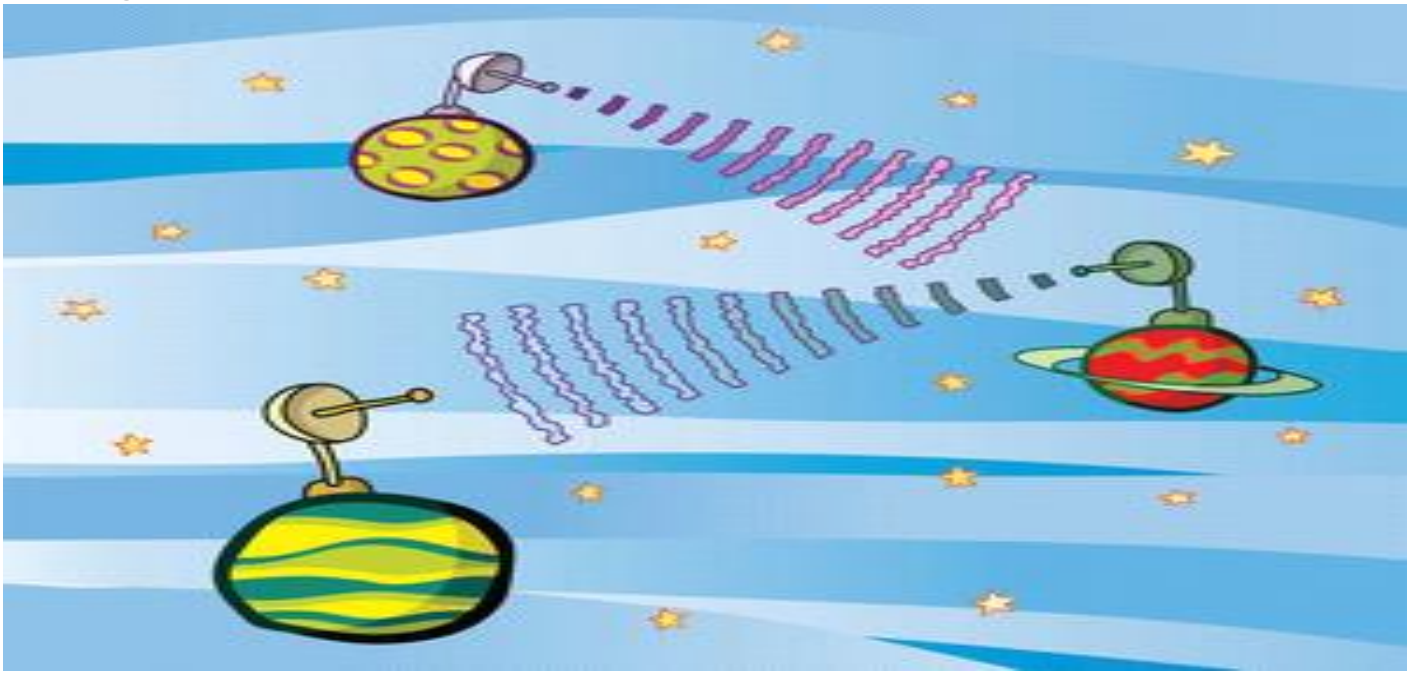
BLE Controlling Pi

Tuesday, June 06, 2017 8:10 PM

Getting BLE to behave on the Pi



Under the Radar

BLE is a convenient choice for wireless communication where WiFi and Internet connectivity aren't available, but getting it to work on a Raspberry Pi can be a minefield of compatibility problems. We look at the main concepts and debugging tools you will need to get the most out of BLE on your Pi.
BLE (Bluetooth Low Energy) is one of several standards developed for short-range wireless communication between devices. By design, BLE consumes very little power, which makes it attractive for always-on applications that don't have continual human interaction and maintenance. As a result, it's turning up in more and more products.

BLE interaction with users is less complex than other communication technologies, because devices exchange data quickly without the usual pairing process. This makes it suitable for iBeacon applications, in which an app detects the proximity of a BLE device (e.g., I'm in a shop or at a particular stand at a conference).

