, Body Location
- (void) getManufacturerName:(CBCharacteristic *)characteristic;
- (void) getBodyLocation:(CBCharacteristic *)characteristic;

// Instance method to perform heart beat animations
- (void) doHeartBeat;
```

Next open ***Main.storyboard***. Look for the right sidebar in your Xcode window; if you don't see one, you might need to use the rightmost button under the View section on the toolbar at the top to make the right hand sidebar visible.

Select and drag a Label, an ImageView, and a TextView control from the ***Object Library*** main view and position them roughly as shown below:



Label
Text: Plain
Heart Rate Monitor
Color: Default
Font: System Bold 28.0
Alignment: Center
Lines: 1
Behavior: Enabled
Highlighted
Baseline: Align Baselines
Line Breaks: Truncate Tail
Autoshrink: Fixed Font Size
Tighten Letter Spacing
Highlighted: Default
Shadow: Default
Shadow Offset: 0 Horizontal, -1 Vertical

View
Mode: Left
Tag: 0
Interaction: User Interaction Enabled
Multiple Touch
Alpha: 1
Background: Default
Tint: Default
Drawing: Opaque, Hidden
Clears Graphics Context
Clip Subviews

Change the text of your label to read "Heart Rate Monitor". Connect the ImageView to the ***heartimage*** property, and the TextView to the ***deviceInfo*** property.

With the project window opened, ensure that you have selected the active scheme configuration ***HeartMonitor\iPhone Simulator***.

Build and run your app; you can do this by selecting ***Product\Run*** from the Xcode menu, or alternatively pressing ***Command*** + ***R***. The iOS simulator will appear, and your app will be displayed on-screen.

As you can see, your application doesn't do much at the moment. However, this is a good check to make sure that all your bits and pieces compile correctly. In the next section, you'll add some functionality to make it talk to the Bluetooth device.

Leveraging the Bluetooth Framework

Open ***HRMViewController.m*** and replace **`viewDidLoad:`** with the following code:

```
- (void)viewDidLoad
{
    [super viewDidLoad];

        // Do any additional setup after loading the view, typically
from a nib.
        self.polarH7DeviceData = nil;
        [self.view setBackgroundColor:[UIColor
groupTableViewBackgroundColor]];
        [self.heartImage setImage:[UIImage imageNamed:@"HeartImage"]];

        // Clear out textView
        [self.deviceInfo setText:@""];
        [self.deviceInfo setTextColor:[UIColor blueColor]];
        [self.deviceInfo setBackgroundColor:[UIColor
groupTableViewBackgroundColor]];
        [self.deviceInfo setFont:[UIFont fontWithName:@"Futura-
CondensedMedium" size:25]];
        [self.deviceInfo setUserInteractionEnabled:NO];

        // Create your Heart Rate BPM Label
        self.heartRateBPM = [[UILabel alloc]
initWithFrame:CGRectMake(55, 30, 75, 50)];
        [self.heartRateBPM setTextColor:[UIColor whiteColor]];
        [self.heartRateBPM setText:[NSString stringWithFormat:@"%i",
0]];
        [self.heartRateBPM setFont:[UIFont fontWithName:@"Futura-
CondensedMedium" size:28]];
        [self.heartImage addSubview:self.heartRateBPM];

        // Scan for all available CoreBluetooth LE devices
        NSArray *services = @[[CBUUID
UUIDWithString:POLARH7_HRM_HEART_RATE_SERVICE_UUID], [CBUUID
UUIDWithString:POLARH7_HRM_DEVICE_INFO_SERVICE_UUID]];
        CBCentralManager *centralManager = [[CBCentralManager alloc]
initWithDelegate:self queue:nil];
        [centralManager scanForPeripheralsWithServices:services
options:nil];
        self.centralManager = centralManager;
}
```

Here you initialize and set up your user interface controls and load your heart image from the Xcode assets library. Next you create the **`CBCentralManager`** object; the first argument sets the delegate — in this case, the view controller. The second argument (the queue) is set to nil, because the Peripheral Manager will run on the main thread.

You then call **`scanForPeripheralsWithServices:`**; this tells the Central Manager to search for all compliant services in range. Finally, you specify a search for all compliant heart rate monitoring devices and retrieve the device information associated with that device.

Adding the Delegate Methods

Once the Peripheral Manager is initialized, you immediately need to check its state. This tells you if the device your app is running on is compliant with the Bluetooth LE standard.

Adding centralManagerDidUpdateState:central

Open ***HRMViewController.m*** and replace **`centralManagerDidUpdateState:central`** with the following code:

```
- (void)centralManagerDidUpdateState:(CBCentralManager *)central
{
    // Determine the state of the peripheral
    if ([central state] == CBCentralManagerStatePoweredOff) {
        NSLog(@"CoreBluetooth BLE hardware is powered off");
    }
    else if ([central state] == CBCentralManagerStatePoweredOn) {
        NSLog(@"CoreBluetooth BLE hardware is powered on and ready");
    }
    else if ([central state] == CBCentralManagerStateUnauthorized) {
        NSLog(@"CoreBluetooth BLE state is unauthorized");
    }
    else if ([central state] == CBCentralManagerStateUnknown) {
        NSLog(@"CoreBluetooth BLE state is unknown");
    }
    else if ([central state] == CBCentralManagerStateUnsupported) {
        NSLog(@"CoreBluetooth BLE hardware is unsupported on this
platform");
    }
}
```

The above method ensures that your device is Bluetooth low energy compliant and it can be used as the central device object of your CBCentralManager. If the state of the central manager is powered on, you'll receive a state of **`CBCentralManagerStatePoweredOn`**. If the state changes to **`CBCentralManagerStatePoweredOff`**, then all peripheral objects that have been obtained from the central manager become invalid and must be re-discovered.

Let's try this out. Build and run your code – on an actual device, not the simulator. You should see the following output in the console:

```
CoreBluetooth[WARNING] <CBCentralManager: 0x14e3a8c0> is not powered on
CoreBluetooth BLE hardware is powered on and ready
```

Adding didDiscoverPeripheral:peripheral:

Remember that in **`viewDidLoad`**, you called **`scanForPeripheralWithServices:`** to start searching for Bluetooth LE devices that have the heart rate or device info services. When one of these devices is found, the **`didDiscoverPeripheral:peripheral:`** delegate method will be called, so implement that next:

```
- (void)centralManager:(CBCentralManager *)central
didDiscoverPeripheral:(CBPeripheral *)peripheral advertisementData:
(NSDictionary *)advertisementData RSSI:(NSNumber *)RSSI
{
    NSString *localName = [advertisementData
objectForKey:CBAdvertisementDataLocalNameKey];
    if ([localName length] > 0) {
        NSLog(@"Found the heart rate monitor: %@", localName);
        [self.centralManager stopScan];
        self.polarH7HRMPeripheral = peripheral;
        peripheral.delegate = self;
        [self.centralManager connectPeripheral:peripheral options:nil];
    }
}
```

When a peripheral with one of the designated services is discovered, the delegate method is called with the peripheral object, the advertisement data, and something called the ***RSSI***.

Note: RSSI stands for ***Received Signal Strength Indicator***. This is a cool parameter, because by knowing the strength of the transmitting signal and the RSSI, you can estimate the current distance between the central and the peripheral.

With this knowledge, you can invoke certain actions like reading data only when the central is close enough to the peripheral; if it's almost out of range then your app could wait until the RSSI is higher before it performs certain actions.

Here you check to make sure that the device has a non-empty local name, and if so you log out the name and store the **`CBPeripheral`** for later reference. You also cease scanning for devices and call a method on the central manager to establish a connection to the peripheral object.

Build and run your code again, but this time make sure you are actually wearing your heart rate monitor (it won't send data unless you're wearing it!). You should see something like the following in the console:

```
Found the heart rate monitor: Polar H7 252D9F
```

Adding centralManager:central:peripheral:

Your next step is to determine if you have established a connection to the peripheral. Open ***HRMViewController.m*** and replace **`centralManager:central:peripheral:`** with the following code:

```
- (void)centralManager:(CBCentralManager *)central
didConnectPeripheral:(CBPeripheral *)peripheral
{
    [peripheral setDelegate:self];
    [peripheral discoverServices:nil];
    self.connected = [NSString stringWithFormat:@"Connected: %@",
peripheral.state == CBPeripheralStateConnected ? @"YES" : @"NO"];
    NSLog(@"%@", self.connected);
}
```

When you establish a local connection to a peripheral, the central manager object calls the **`centralManager:didConnectPeripheral:`** method of its delegate object.

In your implementation of the method above, you first set your view controller to be the delegate of the peripheral object so that it can notify the view controller using callbacks. If no error occurs, you next ask the peripheral to discover the services associated with the device. Finally, you determine the peripheral's current state to see if you've established a connection.

However, if the connection attempt fails, the central manager object calls **`centralManager:didFailToConnectPeripheral:error:`** of its delegate object instead.

Run this code on your device again (still wearing your heart rate monitor), and after a few seconds you should see this on the console:

```
Found the heart rate monitor: Polar H7 252D9F
Connected: YES
```

Adding peripheral:didDiscoverServices:

Once the services of the peripheral are discovered, **`peripheral:didDiscoverServices:`** will be called. So implement that with the following:

```
- (void)peripheral:(CBPeripheral *)peripheral didDiscoverServices:
(NSError *)error
{
    for (CBService *service in peripheral.services) {
        NSLog(@"Discovered service: %@", service.UUID);
        [peripheral discoverCharacteristics:nil forService:service];
    }
}
```

Here you simply iterate through each service discovered, log out its UUID, and call a method to discover the characteristics for that service.

Build and run, and this time you should see something like the following in the console:

```
Discovered service: Unknown (<180d>)
Discovered service: Device Information
Discovered service: Battery
Discovered service: Unknown (<6217ff49 ac7b547e eecf016a 06970ba9>)
```

Does that ***Unknown (<180d>)*** value look familiar to you? It should; it's the heart-rate monitor service id from the href="https://developer.bluetooth.org/gatt/services/Pages/ServicesHome.aspx">services section of the Bluetooth specification that you defined earlier:

```
#define POLARH7_HRM_HEART_RATE_SERVICE_UUID @"180D"
```

Adding peripheral:didDiscoverCharacteristicsForService:

Since you called **`discoverCharacteristics:forService:`**, once the characteristics are found for each service **`peripheral:didDiscoverCharacteristicsForService:`** will be called. So replace that with the following:

```
- (void)peripheral:(CBPeripheral *)peripheral
didDiscoverCharacteristicsForService:(CBService *)service error:
(NSError *)error
{
    if ([service.UUID isEqual:[CBUUID
UUIDWithString:POLARH7_HRM_HEART_RATE_SERVICE_UUID]])  {  // 1
        for (CBCharacteristic *aChar in service.characteristics)
        {
            // Request heart rate notifications
            if ([aChar.UUID isEqual:[CBUUID
UUIDWithString:POLARH7_HRM_MEASUREMENT_CHARACTERISTIC_UUID]]) { // 2
                [self.polarH7HRMPeripheral setNotifyValue:YES
forCharacteristic:aChar];
                NSLog(@"Found heart rate measurement characteristic");
            }
            // Request body sensor location
            else if ([aChar.UUID isEqual:[CBUUID
UUIDWithString:POLARH7_HRM_BODY_LOCATION_CHARACTERISTIC_UUID]]) { // 3
                [self.polarH7HRMPeripheral
readValueForCharacteristic:aChar];
                NSLog(@"Found body sensor location characteristic");
            }
        }
    }
    // Retrieve Device Information Services for the Manufacturer Name
    if ([service.UUID isEqual:[CBUUID
UUIDWithString:POLARH7_HRM_DEVICE_INFO_SERVICE_UUID]])  { // 4
        for (CBCharacteristic *aChar in service.characteristics)
        {
            if ([aChar.UUID isEqual:[CBUUID
UUIDWithString:POLARH7_HRM_MANUFACTURER_NAME_CHARACTERISTIC_UUID]]) {
                [self.polarH7HRMPeripheral
readValueForCharacteristic:aChar];
                NSLog(@"Found a device manufacturer name
characteristic");
            }
        }
    }
}
```

This method lets you determine what characteristics this device has. Taking each numbered comment in turn, you'll see the following actions:

1. First, check if the service is the the heart rate service.
2. If so, iterate through the characteristics array and determine if the characteristic is a heart rate notification characteristic. If so, you subscribe to this characteristic, which tells CBCentralManager to watch for when this characteristic changes and notify your code using **`setNotifyValue:forCharacteristic`** when it does.
3. If the characteristic is the body location characteristic, there is no need to subscribe to it (as it won't change), so just read this value.
4. If the service is the device info service, look for the manufacturer name and read it.

Build and run, and you should see something like the following in the console:

```
Found heart rate measurement characteristic
Found body sensor location characteristic
Found a device manufacturer name characteristic
```

Adding peripheral:didUpdateValueForCharacteristic:

The **`peripheral:didUpdateValueForCharacteristic:`** will be called when CBPeripheral reads a value (or updates a value periodically). You need to implement this method to check to see which characteristic's value has been updated, then call one of the helper methods to read in the value.

So implement the method as follows:

```
- (void)peripheral:(CBPeripheral *)peripheral
didUpdateValueForCharacteristic:(CBCharacteristic *)characteristic
error:(NSError *)error
```

```
{
    // Updated value for heart rate measurement received
    if ([characteristic.UUID isEqual:[CBUUID
UUIDWithString:POLARH7_HRM_MEASUREMENT_CHARACTERISTIC_UUID]]) { // 1
        // Get the Heart Rate Monitor BPM
        [self getHeartBPMData:characteristic error:error];
    }
    // Retrieve the characteristic value for manufacturer name received
    if ([characteristic.UUID isEqual:[CBUUID
UUIDWithString:POLARH7_HRM_MANUFACTURER_NAME_CHARACTERISTIC_UUID]]) {
// 2
        [self getManufacturerName:characteristic];
    }
    // Retrieve the characteristic value for the body sensor location
received
    else if ([characteristic.UUID isEqual:[CBUUID
UUIDWithString:POLARH7_HRM_BODY_LOCATION_CHARACTERISTIC_UUID]]) {  // 3
        [self getBodyLocation:characteristic];
    }

