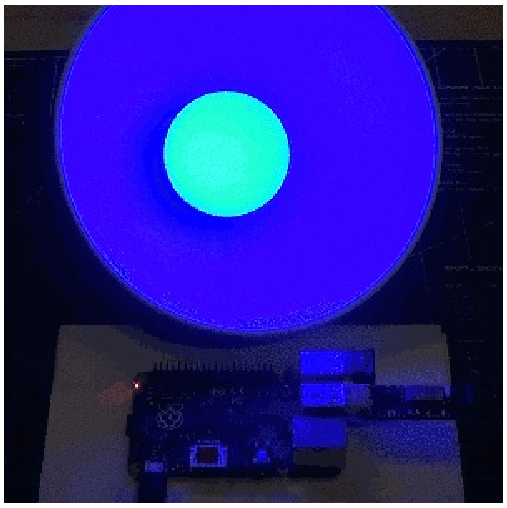
adafruit learning system

Reverse Engineering a Bluetooth Low Energy Light Bulb

Created by Tony DiCola



Last updated on 2015-03-01 08:45:06 PM EST

Guide Contents

Overview

This is a fun project that walks through how to reverse engineer a smart light bulb that uses Bluetooth Low Energy (BLE) to change its color. I stumbled on the Smart Bulb Colorific! (http://adafru.it/eDr) bulb recently at a local store and was intrigued by its Bluetooth-based control and relatively low price (for a 'smart' gadget at least). Because the light bulb uses Bluetooth Low Energy (which is a subset of Bluetooth 4.0) it means any BLE device can in theory control the bulb. But to control the bulb the protocol for communicating with it must be understood, and this guide will show you how to use the Bluefruit LE sniffer and other tools to reverse engineer a Bluetooth Low Energy gadget.

If you'd like to follow along and control a bulb yourself you'll first need a few things:

A Smart Bulb Colorific! light bulb (http://adafru.it/eDs). It's possible other BLE light bulbs can be controlled in a similar manner as discovered here, but I recommend picking up one of these bulbs to be sure you can control it. The bulbs use the Colorific! app on the Android (http://adafru.it/eDt) or iOS (http://adafru.it/eDu) app store for control. You can see a picture of the bulb I used to the left.

- Bluetooth 4.0 USB module (http://adafru.it/ecb). **Make sure the module supports Bluetooth Low Energy.** Older Bluetooth before version 4.0 does not support BLE!

- Raspberry Pi (http://adafru.it/eCB) of any model (A, B, A+, B+, Pi 2, etc.). This guide shows how to use tools and code on the Pi to control the bulb. Another linux computer can be used but we've only tested it on the Pi
- Bluefruit LE Sniffer (http://adafru.it/edE). The Bluefruit LE sniffer is a special version of the Bluefruit LE friend (http://adafru.it/edl) but with a firmware that allows it to watch BLE packets being sent to and from a device.

You should also familiarize yourself with Bluetooth Low Energy by first reading this introductory guide (http://adafru.it/eDv). This will help you understand terminology like GATT, service, and characteristic.

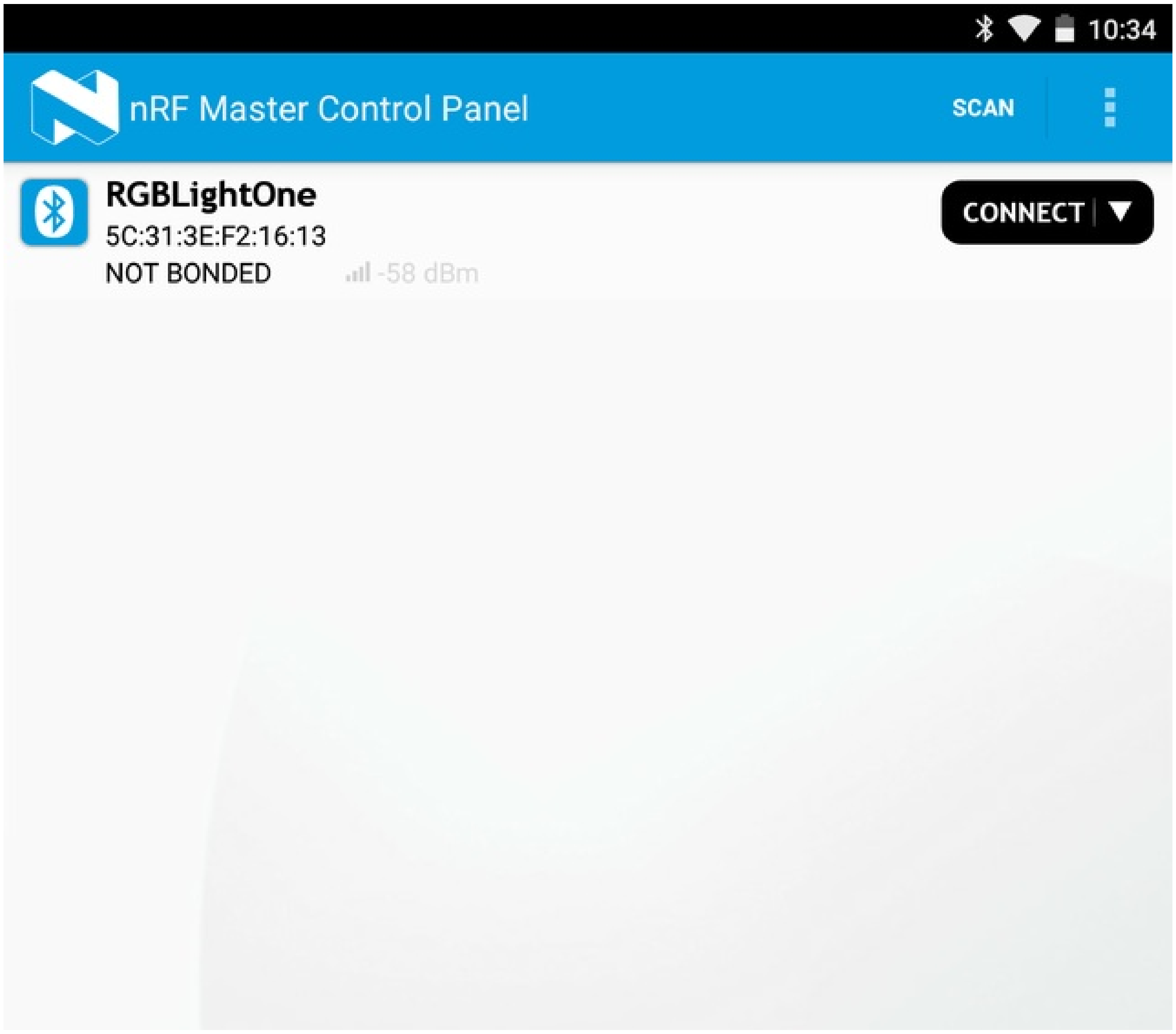
Continue on to learn where to start when exploring a Bluetooth Low Energy device.

Explore GATT

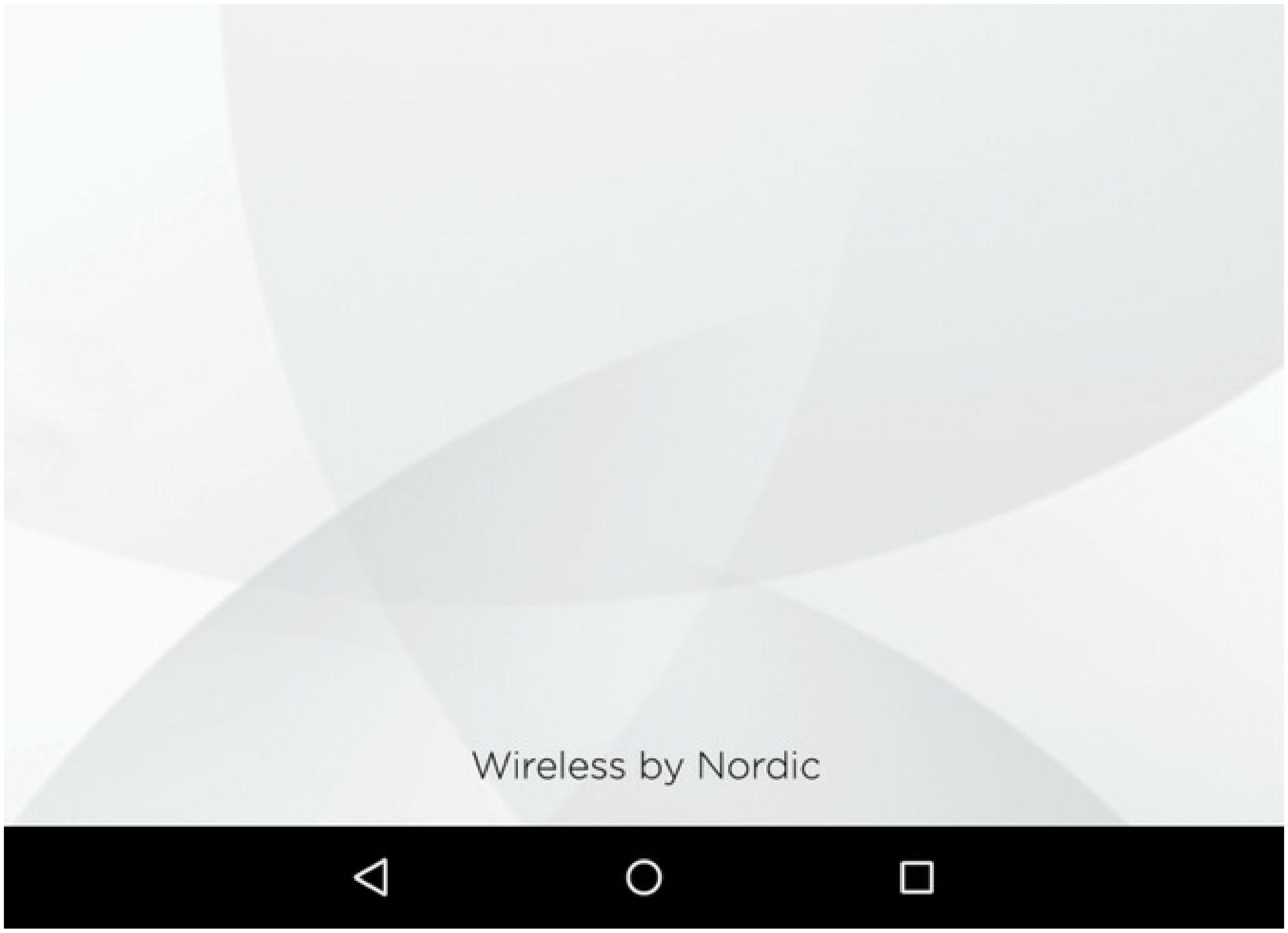
The first step to figuring out how the light bulb works is to investigate the GATT services exposed by the bulb. GATT, or generic attribute profile, is a protocol for interacting with a BLE device. Devices expose a list of services, and each service exposes a list of characteristics which can be read and/or written by a BLE application. Check out this great short introduction guide (http://adafru.it/ech) for more information on Bluetooth Low Energy and GATT.

An easy way to explore the GATT of a BLE device is using a smartphone or tablet and a BLE GATT exploration app. In this case I'll use Nordic Semiconductor's Master Control Panel app for Android (http://adafru.it/eDw). This is a free app that works well at letting you explore the GATT services of a BLE device.