Something I've noticed in building BLE prototypes and projects on the Raspberry Pi, however, is that it's a whole load harder than you'd expect. As others have found with their Rasp Pi applications, supported devices and interoperable libraries change frequently. If you're used to debugging on one machine, and you add to that debugging on several machines, with a substantial amount of the operability occurring in the air, you'll find it's not fun.
In this article, I'll run through some of the core concepts of BLE and how these relate to the more popular tools available on the Pi and in iOS (as an example app platform). I'll also run through the debugging tools to keep close at hand. This article isn't so much a single project as a helpful collection of information lacking from many of the project pages on the web.

BLE Parts
To begin, I'll look at what BLE is suited to, avoiding the holy war of wireless standards and mostly looking at BLE's strengths.
BLE is designed for lower power consumption and is increasingly supported by phones, devices, and add-ons for the Raspberry Pi. Anyone who has paired their Fitbit to their phone or their phone to their car knows the typical process, but if you get into the BLE spec a bit more, some great hacks can allow you to change the user experience.

What you'll be used to is the scan and pair process, but underneath the hood, you can scan without pairing to see who's nearby. With a better understanding of BLE, you can build better projects and hack deeper into what's possible.

BLE, like any local wireless, is a way to transfer data between devices. To do this, the devices have to establish a connection, which in turn requires that they find each other. Therefore, the first stage is for some devices to advertise and others to scan for the advertised devices. The devices advertise their name, but when each device is interrogated, it will tell you more about its services and, within each of those, its characteristics.
The BLE spec has a list of standard services for things like heart rate, glucose, phone alert state, and many others. Each of these is designed to contain that specific type of information, be it the battery level service, which simply provides the level of a device's battery, or blood pressure, which provides a more complex set of information [1].

Each part of the service is called a characteristic, and this is where you start to read and write data. Think of the characteristics as fields in a table. Some of these fields can only be read, such as the name of the device while others can be written such as toggling a light switch on and off.
BLE devices take on one of two roles: the central or the peripheral. The central scans for peripherals and uses them to get things done. For example, your phone looks for a BLE fitness device and gets data from it. The peripheral typically advertises, so nearby centrals can get its data.

This simplified description should provide a good understanding to let you accomplish more with BLE rather than just following the project instructions. For example, one of the first things you can hack with is the fact that you don't have to pair with a device, you can just scan for it and pick up its presence. If you scan for nearby devices, you can remember their device IDs and start working out who is in the room.

What Can You Do with BLE?

Controlling devices, monitoring, manipulating lighting, sensing temperatures – any data that is fairly small can easily be transferred over BLE, meaning that most sensing and control easily falls into its scope.

Bundling this ability with a wired connection opens up the possibility of home control [2], as well as iBeacons and anything that involves wireless control, such as turning a lamp on and off [3].

iBeacon is a technology developed by Apple and based on BLE that allows an app running on a smartphone to find "beacons" and react accordingly. The beacon itself is fairly dumb; it simply emits a signal that uniquely identifies itself. The app can then work out what it's near and give the user access to information or device features.

BLE on the Pi

To get started working with BLE on the Raspberry Pi, you need a clean Linux installation, a compatible BLE device, and a way of writing a BLE dummy peripheral for which an app, or another Raspberry Pi, can scan.

Once you have your BLE devices, start by grabbing a clean Raspbian install [4]. Select the full download, not the network installation, because it will save you a few steps. Unzip the NOOBS download and put it on an empty SD card. When you boot the Pi up, you'll be prompted to select which operating system to install; select *Raspbian*. The Pi will then install Raspbian on the SD card and reboot. Once this is done, log in and get started.
The first thing to install is the dependencies:

```
sudo apt-get install libdbus-1-dev libglib2.0-dev libdbus-glib-1-dev
sudo apt-get install libusb-dev libudev-dev libreadline-dev
```