    // Add your constructed device information to your UITextView
    self.deviceInfo.text = [NSString stringWithFormat:@"%@\n%@\n%@\n",
self.connected, self.bodyData, self.manufacturer];  // 4
}
```

Looking at each numbered section in turn:

1. First check that a notification has been received to read heart rate BPM information. If so, call your instance method **`getHeartRateBPM:characteristic error:`** and pass in the value of the characteristic.
2. Next, check if a notification has been received to obtain the manufacturer name of the device. If so, call your instance method **`getManufacturerName:characteristic:`** and pass in the characteristic value.
3. Check if a notification has been received to determine the location of the device on the body. If so, call your instance method **`getBodyLocation:characteristic:`** and pass in the characteristic value.
4. Finally, concatenate each of your values and output them to your ***UITextView*** control.

You can build and run if you want, but only a few null values will be written to the text field, because you haven't implemented the helper methods yet. Let's do that next.

Adding getHeartRateBPM:(CBCharacteristic *)characteristic error:(NSError *)error

To understand how to interpret the data from a characteristic, you have to check the Bluetooth specification. For example, check out the entry for heart rate measurement.

You'll see that a heart rate measurement consists of a number of flags, followed by the heart rate measurement itself, some energy information, and other data. You need to write a method to read this, so implement **`getHeartRateBPM:(CBCharacteristic *)characteristic error:(NSError *)error`** in ***HRMViewController.m*** as follows:

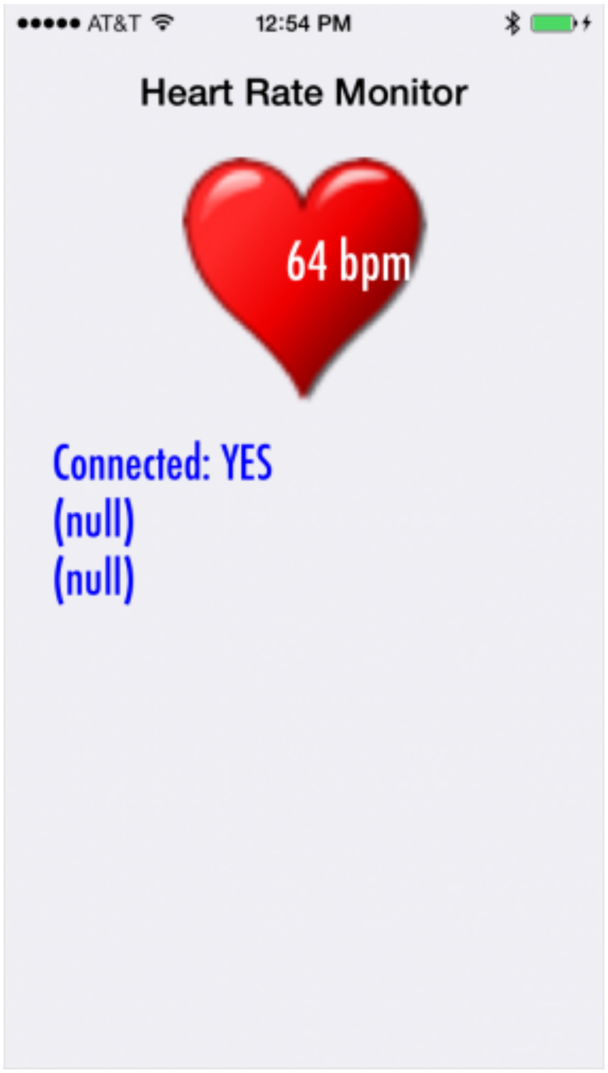
```
- (void) getHeartBPMData:(CBCharacteristic *)characteristic error:
(NSError *)error
{
    // Get the Heart Rate Monitor BPM
    NSData *data = [characteristic value];      // 1
    const uint8_t *reportData = [data bytes];
    uint16_t bpm = 0;

    if ((reportData[0] & 0x01) == 0) {          // 2
        // Retrieve the BPM value for the Heart Rate Monitor
        bpm = reportData[1];
    }
    else {
        bpm = CFSwapInt16LittleToHost(*(uint16_t *)(&reportData[1]));
// 3
    }
    // Display the heart rate value to the UI if no error occurred
    if( (characteristic.value)  || !error ) {   // 4
        self.heartRate = bpm;
        self.heartRateBPM.text = [NSString stringWithFormat:@"%i bpm",
bpm];
        self.heartRateBPM.font = [UIFont fontWithName:@"Futura-
CondensedMedium" size:28];
        [self doHeartBeat];
        self.pulseTimer = [NSTimer scheduledTimerWithTimeInterval:(60.
/ self.heartRate) target:self selector:@selector(doHeartBeat)
userInfo:nil repeats:NO];
    }
    return;
}
```

The above method runs each time the peripheral sends new data; it's responsible for handling heart monitor device notifications received by the peripheral delegate.

Once again, going through the numbered comments one by one reveals the following:

1. Convert the contents of your characteristic value to a data object. Next, get the byte sequence of your data object and assign this to your **`reportData`** object. Then initialize your **`bpm`** variable which will store the heart rate information.
2. Next, obtain the first byte at index 0 in the array as defined by **`reportData[0]`** and mask out all but the 1st bit. The result returned will either be 0, which means that the 2nd bit is not set, or 1 if it is set. If the 2nd bit is not set, retrieve the BPM value at the second byte location at index 1 in the array.
3. If the second bit is set, retrieve the BPM value at second byte location at index 1 in the array and convert this to a 16-bit value based on the host's native byte order.
4. Output the value of **`bpm`** to your **`heartRateBPM`** UILabel control, and set the fontName and fontSize. Assign the value of **`bpm`** to **`heartRate`**, and again set the control's font type and size. Finally, set up a timer object **`[NSTimer scheduledTimerWithTimeInterval:(60. / self.heartRate) target:self selector:@selector(doHeartBeat) userInfo:nil repeats:NO];`** which calls **`doHeartBeat:`** at 60-second intervals; this performs the basic animation that simulates the beating of a heart through the use of Core Animation.

Build and run, and at long last you'll see your heart beat on display in the app!



Connected: YES
(null)
(null)

My resting pulse

Adding getManufacturerName:(CBCharacteristic *)characteristic

Next let's add the code to read the manufacturer name characteristic. Implement **`getManufacturerName:(CBCharacteristic *)characteristic`** in ***HRMViewController.m*** as follows:

```
// Instance method to get the manufacturer name of the device
- (void) getManufacturerName:(CBCharacteristic *)characteristic
{
    NSString *manufacturerName = [[NSString alloc]
initWithData:characteristic.value encoding:NSUTF8StringEncoding];  // 1
    self.manufacturer = [NSString stringWithFormat:@"Manufacturer: %@",
manufacturerName];    // 2
    return;
}
```

The above method executes each time the peripheral sends new data; it handles heart monitor device notifications received by the Peripheral delegate.

This method isn't terribly long or complicated, but take a look at each commented section to see what's going on:

1. Take the value of the characteristic discovered by your peripheral to obtain the manufacturer name. Use **`initWithData:`** to return the contents of your characteristic object as a data object and tell NSString that you want to use **`NSUTF8StringEncoding`** so it can be interpreted as a valid string.
2. Next, assign the value of the manufacturer name to **`self.manufacturer`** so that you can display this value in your UITextView control.

You could build and run here, but I'd recommend skipping to the next one first.

Adding getBodyLocation:(CBCharacteristic *)characteristic

The last step is to add the code to read the body sensor location characteristic. Replace **`getBodyLocation:(CBCharacteristic *)characteristic`** in ***HRMViewController.m*** with the following code:

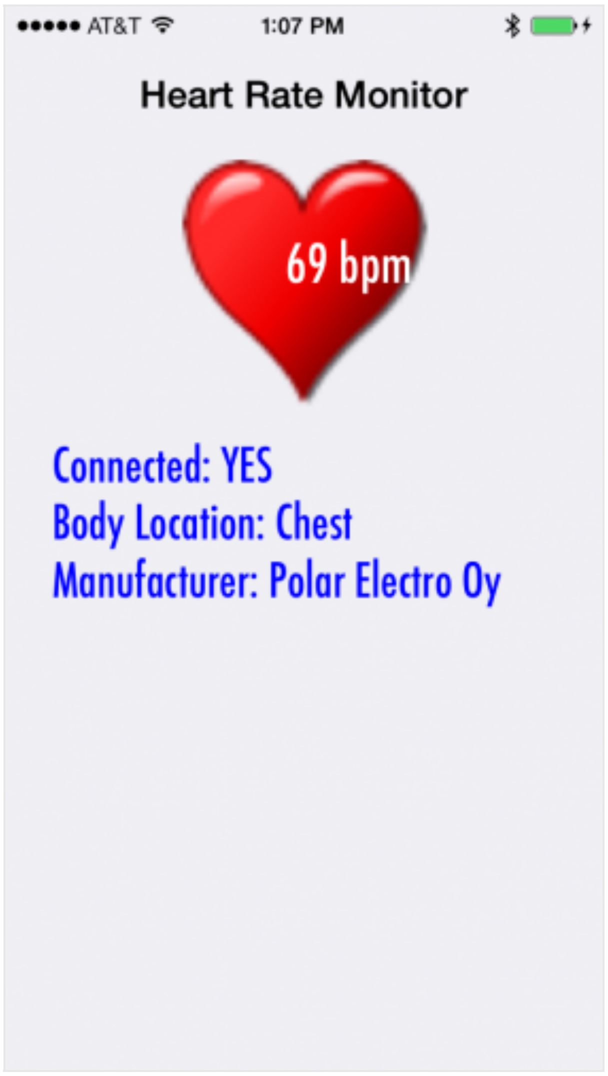
```
- (void) getBodyLocation:(CBCharacteristic *)characteristic
{
    NSData *sensorData = [characteristic value];         // 1
    uint8_t *bodyData = (uint8_t *)[sensorData bytes];
    if (bodyData ) {
        uint8_t bodyLocation = bodyData[0];  // 2
        self.bodyData = [NSString stringWithFormat:@"Body Location:
%@", bodyLocation == 1 ? @"Chest" : @"Undefined"]; // 3
    }
    else {  // 4
        self.bodyData = [NSString stringWithFormat:@"Body Location:
N/A"];
    }
    return;
}
```

The above method executes each time the peripheral sends new data; it handles heart monitor device notifications received by the Peripheral delegate.

Stepping through the numbered comments reveals the following:

1. Use the value of the characteristic discovered by your peripheral to obtain the heart rate monitor's body location. Next, convert the characteristic value to a data object consisting of byte sequences and assign this to your **`bodyData`** object.
2. Next, determine if you have device body location data to report and access the first byte at index 0 in your array as defined by **`bodyData[0]`**.
3. Next, determine the body location of the device using the **`bodyLocation`** variable; here you're only interested in the location on the chest. Finally, assign the body location data to **`bodyData`** so that it can be displayed in your UITextView control.
4. If no data is available, assign N/A as the body location and assign it to **`self.bodyData`** variable so that it can be displayed in your UITextView control.

Build and run, and now the text view shows the manufacturer name and body location properly:



Connected: YES
Body Location: Chest
Manufacturer: Polar Electro Oy

Make Your Heart Beat a Little Faster!

Congratulations, you now have a working heart rate monitor, and even more importantly have a good understanding of how Core Bluetooth works. You can apply these same techniques to a variety of Bluetooth LE devices.

Before you go, we have one little bonus for you. For fun, let's make the heart image beat in time with the BPM data from the heart monitor.

Open ***HRMViewController.m*** and replace **`doHeartBeat`** as follows:

```
- (void) doHeartBeat
{
    CALayer *layer = [self heartImage].layer;
    CABasicAnimation *pulseAnimation = [CABasicAnimation
animationWithKeyPath:@"transform.scale"];
    pulseAnimation.toValue = [NSNumber numberWithFloat:1.1];
    pulseAnimation.fromValue = [NSNumber numberWithFloat:1.0];

    pulseAnimation.duration = 60. / self.heartRate / 2.;
    pulseAnimation.repeatCount = 1;
    pulseAnimation.autoreverses = YES;
    pulseAnimation.timingFunction = [CAMediaTimingFunction
functionWithName:kCAMediaTimingFunctionEaseIn];
    [layer addAnimation:pulseAnimation forKey:@"scale"];

    self.pulseTimer = [NSTimer scheduledTimerWithTimeInterval:(60. /
self.heartRate) target:self selector:@selector(doHeartBeat)
userInfo:nil repeats:NO];
}
```

In the method above, you first create a **`CALayer`** class to manage your image-based content for the animation. You then create a **`pulseAnimation`** variable to perform basic, single-keyframe animation for your layer. Finally, you use the **`CAMediaTimingFunction`** that defines the pacing of the animation.

Build and run your app; you should see the heart image pulsate with each heartbeat received from the heart monitor. Try some light exercise (or try coding an Android app!) and watch your heart rate rise!

Where to Go From Here?

In this tutorial you've learned about Core Bluetooth LE and how you can use this to connect with Low Energy peripheral devices to retrieve certain attributes pertaining to the device.

Another example of Core Bluetooth devices is iBeacons. If you'd like to learn more about that, check out the ***What's New in Core Location*** chapter in iOS 7 by Tutorials. The book contains more info and examples of iBeacons along with tons of other chapters on almost everything else in iOS 7.

Here is the completed sample project with all of the code from the above tutorial. If you liked this tutorial and would like to see more Core Bluetooth tutorials in the future, please let me know in the forums!