With a light bulb turned on, I started the master control panel app and quickly saw the light bulb advertising itself as a BLE device:

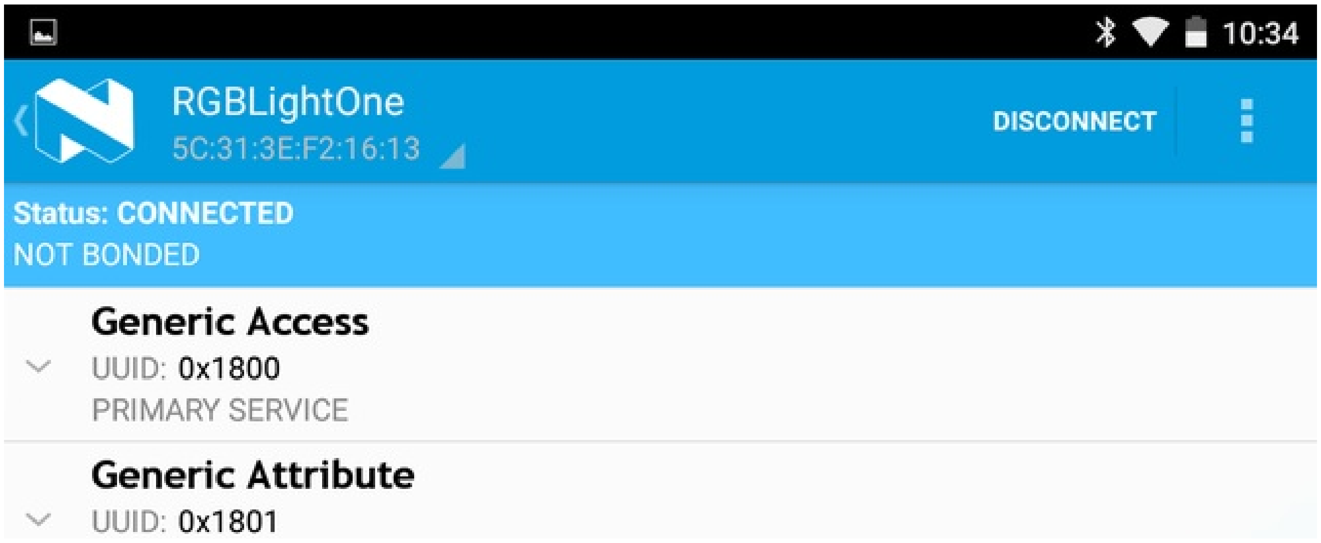


RGBLightOne
5C:31:3E:F2:16:13
NOT BONDED
-58 dBm



You can see master control panel lists all of the BLE devices that are advertising themselves, including this 'RGBLightOne' device that must be my light bulb. The most important thing to note here is the address of the device, 5C:31:3E:F2:16:13. The address is a unique ID that will be different for every device.

Next I touched the connect button to connect and discover the GATT services exposed by the bulb:

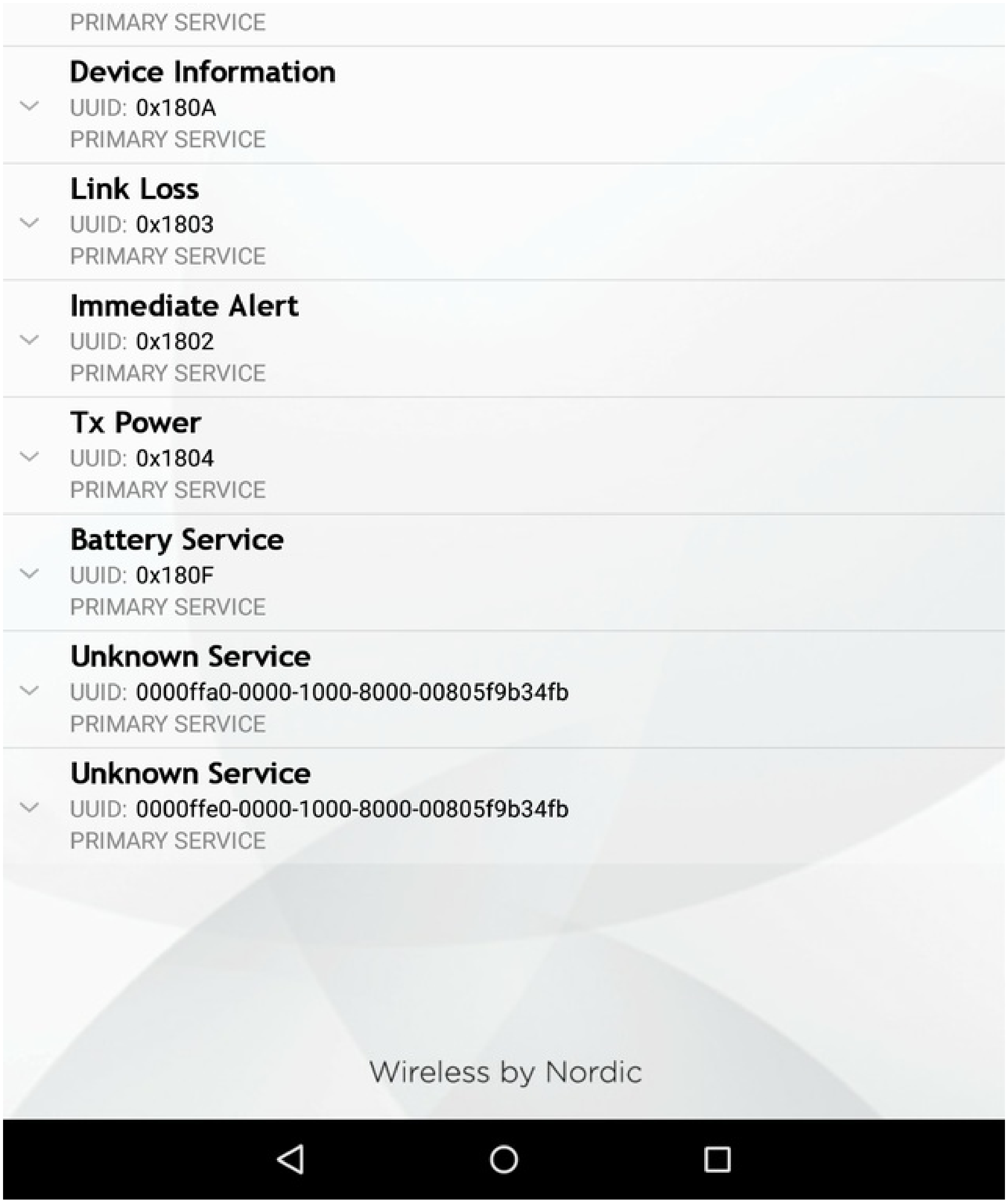


RGBLightOne
5C:31:3E:F2:16:13

Status: CONNECTED
NOT BONDED

Generic Access
UUID: 0x1800
PRIMARY SERVICE

Generic Attribute
UUID: 0x1801


PRIMARY SERVICE

Device Information
UUID: 0x180A
PRIMARY SERVICE

Link Loss
UUID: 0x1803
PRIMARY SERVICE

Immediate Alert
UUID: 0x1802
PRIMARY SERVICE

Tx Power
UUID: 0x1804
PRIMARY SERVICE

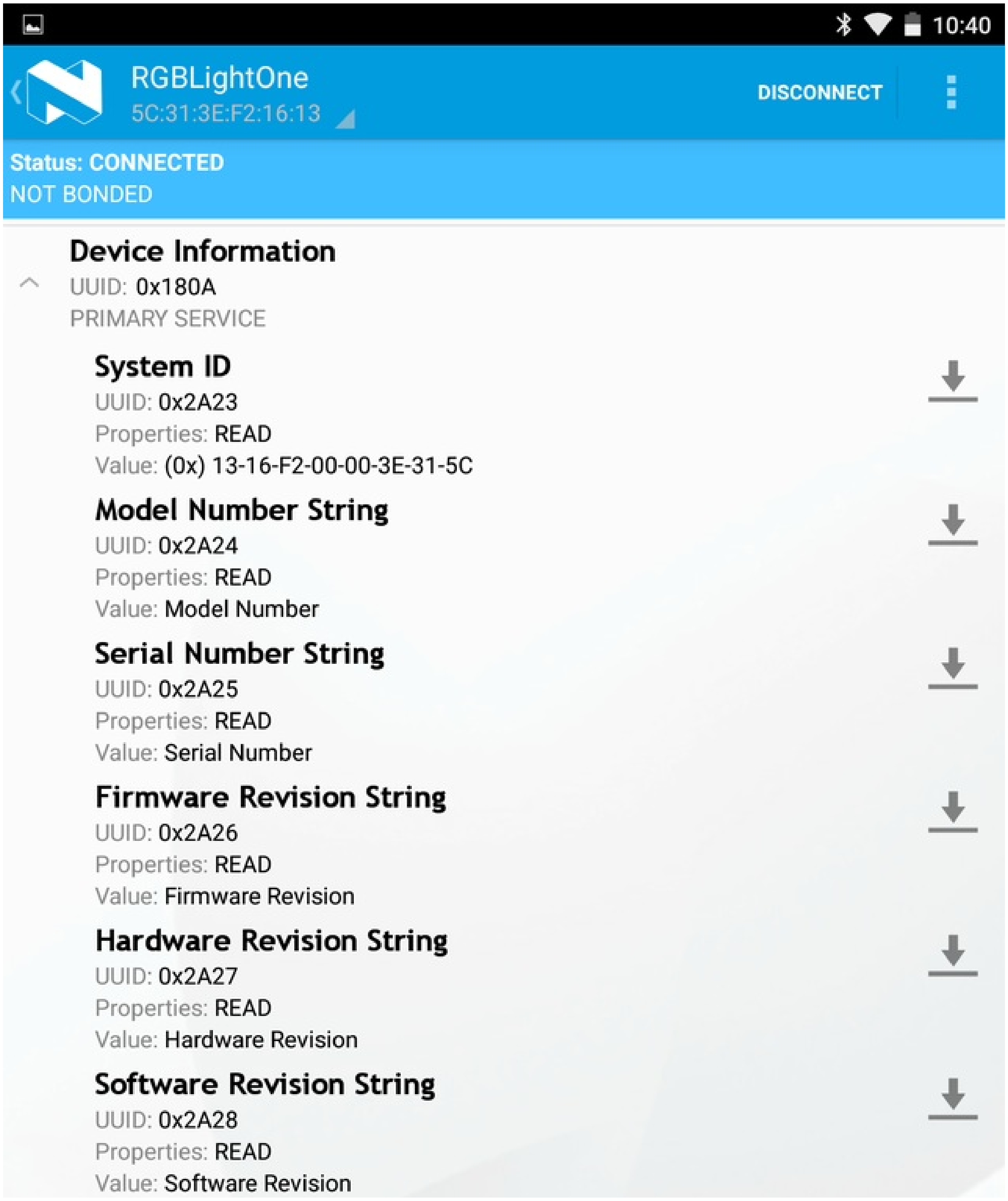
Battery Service
UUID: 0x180F
PRIMARY SERVICE

Unknown Service
UUID: 0000ffa0-0000-1000-8000-00805f9b34fb
PRIMARY SERVICE

Unknown Service
UUID: 0000ffe0-0000-1000-8000-00805f9b34fb
PRIMARY SERVICE

Now things are getting a little interesting! You can see the bulb exposes a handful of services. Notice each service has a UUID (universal unique ID) and some of the services have been recognized by master control panel as general services defined by the BLE spec (http://adafru.it/ddA). Short 4 hex character (16 bit) UUIDs identify these common services, like 0x180A, 0x1803, etc.