Next, you have to install Bluez, the package that supports BLE on Linux. A lot of pages on the web will walk you through installing this from source for the latest Raspbian (at time of writing, October 2014), which includes a working version of Bluez, albeit a slightly old version. To install bluez, type:

```
sudo apt-get install bluez
```

Next you need to plug in your BLE USB dongle. When choosing the dongle, make sure that it's a Rasp Pi-compatible BLE v4.0 dongle. Also, save yourself hours of debugging and plug the dongle into one of the Pi's USB ports rather than to a USB hub. You can see the dongle using lsusb (Listing 1).
Listing 1

```
lsusb
root@raspberrypi:/home/pi/ble# lsusb
Bus 001 Device 002: ID 0424:9512 Standard Microsystems Corp.
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 001 Device 003: ID 0424:ec00 Standard Microsystems Corp.
Bus 001 Device 012: ID 0cf3:3005 Atheros Communications, Inc. AR3011 Bluetooth
Bus 001 Device 005: ID 05e3:0608 Genesys Logic, Inc. USB-2.0 4-Port HUB
Bus 001 Device 013: ID 0a5c:21e8 Broadcom Corp.
Bus 001 Device 007: ID 148f:5370 Ralink Technology, Corp. RT5370 Wireless Adapter
Bus 001 Device 010: ID 1997:1221
```

This command lists the available USB devices. Next, choose the device for inspection:

```
root@raspberrypi:/home/pi/ble
# sudo lsusb -v -d 0a5c:
```

If everything is working, you'll get some very long output containing the device descriptor, status, and so on. If your BLE is plugged into a USB hub rather than the Raspberry Pi, this command returns nothing or an error.
Now that you know the dongle is working, you need to learn about any BLE devices, so you need to use one of the HCI tools bundled with Bluez. The

```
hciconfig
```

command lists the devices, whereas

```
hciconfig hci0
```

shows the configuration for just one device. You can also set things like the name of the device using:

```
hciconfig hci0 name <MyPiDongle>
```

Another tool you'll use later is hcidump, so you should install this now with the

```
apt-get install bluez-hcidump
```

command.

Sending and Receiving with bleno

Once you have the device plugged in and you can see it, it's time to code. For a Rasp Pi to be discoverable, you need to advertise some services, and you can do this a few different ways: directly on the command line using the hciconfig and hcitool commands or with the use of a programming language.

For the purposes of this example, I'll use node.js, which is event based and lends itself to listening for and responding to BLE events. Start by installing node from the source:

sudo apt-get install nodejs npm

To check the version installed, use:

nodejs -v

npm -v

The first command should give you *TODO* and the second should return the version of npm installed, which was 1.3.21 at the time of writing. (Incidentally, this process is the worst part of getting everything to work: tracking down incompatible versions. Be prepared to uninstall and revert back to older versions of software in the future, at least until all the components reach a consensus.)

Next, start a project and install the bleno module [5] for node.js:

mkdir ~/bleproject/ && cd ~/bleproject/

npm install bleno

Bleno is a node module for building BLE devices that assume the peripheral role. If you're looking at building a BLE device that takes on the central role, such as a Raspberry Pi that connects to lots of BLE devices harvesting data, you'll need its counterpart, noble [6].

In the hierarchy of BLE concepts, you have to advertise a service, and that service needs characteristics. In bleno you can do so with

bleno.startAdvertising(<name>, <serviceUuids>)

which starts the advertising. The <name> will appear in nearby scanning devices, and the <serviceUuids> will compose an array of UUIDs that identify each service. This is implemented in lines 23-30 of Listing 2. To provide a service, you need to have some characteristics and descriptors, as in Listing 2, which is stub code that I'll build up later.