When a service is clicked on it drills down into the characteristics exposed by the service. For example this is what I see when I look at the device information service and read its characteristics:



Status: CONNECTED
NOT BONDED

Device Information
UUID: 0x180A
PRIMARY SERVICE

System ID
UUID: 0x2A23
Properties: READ
Value: (0x) 13-16-F2-00-00-3E-31-5C

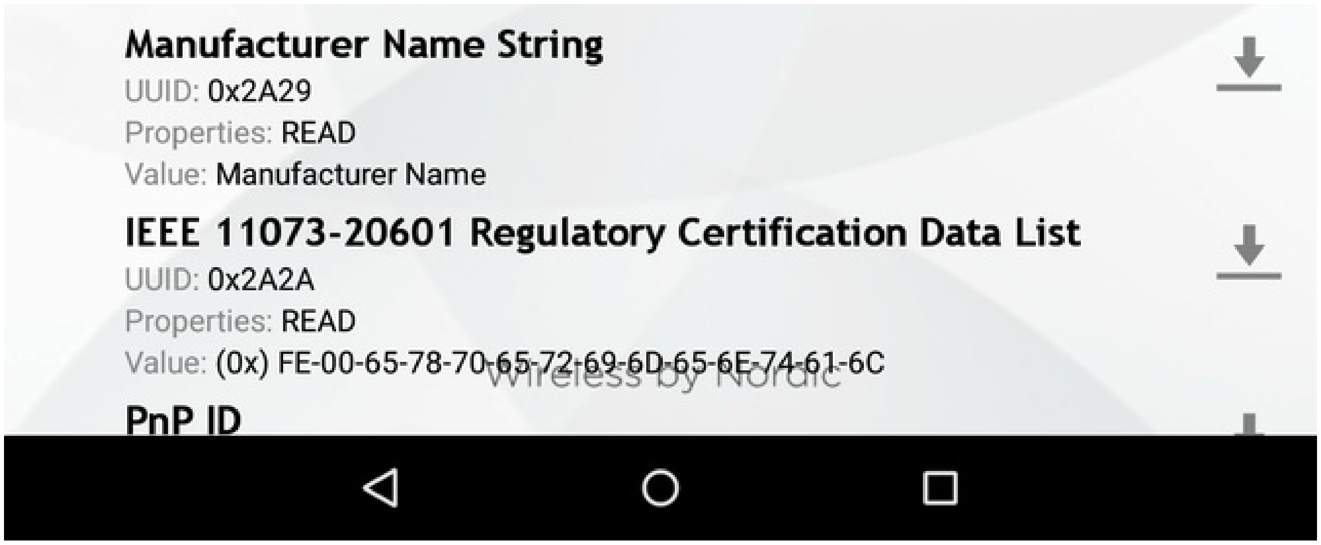
Model Number String
UUID: 0x2A24
Properties: READ
Value: Model Number

Serial Number String
UUID: 0x2A25
Properties: READ
Value: Serial Number

Firmware Revision String
UUID: 0x2A26
Properties: READ
Value: Firmware Revision

Hardware Revision String
UUID: 0x2A27
Properties: READ
Value: Hardware Revision

Software Revision String
UUID: 0x2A28
Properties: READ
Value: Software Revision


Manufacturer Name String
UUID: 0x2A29
Properties: READ
Value: Manufacturer Name

IEEE 11073-20601 Regulatory Certification Data List
UUID: 0x2A2A
Properties: READ
Value: (0x) FE-00-65-78-70-65-72-69-6D-65-6E-74-61-6C

PnP ID

Unfortunately there isn't much useful information in the device information service of the bulb. In fact the strings like model number, serial number, etc. appear to be set to default values like "Model Number" and "Serial Number".

Looking further at the services list I see two unknown services at the bottom. These are custom services that the manufacturer defined and are identifiable by their full 128-bit UUIDs, **0000ffe0-0000-1000-8000-00805f9b34fb** and **0000ffa0-0000-1000-8000-00805f9b34fb**. If I'm lucky the manufacturer will have documented these services so I can learn how to use the characteristics they expose.

In this case I did some searching online and found the two unknown services are actually defined by Texas Instrument's CC2540 (http://adafru.it/cF2) development kit as an accelerometer and simple keys service (http://adafru.it/eDx). This is a very interesting insight as it helps identify what hardware is powering this bulb, it's very likely a TI CC2540 BLE system on a chip.

Looking at all the device services for the bulb it's a bit puzzling why none of them appear to be related to LEDs or a light bulb. What likely happened is that the manufacturer adapted an existing BLE board (like the CC2540) and sample code to their needs instead of defining custom light bulb control services. This makes understanding how the bulb works a little more challenging since I'll need to look at the BLE commands sent by the bulb's control application to see what characteristics control the bulb.

Continue on to learn how to sniff Bluetooth Low Energy traffic and take a deeper look at how the bulb works.

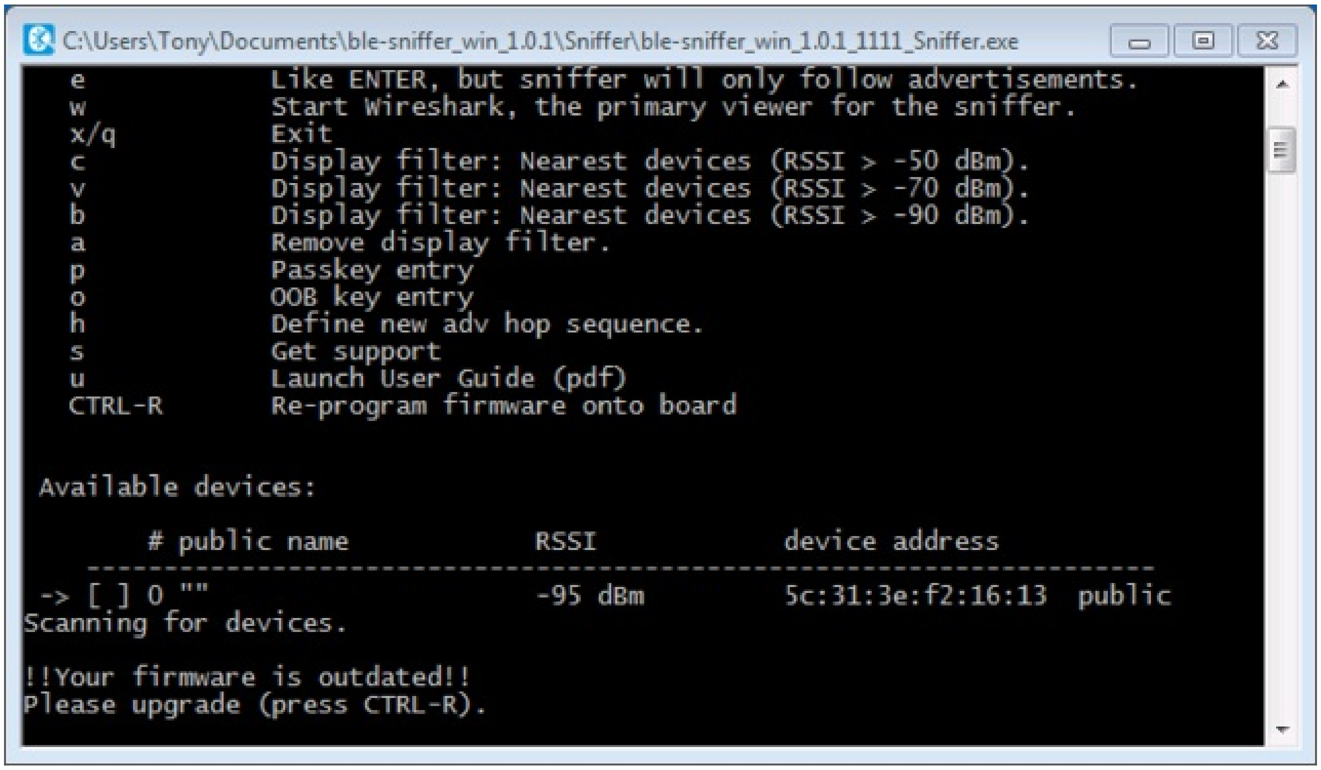
Sniff Protocol

Now that I've examined the GATT for the bulb and hit a dead end, I'll capture packets from the light bulb's control application to understand how it controls the bulb. This is possible using a BLE sniffer like the Bluefruit LE sniffer (http://adafru.it/edE) that's based on a Nordic nRF51822 chip (http://adafru.it/eDz). Using a special firmware and tools from Nordic (http://adafru.it/eDA)I can watch the BLE commands sent to the bulb to change its color. Nordic's sniffing tool even allows the use of Wireshark (http://adafru.it/eDB), a powerful and popular packet analysis tool, to examine the traffic.

If you're doing your own BLE device sniffing with the Bluefruit LE sniffer, make sure to read the guide on its usage first (http://adafru.it/eDC) as it explains how to install and setup the software. In my case I'll be using Nordic's tool on Windows as it lets me directly see the data in Wireshark.

One thing to note, after you've finished examining the GATT for a device be sure to disconnect or turn off any applications which were connected to the light bulb. **BLE only allows one connection to a device at a time**, and if you leave a GATT control app running then you won't be able to run the sniffer or other things that talk to the bulb!

First I begin by plugging in the Bluefruit LE sniffer and running's Nordics's sniffer application. After a few moments the tools scans and lists all available BLE devices. Notice the device it finds has the same address for the light bulb that I saw earlier while exploring its GATT--this helps me know I'm looking at the correct device:

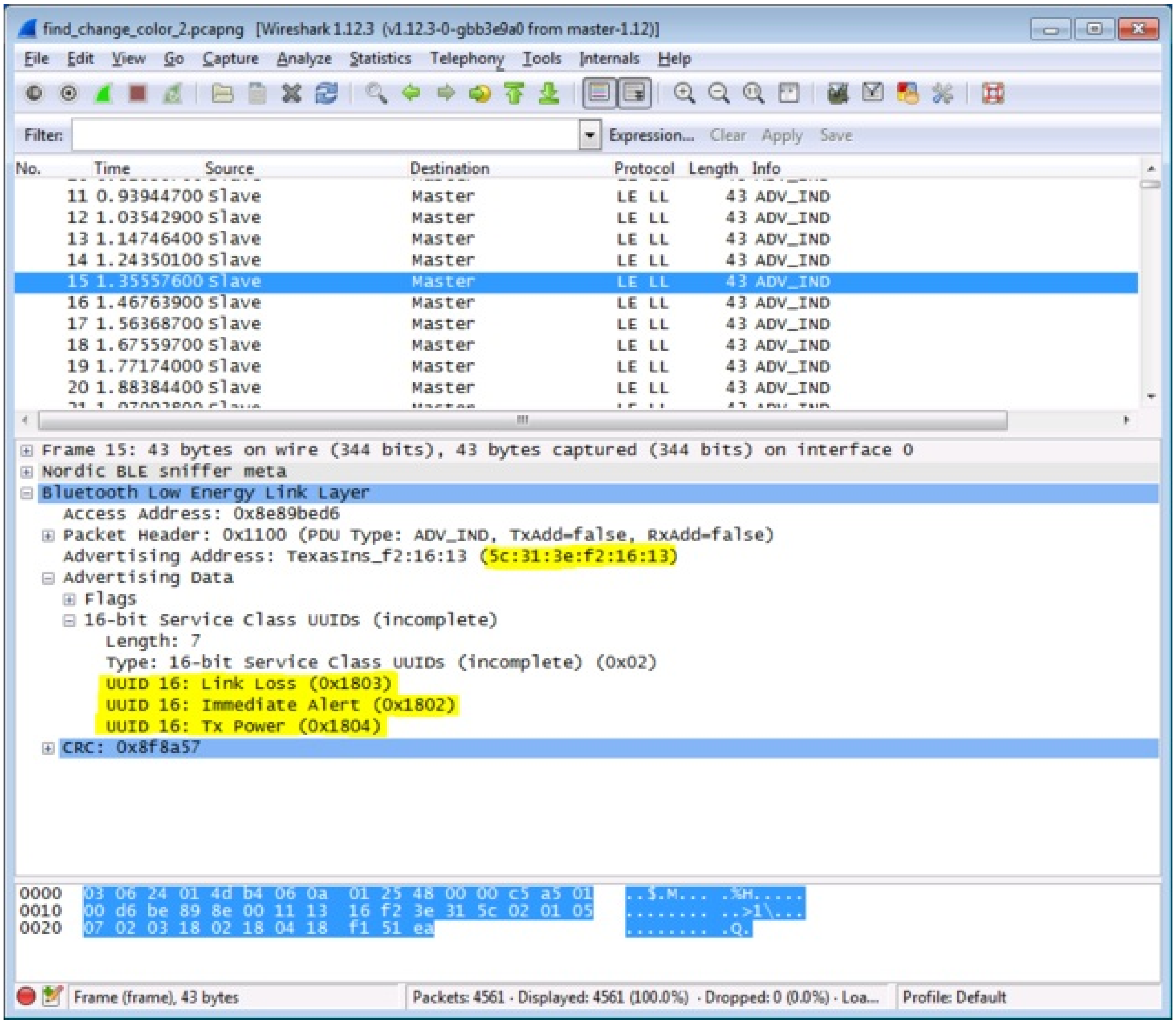

C:\Users\Tony\Documents\ble-sniffer_win_1.0.1\Sniffer\ble-sniffer_win_1.0.1_1111_Sniffer.exe

```
   e           Like ENTER, but sniffer will only follow advertisements.
   w           Start Wireshark, the primary viewer for the sniffer.
   x/q         Exit
   c           Display filter: Nearest devices (RSSI > -50 dBm).
   v           Display filter: Nearest devices (RSSI > -70 dBm).
   b           Display filter: Nearest devices (RSSI > -90 dBm).
   a           Remove display filter.
   p           Passkey entry
   o           OOB key entry
   h           Define new adv hop sequence.
   s           Get support
   u           Launch User Guide (pdf)
   CTRL-R      Re-program firmware onto board


 Available devices:

        # public name            RSSI           device address
    ----------------------------------------------------------------------
 -> [ ] 0 ""                     -95 dBm        5c:31:3e:f2:16:13  public
Scanning for devices.

!!Your firmware is outdated!!
Please upgrade (press CTRL-R).
```

Then I press **0** to select device zero in the list and then press **w** to start Wireshark with the packet capture.

One thing to note before you sniff BLE traffic with the Bluefruit LE sniffer is that it can be sensitive to noise from other BLE devices. Try to turn off all the other nearby BLE devices like tablets, phones, etc. Also be sure as few programs as possible are running on your PC becuase the tool needs to grab data from the Bluefruit sniffer as quickly as possible to prevent dropping packets.

Once Wireshark loads I quickly see a flood of advertising packets like:


find_change_color_2.pcapng [Wireshark 1.12.3 (v1.12.3-0-gbb3e9a0 from master-1.12)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply Save

```
No.  Time         Source  Destination  Protocol  Length  Info
  11 0.93944700  Slave   Master       LE LL     43 ADV_IND
  12 1.03542900  Slave   Master       LE LL     43 ADV_IND
  13 1.14746400  Slave   Master       LE LL     43 ADV_IND
  14 1.24350100  Slave   Master       LE LL     43 ADV_IND
  15 1.35557600  Slave   Master       LE LL     43 ADV_IND
  16 1.46763900  Slave   Master       LE LL     43 ADV_IND
  17 1.56368700  Slave   Master       LE LL     43 ADV_IND
  18 1.67559700  Slave   Master       LE LL     43 ADV_IND
  19 1.77174000  Slave   Master       LE LL     43 ADV_IND
  20 1.88384400  Slave   Master       LE LL     43 ADV_IND
```

```
⊞ Frame 15: 43 bytes on wire (344 bits), 43 bytes captured (344 bits) on interface 0
⊞ Nordic BLE sniffer meta
⊟ Bluetooth Low Energy Link Layer
    Access Address: 0x8e89bed6
  ⊞ Packet Header: 0x1100 (PDU Type: ADV_IND, TxAdd=false, RxAdd=false)
    Advertising Address: TexasIns_f2:16:13 (5c:31:3e:f2:16:13)
  ⊟ Advertising Data
    ⊞ Flags
    ⊟ 16-bit Service Class UUIDs (incomplete)
        Length: 7
        Type: 16-bit Service Class UUIDs (incomplete) (0x02)
        UUID 16: Link Loss (0x1803)
        UUID 16: Immediate Alert (0x1802)
        UUID 16: Tx Power (0x1804)
  ⊞ CRC: 0x8f8a57
```

```
0000  03 06 24 01 4d b4 06 0a  01 25 48 00 00 c5 a5 01   ..$.M... .%H.....
0010  00 d6 be 89 8e 00 11 13  16 f2 3e 31 5c 02 01 05   ........ ..>1\...
0020  07 02 03 18 02 18 04 18  f1 51 ea                  ........ .Q.
```

Frame (frame), 43 bytes
Packets: 4561 · Displayed: 4561 (100.0%) · Dropped: 0 (0.0%) · Loa...
Profile: Default

If you haven't used Wireshark before the interface can be a little daunting. I recommend reading the official documentation and watching videos (http://adafru.it/eDD) on using it to get a quick overview of the tool. At a very high level Wireshark's main window shows you three things:

- The top third is the list of packets that have been captured. When wire shark is capturing packets this list will quickly grow. You can scroll up and down to see packets as they've been received, and you can click a packet to see more information about it.

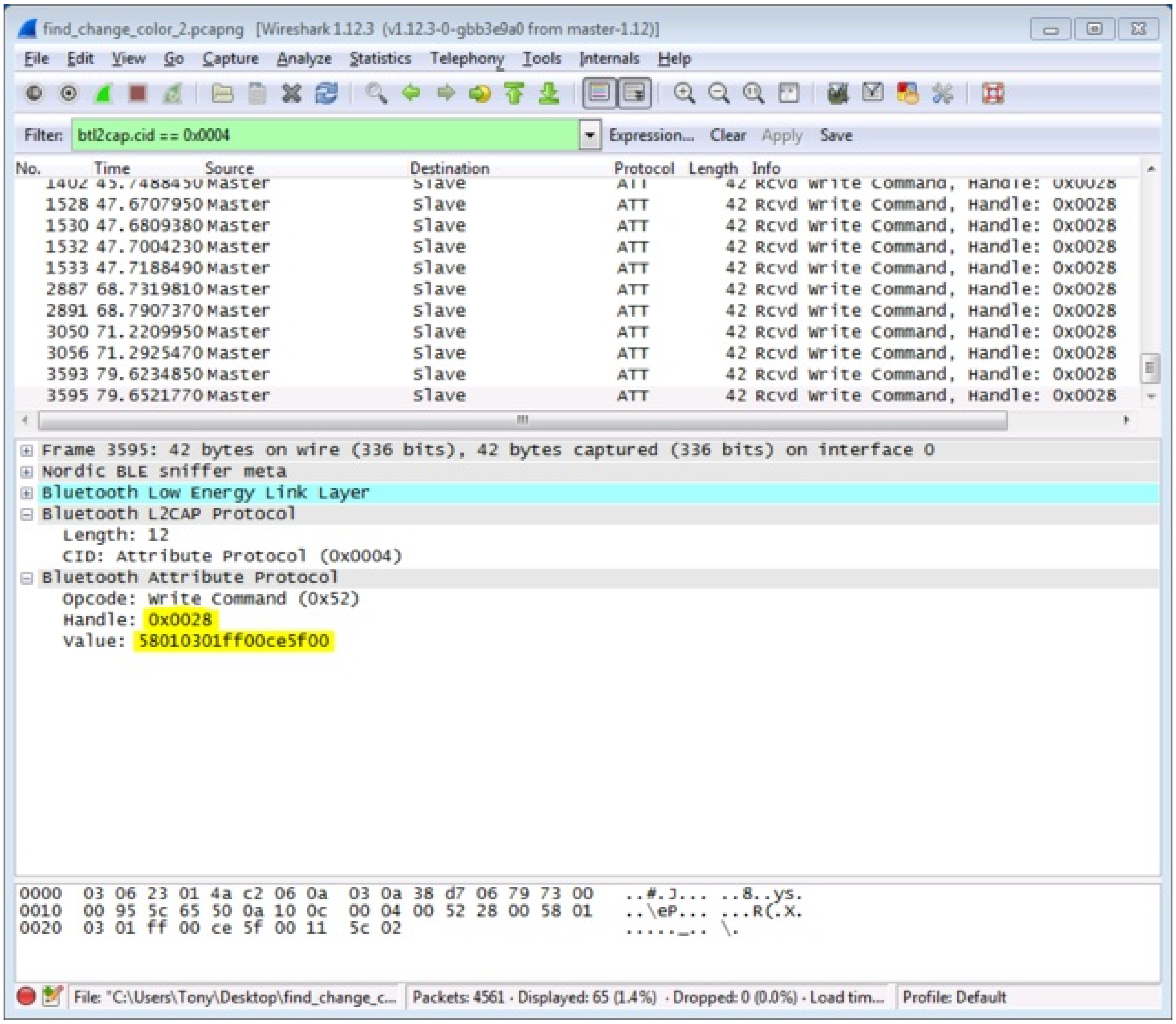
- The middle third is information that's been decoded from the packet. You can drill in to specific frames inside the packet to see what it's doing. Think of a network packet kind of like an onion, where you have layers of information that get more detailed as you peel them back and look deeper into the packet.
- The bottom third is the raw hexadecimal and ASCII representation of the packet data. It's interesting to note that as you click around information in the middle pane you can see highlighted the raw representation in the bottom pane.

In the picture above I've selected a BLE advertising packet and highlighted a few interesting details in the middle pane. You can see this packet is coming from the light bulb since it has the same address for the bulb that I learned earlier from exploring its GATT. You can also see the advertisement packet includes a few of the services I saw earlier that the bulb exposes.

Now it's time to load the bulb's control application and see what happens when I change the bulb's color. I loaded up the control app on an iOS device (note: for some reason I couldn't capture packets from the Colorific! Android application--my suspicion is the app might be generating packets that are malformed or confusing the nRF sniffer), found the bulb, and changed the bulb's color a few times. As I did this I saw small bursts of interesting packets scroll by in Wireshark. Unfortunately the continuous stream of other BLE communication packets added a lot of noise that made it difficult to focus on just the bulb control packets. However Wireshark has a great capability for filtering packets that I can use to help focus on on just the control packets.

To filter the packets I first stopped the capture of packets by pressing the red square stop button in Wireshark's toolbar. This is also a good point to save the capture file so it can be examined again without having to sniff device traffic.

Once the capture stopped I scrolled through the packet list and found some interesting packets under the ATT protocol. These are packets which contain commands for reading and writing BLE characteristics and are what I need to examine in more detail. To filter out all the other packets I entered the expression '**btl2cap.cid == 0x0004**' (without quotes) in the filter box below the toolbar, then pressed enter. Wireshark immediately hid all the other packets and just showed me the ATT packets:


find_change_color_2.pcapng [Wireshark 1.12.3 (v1.12.3-0-gbb3e9a0 from master-1.12)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter: btl2cap.cid == 0x0004
Expression... Clear Apply Save

```
No.    Time        Source   Destination  Protocol Length Info
 1402 45.7488450 Master   Slave        ATT      42 Rcvd Write Command, Handle: 0x0028
 1528 47.6707950 Master   Slave        ATT      42 Rcvd Write Command, Handle: 0x0028
 1530 47.6809380 Master   Slave        ATT      42 Rcvd Write Command, Handle: 0x0028
 1532 47.7004230 Master   Slave        ATT      42 Rcvd Write Command, Handle: 0x0028
 1533 47.7188490 Master   Slave        ATT      42 Rcvd Write Command, Handle: 0x0028
 2887 68.7319810 Master   Slave        ATT      42 Rcvd Write Command, Handle: 0x0028
 2891 68.7907370 Master   Slave        ATT      42 Rcvd Write Command, Handle: 0x0028
 3050 71.2209950 Master   Slave        ATT      42 Rcvd Write Command, Handle: 0x0028
 3056 71.2925470 Master   Slave        ATT      42 Rcvd Write Command, Handle: 0x0028
 3593 79.6234850 Master   Slave        ATT      42 Rcvd Write Command, Handle: 0x0028
 3595 79.6521770 Master   Slave        ATT      42 Rcvd Write Command, Handle: 0x0028
```

```
⊞ Frame 3595: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
⊞ Nordic BLE sniffer meta
⊞ Bluetooth Low Energy Link Layer
⊟ Bluetooth L2CAP Protocol
    Length: 12
    CID: Attribute Protocol (0x0004)
⊟ Bluetooth Attribute Protocol
    Opcode: Write Command (0x52)
    Handle: 0x0028
    Value: 58010301ff00ce5f00
```

```
0000  03 06 23 01 4a c2 06 0a  03 0a 38 d7 06 79 73 00   ..#.J... ..8..ys.
0010  00 95 5c 65 50 0a 10 0c  00 04 00 52 28 00 58 01   ..\eP... ...R(.X.
0020  03 01 ff 00 ce 5f 00 11  5c 02                     ......_.. \.
```

File: "C:\Users\Tony\Desktop\find_change_c... Packets: 4561 · Displayed: 65 (1.4%) · Dropped: 0 (0.0%) · Load tim... Profile: Default

Another way to filter to just the ATT packets is to select an ATT packet and drill into the **Bluetooth L2CAP Protocol** in the middle pane. Click the **CID: Attribute Protocol (0x0004)** line to select that part of the packet which identifies it as an ATT packet, then right click and choose the **Apply as Filter -> Selected** menu item. This will set the filter expression to only show packets with that exact attribute protocol value.