Listing 2

Characteristics and Descriptors

```
01 var Descriptor = bleno.Descriptor;
02 var descriptor = new Descriptor({
03    uuid: '2901',
04    value: 'value' // static value, must be of type Buffer or string if set
05 });
06
07 var Characteristic = bleno.Characteristic;
08 var characteristic = new Characteristic({
09     uuid: 'fff1',
10     properties: [ 'read', 'write', 'writeWithoutResponse' ]
11     value: 'ff', // optional static value, must be of type Buffer
12     descriptors: [ descriptor ]
13 });
14 var PrimaryService = bleno.PrimaryService;
15 var primaryService = new PrimaryService({
16     uuid: 'fffffffffffffffffffffffffffffff0',
17     characteristics: [ characteristic ]
18 });
19 var services = [ primaryService ];
20 bleno.on('advertisingStart', function(error) {
21     bleno.setServices( services );
22 });
23 bleno.on('stateChange', function(state) {
24     console.log('BLE stateChanged to: ' + state);
25     if (state === 'poweredOn') {
26       bleno.startAdvertising('MyDevice',['fffffffffffffffffffffffffffffff0']);
27     } else {
28       bleno.stopAdvertising();
29     }
30 });
```

The hierarchy of advertising, services, and characteristics comes up again and again, so I'll walk you through the code. The line that starts the advertising is

bleno.startAdvertising('MyDevice', ['fffffffffffffffffffffffffffffff0']);

The first argument is the name of your device, which will appear in nearby BLE devices that are scanning. The second is the service UUID, which can either be custom, as in the example, or from a list of standard services. Behind this advert are some services created with new PrimaryService and registered with the setServices method. Each service bundles up some characteristics that can be read and/or written. In turn, the service has a characteristic, which has descriptors.

If you fire this script up with

sudo node script.js

(you need sudo to access the BLE device), you'll be able to scan for it with a BLE tool. You'll see the device appear and its advertising characteristic. Tap into the device in the BLE tool and then into the characteristic. Now you're at the point of being able to respond to these interactions which is done via callbacks such as onReadRequest, which is added to the Characteristic (Listing 3).

Listing 3

onReadRequest

```
01 onReadRequest: function(offset, callback) {
02     console.log('We got an onReadRequest!');
03     callback(Characteristic.RESULT_ATTR_NOT_LONG, null);
04 }
```

Run this again with

```
sudo node script.js
```

and when you read the characteristic using LightBlue, or whatever tool you have, you'll see the callback happen. The other important callback is shown in Listing 4.

Listing 4

onWriteRequest

```
01 onWriteRequest : function(newData, offset, withoutResponse, callback) {
02     console.log('got newData: ' + newData.toString('utf8'));
03     callback(bleno.Characteristic.RESULT_SUCCESS);
04 }
```

However, it would be more interesting to let devices write to a characteristic and something to happen. For example, to determine whether a red or blue light goes on consider the code in Listing 5. What you have here is enough to start controlling anything through BLE and the Pi.

Listing 5

Red or Blue Light?

```
01 onWriteRequest: function(data, offset, withoutResponse, callback) {
02     console.log('We got: ' + data); // THIS BIT HERE is where we get data
03     if (data == 'red') {
04         // Turn on the red light
05     } else
06     if (data == 'green') {
07         // Turn on the green...
08     }
09     callback(Characteristic.RESULT_SUCCESS);
10 }
```

Because nodejs is running on the Raspberry Pi, you can get flashing lights, moving robots, and the like in the callback. Suppose you have a script called toggle-lights.sh, which toggles the lights on and off. By calling this in the callback function, any BLE app can toggle the lights remotely (Listing 6).

Listing 6

Toggle Lights Remotely

```
01 onWriteRequest : function(newData, offset, withoutResponse, callback) {
02     console.log('got newData: ' + newData.toString('utf8'));
03     exec('./toggle-lights.sh ');
04     callback(bleno.Characteristic.RESULT_SUCCESS);
05 }
```

Getting the set of characteristics right for your device is like designing an API. Work out the parameters to pass between the Rasp Pi and whatever device will control it, and then implement those as characteristics.

From <http://www.raspberry-pi-geek.com/Archive/2014/08/Getting-BLE-to-behave-on-the-Pi/(offset)/2>

From <http://www.raspberry-pi-geek.com/Archive/2014/08/Getting-BLE-to-behave-on-the-Pi>