I can now restart packet capture with the filter applied and see only the ATT packets and none of the other BLE packet noise. This is great because it removes the packets I don't care about and lets me easily see the packets that matter.

As I changed the color of the bulb in the app I saw BLE characteristic write commands like the picture above shows. Notice the parts of the packet I've highlighted which show the handle that identifies the characteristic being updated, and the value which are the bytes to use as the new value of the characteristic.

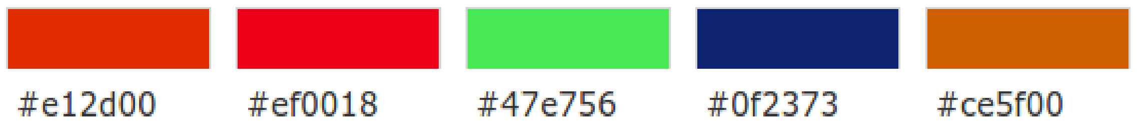
Now that I can see the packets associated with a color change I can try to reverse engineer what the bytes inside the packet mean. Reverse engineering can be more of an art than a science as it helps to have intuition and experience with what you're trying to reverse engineer. Unfortunately there is no fool-proof process for reverse engineering any device!

In this case I know the application is probably sending the bulb a new color, and colors are typically represented in a few common ways like as a 24-bit value (with 1 byte/8 bits for each red, green, and blue component), a floating point value for each color component, or perhaps even a string or ID that identifies the color inside a palette or lookup table. If I'm really unlucky the color information might be encoded or even encrypted in some way that I can't decipher--there is no guarantee I will be able to figure out the protocol!

I started by comparing the value of multiple color change packets to see how they differed. These are the values I saw (in hexadecimal):

- 58010301FF00E12D00
- 58010301FF00EF0018
- 58010301FF0047E756
- 58010301FF000F2373
- 58010301FF00CE5F00

Do you see a pattern? The first few bytes are exactly the same and only the last 3 bytes are changing. I know a 24-bit representation of color with a byte for each red, green, and blue component is quite common so could these last three bytes be the color of the bulb? Let's look at the colors if I assume the first changing byte is red, the second is green, and the third is blue:



Aha! I can see these color values are the red, green, blue, and orange colors that I had sent to the bulb with its control application! By just looking at a few packets it's trivially easy to reverse engineer this bulb's protocol. It appears that sending a write to the characteristic with handle 0x0028 and providing a value that starts with 0x58010301FF00 and ends with a byte for the red color, green color, and blue color will change the bulb's color.

Before I get too excited about this discovery I need to replicate it myself to confirm the protocol works as I suspect. What the 6 bytes in front of the color represent are a complete mystery and might complicate understanding how the protocol works. Are they a static value that uniquely identifies this write as a color update, could they be a magic value like a session ID the bulb told the app to send ahead of time, or perhaps something else entirely? Trying to replay one of the captured packets myself will help me confirm if I've really figured out the protocol.

Continue on to learn how to use the bluez Bluetooth stack to send BLE packets to the bulb.

Control With Bluez

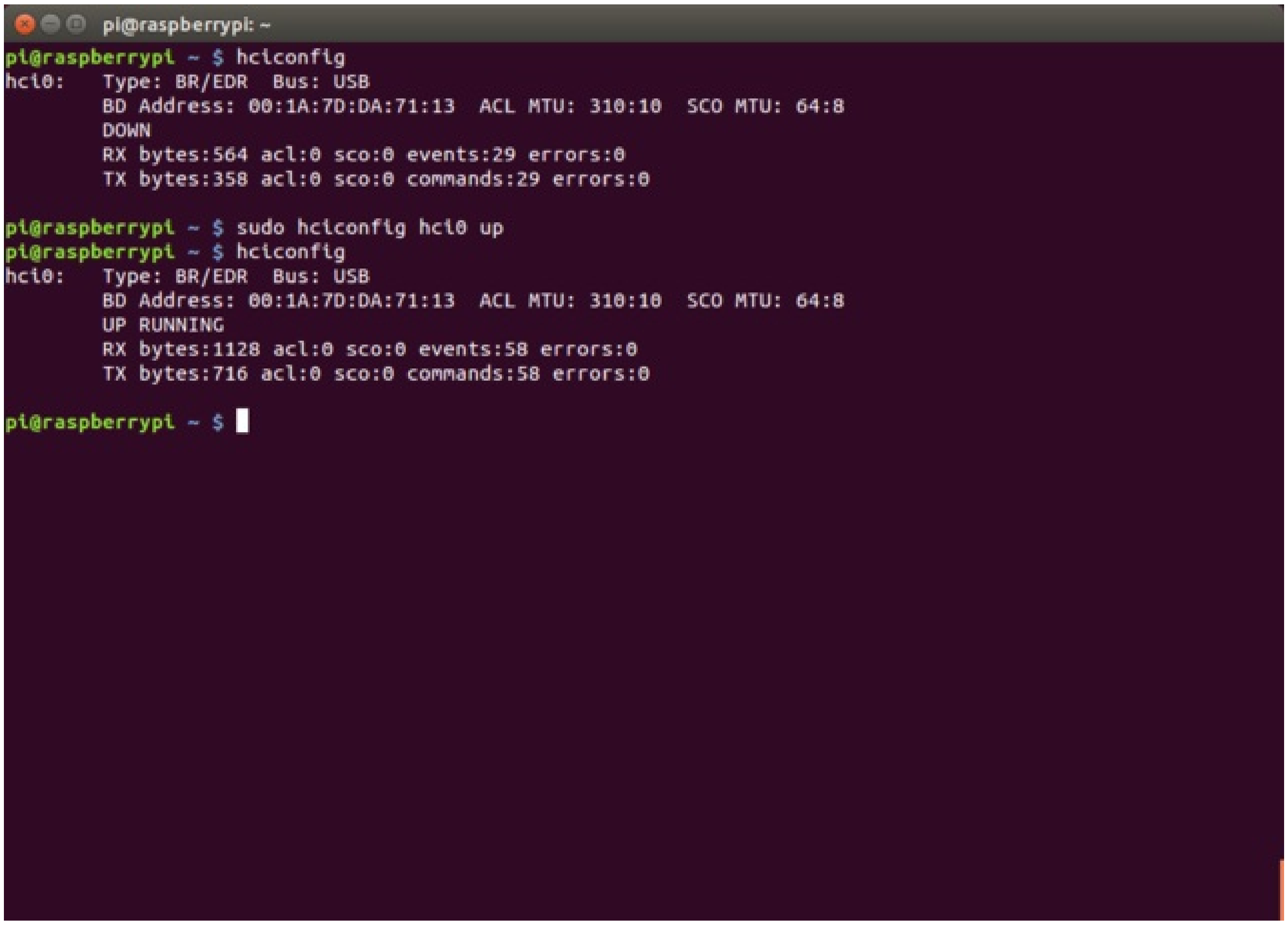
Now for some real fun, I'll try to control the light bulb using a BLE adapter on a computer. I'm going to use a Raspberry Pi, Bluetooth 4.0 USB adapter, and the bluez Bluetooth stack (http://adafru.it/eDE) becuase it's easy to setup and use. Unfortunately there is no cross-platform Bluetooth stack or API that works across Windows, Mac, Linux, etc. so if you want to use a different platform you'll need to look at that platform's Bluetooth Low Energy stack and API.

To setup the Bluetooth dongle and bluez I followed the steps in the setting up section of the Pi Beacon guide here (http://adafru.it/eDF). A couple small changes I made were to download and build the latest version of bluez (5.28 as of the time of this writing) and to manually install bluez's GATT tool. For some reason bluez does not install its GATT tool anymore as this bug notes (http://adafru.it/eDG), however an easy workaround is to manually install it by executing this command inside the bluez source directory after it has been compiled and installed:

```
sudo cp attrib/gatttool /usr/bin/
```

Confirm you can access gatttool by running '**gatttool --help**' (without quotes) to see the usage information of the tool. If you see an error that gatttool can't be found, double check it has been compiled by bluez and it's in to the /usr/bin/ directory.

Now I brought up the Bluetooth USB adapter on the Pi by running the command **hciconfig** to find the name of the adapter (it should be **hci0** assuming it's the only Bluetooth adapter connected to the Pi). Then running '**sudo hciconfig hci0 up**' (without quotes) to turn on the adapter. Finally running **hciconfig** again should show the adapter is in the **UP RUNNING** state as shown below:

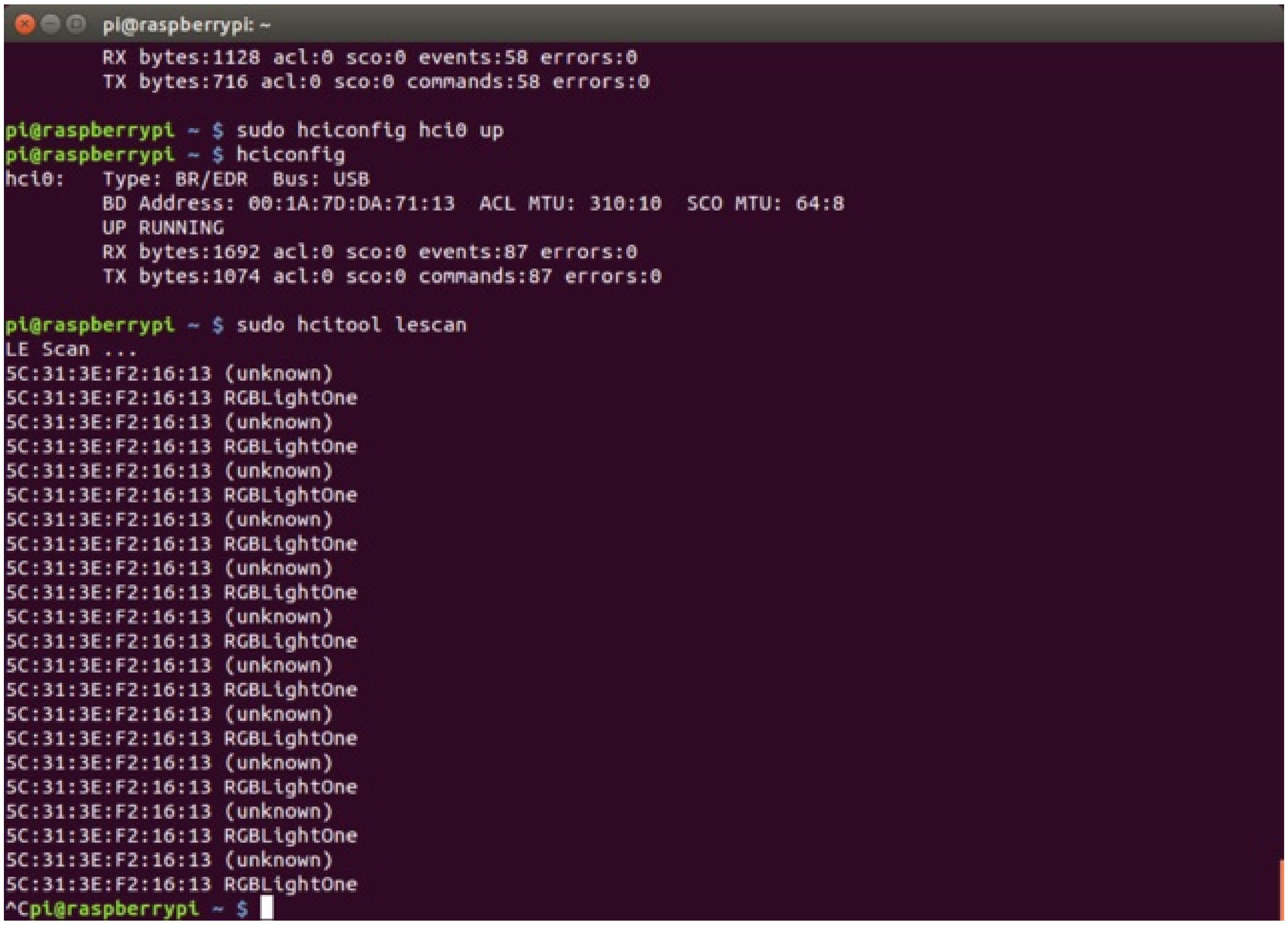


```
pi@raspberrypi ~ $ hciconfig
hci0:	Type: BR/EDR  Bus: USB
	BD Address: 00:1A:7D:DA:71:13  ACL MTU: 310:10  SCO MTU: 64:8
	DOWN
	RX bytes:564 acl:0 sco:0 events:29 errors:0
	TX bytes:358 acl:0 sco:0 commands:29 errors:0

pi@raspberrypi ~ $ sudo hciconfig hci0 up
pi@raspberrypi ~ $ hciconfig
hci0:	Type: BR/EDR  Bus: USB
	BD Address: 00:1A:7D:DA:71:13  ACL MTU: 310:10  SCO MTU: 64:8
	UP RUNNING
	RX bytes:1128 acl:0 sco:0 events:58 errors:0
	TX bytes:716 acl:0 sco:0 commands:58 errors:0

pi@raspberrypi ~ $ 
```

Now that the adapter is up I can scan for BLE devices by running the command:

```
sudo hcitool lescan
```

Information about nearby BLE devices will be displayed. Press **Ctrl-C** to stop the scanning process. Notice from the picture below my light bulb address and name are visible during the scan:



```
pi@raspberrypi: ~
        RX bytes:1128 acl:0 sco:0 events:58 errors:0
        TX bytes:716 acl:0 sco:0 commands:58 errors:0

pi@raspberrypi ~ $ sudo hciconfig hci0 up
pi@raspberrypi ~ $ hciconfig
hci0:   Type: BR/EDR  Bus: USB
        BD Address: 00:1A:7D:DA:71:13  ACL MTU: 310:10  SCO MTU: 64:8
        UP RUNNING
        RX bytes:1692 acl:0 sco:0 events:87 errors:0
        TX bytes:1074 acl:0 sco:0 commands:87 errors:0

pi@raspberrypi ~ $ sudo hcitool lescan
LE Scan ...
5C:31:3E:F2:16:13 (unknown)
5C:31:3E:F2:16:13 RGBLightOne
5C:31:3E:F2:16:13 (unknown)
5C:31:3E:F2:16:13 RGBLightOne
5C:31:3E:F2:16:13 (unknown)
5C:31:3E:F2:16:13 RGBLightOne
5C:31:3E:F2:16:13 (unknown)
5C:31:3E:F2:16:13 RGBLightOne
5C:31:3E:F2:16:13 (unknown)
5C:31:3E:F2:16:13 RGBLightOne
5C:31:3E:F2:16:13 (unknown)
5C:31:3E:F2:16:13 RGBLightOne
5C:31:3E:F2:16:13 (unknown)
5C:31:3E:F2:16:13 RGBLightOne
5C:31:3E:F2:16:13 (unknown)
5C:31:3E:F2:16:13 RGBLightOne
5C:31:3E:F2:16:13 (unknown)
5C:31:3E:F2:16:13 RGBLightOne
5C:31:3E:F2:16:13 (unknown)
5C:31:3E:F2:16:13 RGBLightOne
5C:31:3E:F2:16:13 (unknown)
5C:31:3E:F2:16:13 RGBLightOne
^Cpi@raspberrypi ~ $
```

If you don't see your light bulb in the scan then make sure it is turned on and that no devices are connected to the bulb. Remember **BLE only allows one device to be connected at a time** so if you left the bulb's control application or a BLE GATT app running then it might still be connected to the bulb and preventing you from finding the bulb. Close out of all apps using the bulb and try running the scan again until you see the bulb being advertised.

Next I run the bluez GATT tool to interact with the bulb. I ran this command to start GATT tool's command shell:

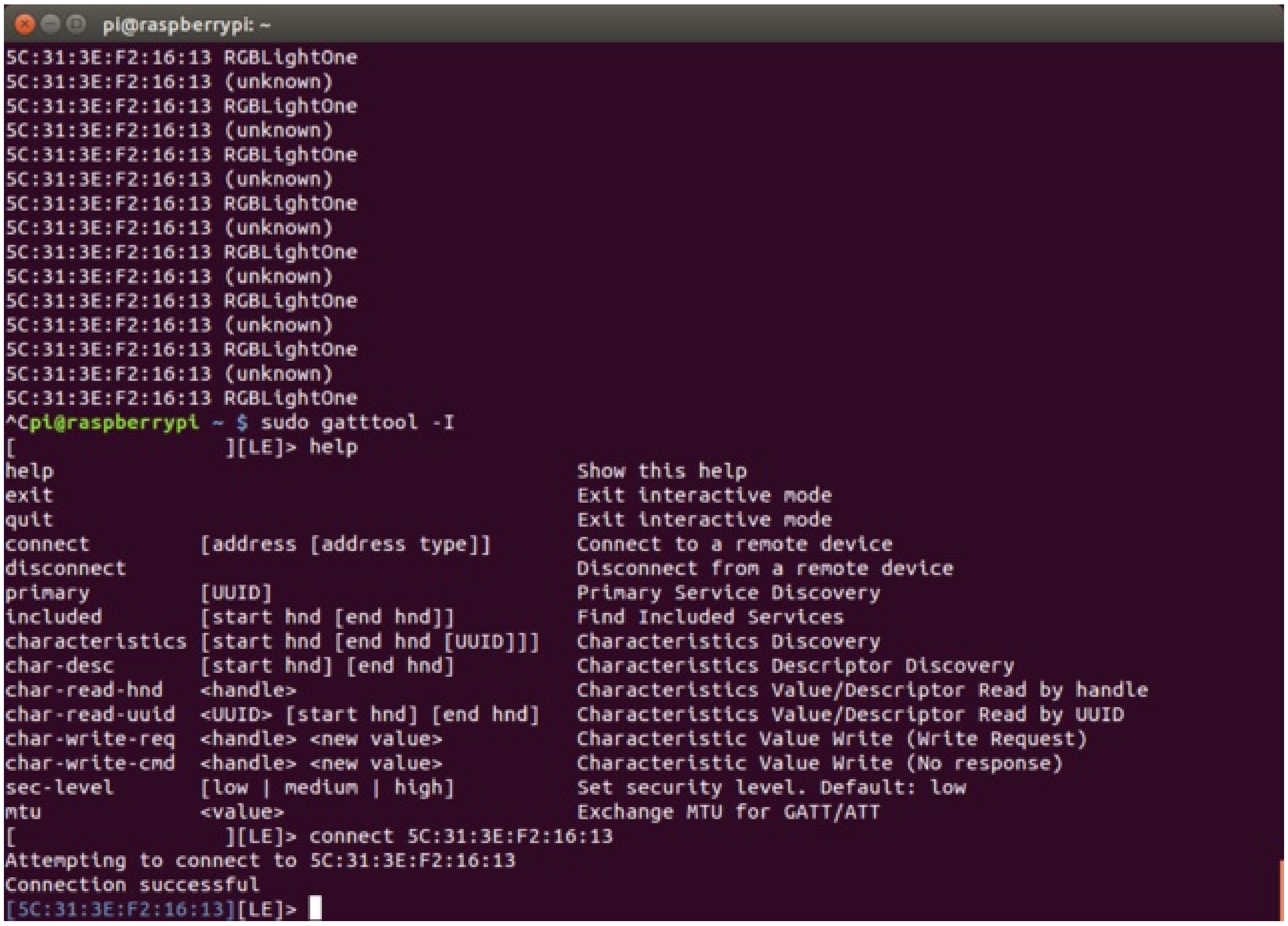
```
sudo gatttool -I
```

A command prompt is shown and I can type **help** and press enter to see a list of commands.

Now I connect to the bulb by issuing a connect command:

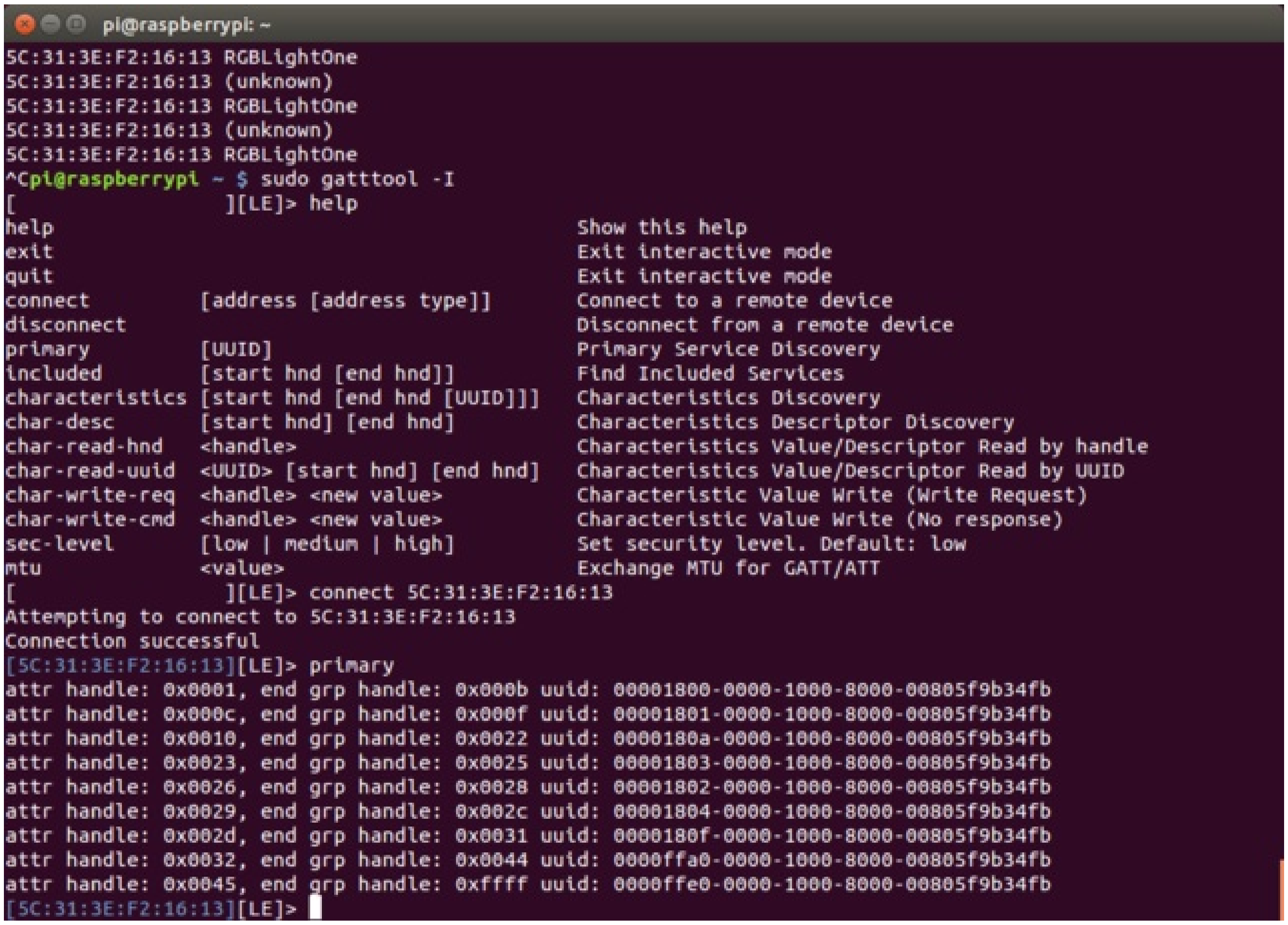
```
connect <bulb address>
```

Where is the address of the bulb that was found with the previous scan command. In my case I ran '**connect 5C:31:3E:F2:16:13**' (without quotes) to connect to my bulb. After a moment a 'Connection successful' message should be displayed like below:



```
5C:31:3E:F2:16:13 RGBLightOne
5C:31:3E:F2:16:13 (unknown)
5C:31:3E:F2:16:13 RGBLightOne
5C:31:3E:F2:16:13 (unknown)
5C:31:3E:F2:16:13 RGBLightOne
5C:31:3E:F2:16:13 (unknown)
5C:31:3E:F2:16:13 RGBLightOne
5C:31:3E:F2:16:13 (unknown)
5C:31:3E:F2:16:13 RGBLightOne
5C:31:3E:F2:16:13 (unknown)
5C:31:3E:F2:16:13 RGBLightOne
5C:31:3E:F2:16:13 (unknown)
5C:31:3E:F2:16:13 RGBLightOne
5C:31:3E:F2:16:13 (unknown)
5C:31:3E:F2:16:13 RGBLightOne
^Cpi@raspberrypi ~ $ sudo gatttool -I
[                 ][LE]> help
help                                           Show this help
exit                                           Exit interactive mode
quit                                           Exit interactive mode
connect         [address [address type]]       Connect to a remote device
disconnect                                     Disconnect from a remote device
primary         [UUID]                         Primary Service Discovery
included        [start hnd [end hnd]]          Find Included Services
characteristics [start hnd [end hnd [UUID]]]   Characteristics Discovery
char-desc       [start hnd] [end hnd]          Characteristics Descriptor Discovery
char-read-hnd   <handle>                       Characteristics Value/Descriptor Read by handle
char-read-uuid  <UUID> [start hnd] [end hnd]   Characteristics Value/Descriptor Read by UUID
char-write-req  <handle> <new value>           Characteristic Value Write (Write Request)
char-write-cmd  <handle> <new value>           Characteristic Value Write (No response)
sec-level       [low | medium | high]          Set security level. Default: low
mtu             <value>                        Exchange MTU for GATT/ATT
[                 ][LE]> connect 5C:31:3E:F2:16:13
Attempting to connect to 5C:31:3E:F2:16:13
Connection successful
[5C:31:3E:F2:16:13][LE]>
```

Once connected to the bulb some commands can be run to examine the bulb in more detail. In particular the **primary** command will list services exposed by the bulb like below:



```
pi@raspberrypi: ~
5C:31:3E:F2:16:13 RGBLightOne
5C:31:3E:F2:16:13 (unknown)
5C:31:3E:F2:16:13 RGBLightOne
5C:31:3E:F2:16:13 (unknown)
5C:31:3E:F2:16:13 RGBLightOne
^Cpi@raspberrypi ~ $ sudo gatttool -I
[                 ][LE]> help
help                                           Show this help
exit                                           Exit interactive mode
quit                                           Exit interactive mode
connect         [address [address type]]       Connect to a remote device
disconnect                                     Disconnect from a remote device
primary         [UUID]                         Primary Service Discovery
included        [start hnd [end hnd]]          Find Included Services
characteristics [start hnd [end hnd [UUID]]]   Characteristics Discovery
char-desc       [start hnd] [end hnd]          Characteristics Descriptor Discovery
char-read-hnd   <handle>                       Characteristics Value/Descriptor Read by handle
char-read-uuid  <UUID> [start hnd] [end hnd]   Characteristics Value/Descriptor Read by UUID
char-write-req  <handle> <new value>           Characteristic Value Write (Write Request)
char-write-cmd  <handle> <new value>           Characteristic Value Write (No response)
sec-level       [low | medium | high]          Set security level. Default: low
mtu             <value>                        Exchange MTU for GATT/ATT
[                 ][LE]> connect 5C:31:3E:F2:16:13
Attempting to connect to 5C:31:3E:F2:16:13
Connection successful
[5C:31:3E:F2:16:13][LE]> primary
attr handle: 0x0001, end grp handle: 0x000b uuid: 00001800-0000-1000-8000-00805f9b34fb
attr handle: 0x000c, end grp handle: 0x000f uuid: 00001801-0000-1000-8000-00805f9b34fb
attr handle: 0x0010, end grp handle: 0x0022 uuid: 0000180a-0000-1000-8000-00805f9b34fb
attr handle: 0x0023, end grp handle: 0x0025 uuid: 00001803-0000-1000-8000-00805f9b34fb
attr handle: 0x0026, end grp handle: 0x0028 uuid: 00001802-0000-1000-8000-00805f9b34fb
attr handle: 0x0029, end grp handle: 0x002c uuid: 00001804-0000-1000-8000-00805f9b34fb
attr handle: 0x002d, end grp handle: 0x0031 uuid: 0000180f-0000-1000-8000-00805f9b34fb
attr handle: 0x0032, end grp handle: 0x0044 uuid: 0000ffa0-0000-1000-8000-00805f9b34fb
attr handle: 0x0045, end grp handle: 0xffff uuid: 0000ffe0-0000-1000-8000-00805f9b34fb
[5C:31:3E:F2:16:13][LE]>
```

The output of the primary command is a raw list of the characteristic handles and service UUIDs implemented by the bulb. This is the same information from earlier when exploring the GATT, but with a bit lower level of detail.

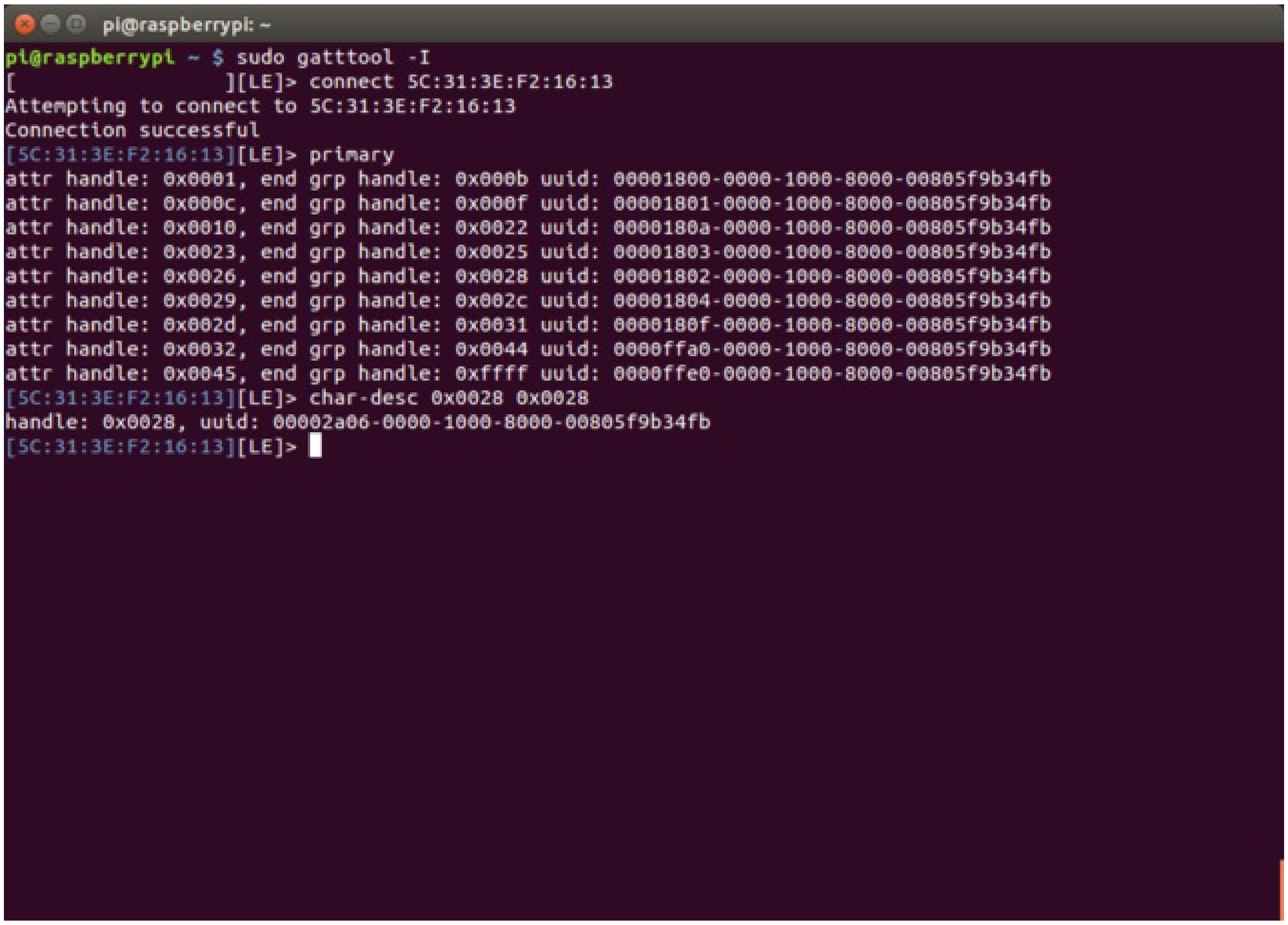
In particular I can see the characteristic handle 0x0028 falls within the range of the 0x1802 service UUID (for UUIDS that have the form 0000xxxx-0000-1000-8000-00805f9b34fb they are officially recognized UUIDs and are abbreviated with the shorter 16-bit UUID). Looking at the Bluetooth services list (http://adafru.it/ddA) the 0x1802 service is the Immediate Alert service (http://adafru.it/eDH), which is typically used for proximity sensors and similar devices. Very odd that the light bulb appears to be using this service to control its color!

Another command that can be run is the **char-desc** command to get details on a particular characteristic. I ran this command to query the 0x0028 characteristic:

```
char-desc 0x0028 0x0028
```

(both 0x0028 values are necessary as the command takes a range of beginning and ending handles)

I saw a response like the following which tells me what UUID represents this characteristic:



```
pi@raspberrypi: ~
pi@raspberrypi ~ $ sudo gatttool -I
[                 ][LE]> connect 5C:31:3E:F2:16:13
Attempting to connect to 5C:31:3E:F2:16:13
Connection successful
[5C:31:3E:F2:16:13][LE]> primary
attr handle: 0x0001, end grp handle: 0x000b uuid: 00001800-0000-1000-8000-00805f9b34fb
attr handle: 0x000c, end grp handle: 0x000f uuid: 00001801-0000-1000-8000-00805f9b34fb
attr handle: 0x0010, end grp handle: 0x0022 uuid: 0000180a-0000-1000-8000-00805f9b34fb
attr handle: 0x0023, end grp handle: 0x0025 uuid: 00001803-0000-1000-8000-00805f9b34fb
attr handle: 0x0026, end grp handle: 0x0028 uuid: 00001802-0000-1000-8000-00805f9b34fb
attr handle: 0x0029, end grp handle: 0x002c uuid: 00001804-0000-1000-8000-00805f9b34fb
attr handle: 0x002d, end grp handle: 0x0031 uuid: 0000180f-0000-1000-8000-00805f9b34fb
attr handle: 0x0032, end grp handle: 0x0044 uuid: 0000ffa0-0000-1000-8000-00805f9b34fb
attr handle: 0x0045, end grp handle: 0xffff uuid: 0000ffe0-0000-1000-8000-00805f9b34fb
[5C:31:3E:F2:16:13][LE]> char-desc 0x0028 0x0028
handle: 0x0028, uuid: 00002a06-0000-1000-8000-00805f9b34fb
[5C:31:3E:F2:16:13][LE]>
```

Going to the Bluetooth characteristic list (http://adafru.it/ddC) I can see this characteristic is the alert level (http://adafru.it/eDI) characteristic. I can see from the characteristic definition that it is normally only takes one byte which is interpreted as a level of alert. However from the protocol sniffing earlier it looks like the bulb has overloaded the alert level characteristic to take 9 bytes, including the 3 bytes of RGB color information--very strange, indeed!

Now for the moment of truth, can I change the color of the bulb by writing to this characteristic based on what was seen from sniffing the protocol? I executed commands like these to write new values to the 0x0028 alert level characteristic:

```
char-write-cmd 0x0028 58010301ff00ff0000
char-write-cmd 0x0028 58010301ff0000ff00
char-write-cmd 0x0028 58010301ff000000ff
char-write-cmd 0x0028 58010301ff00000000
char-write-cmd 0x0028 58010301ff00ffffff
```

Woo hoo! The light bulb changed its color after each command was run! In particular, writing

58010301ff00ff0000 made the bulb turn **red**, **58010301ff0000ff00** turned the bulb **green**, and **58010301ff000000ff** turned the bulb **blue.** Sending RGB colors of 000000 and ffffff turned the bulb off and on at full bright white respectively too. This confirms what I suspected from the protocol reverse engineering, the last 3 bytes of the message represent the red, green, and blue color of the bulb.

I could mix and match colors in between too, for example sending **ffee00** for a yellow color. An HTML color picker (http://adafru.it/eDJ) is helpful for finding color byte values quickly.

To really confirm my understanding of the bulb's protocol I duplicated this GATT tool control process with a second bulb. I turned on a second bulb, scanned for BLE devices again to find the new bulb's address, and connected with GATT tool to change its color. The same characteristic write commands above worked to control the second bulb's color! It appears the 6 bytes at the start of the characteristic write are just a fixed value and luckily do not represent anything meaningful or necessary for changing the bulb's color.

Now that the protocol is understood I had some fun by controlling the light from code. As a simple example I made a python script (http://adafru.it/eDK) to use bluez's GATT tool and cycle through a rainbow of hues (note that if you are a more experienced bluez user you might realize that GATT tool can be controlled from the command line directly, however in my testing I couldn't get GATT tool to control the bulb outside of an interactive session, perhaps because of a bug inside the tool). If you'd like to try yourself you can grab the script from this repository and install its dependencies by executing these commands on the Pi:

```
sudo apt-get update
sudo apt-get install git build-essential python-dev python-pip
sudo pip install pexpect
cd ~
git clone https://github.com/adafruit/BLE_Colorific.git
cd BLE_Colorific
```

Then run the script and provide it the address of the bulb to control as the first parameter. Be sure to run the script as a root user with sudo too. For example to control my bulb I ran:

```
sudo python colorific.py 5C:31:3E:F2:16:13
```

You should see the bulb cycle through all the hues of color. If you'd like to adjust the speed, range of hues, etc. edit the colorific.py file and change some of the variables defined near the top (look at the comments to see what they mean).

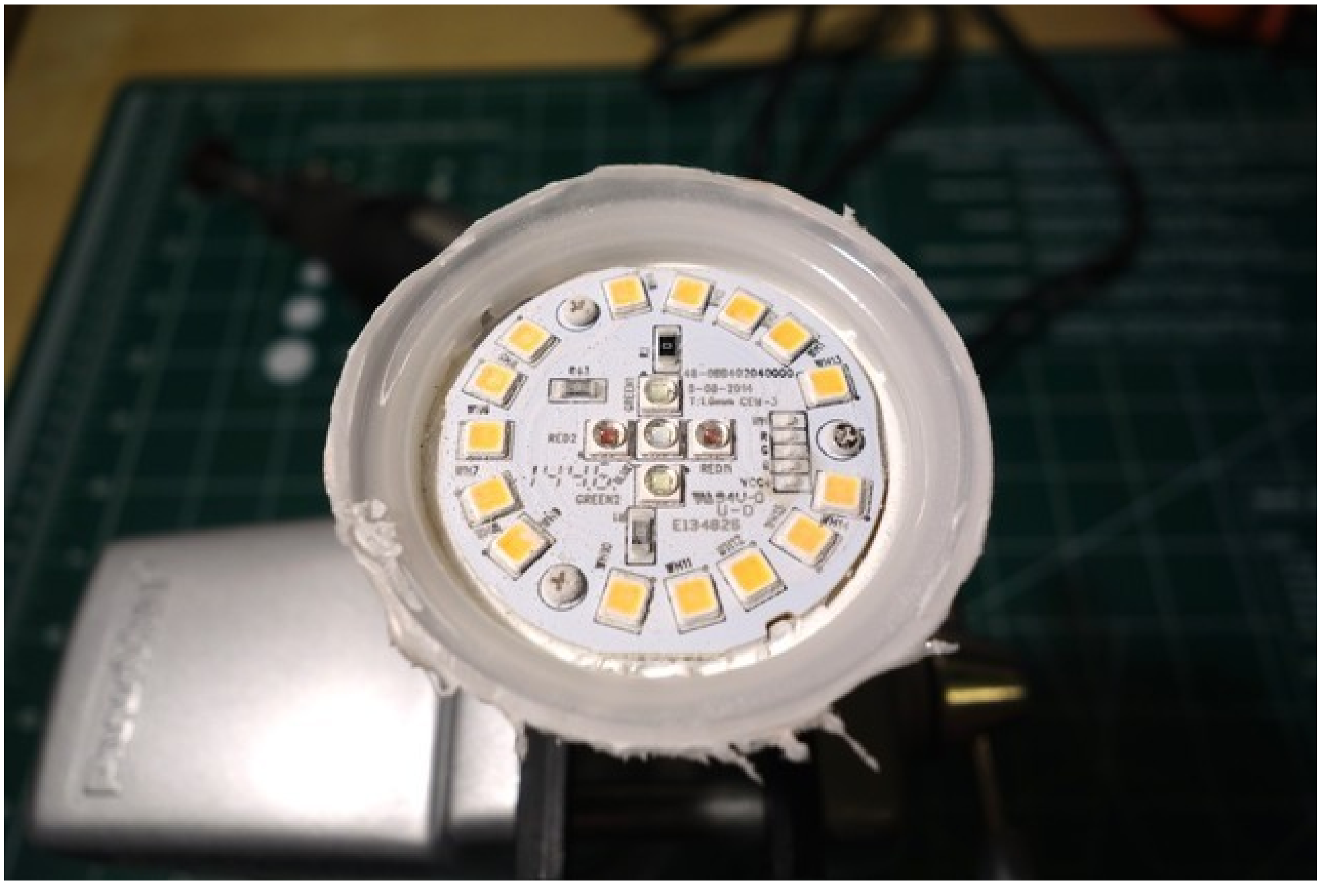
Phew! That was a decent amount of work but it's quite satisfying to have control over the bulb. The sky is the limit as far as interesting things to do with a bulb that you can control yourself over

Bluetooth Low Energy! For example you could make the bulb change color if you get new emails or even based on the local weather forecast. You can animate the bulb like the demo of cycling through hues or do something more interesting like flashing the bulb in time with music. Be creative and turn the bulb into something fun and interesting!

Inside The Bulb

Out of curiosity I decided to cut open one of the bulbs to see if there were more details on what BLE board is being used inside the bulb. From what was found earlier while exploring the bulb's GATT I expect a Texas Instruments CC2540 chip is inside the bulb.

I cut through the plastic diffuser around the bulb with a Dremel and saw a little circuit board with the LEDs:

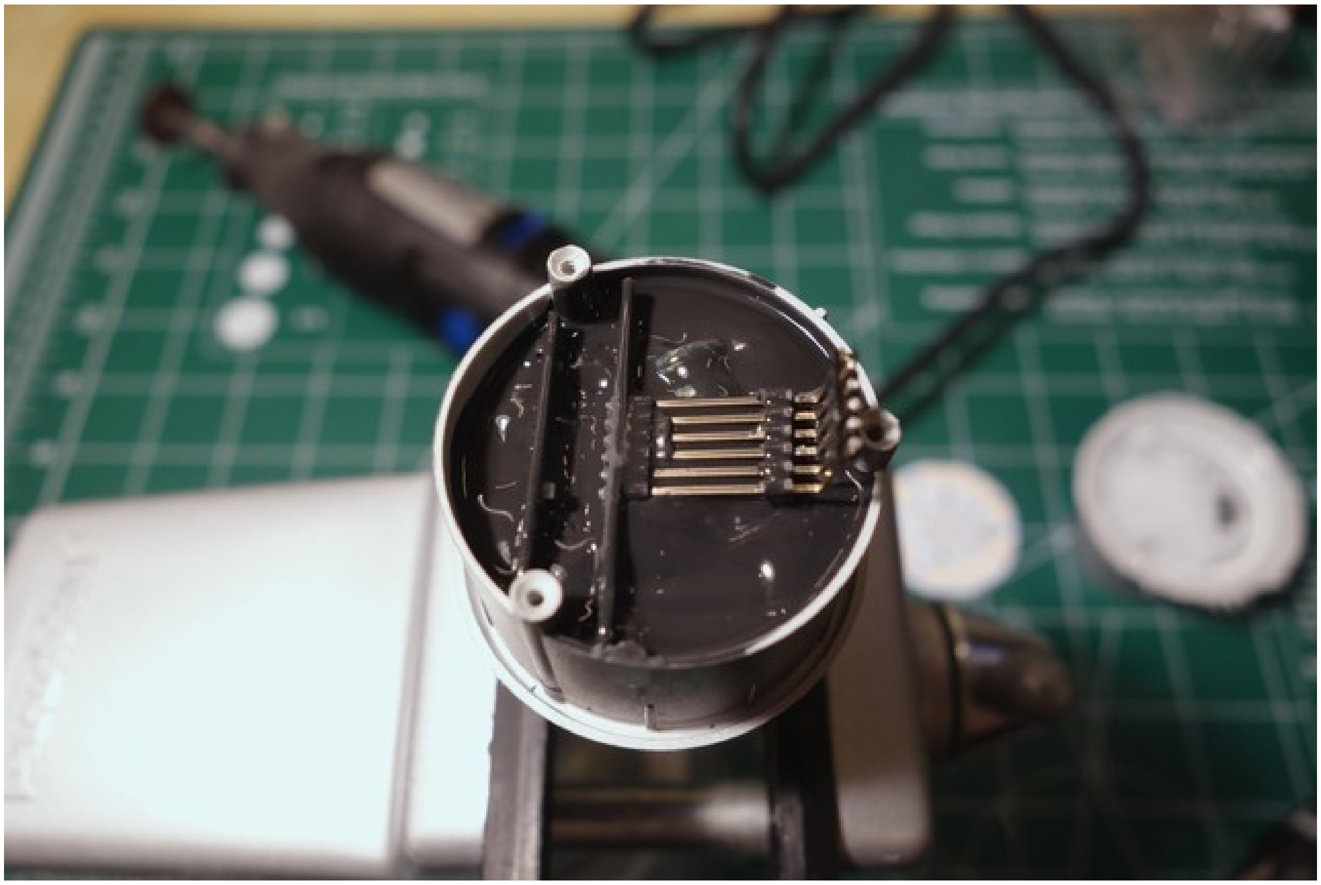


It's interesting to see a ring of white LEDs (soft white actually, somewhere around 3500K I would guess) around the perimeter of the board. In the middle are the colored LEDs, with two green, two red, and one blue LED. Presumably when the bulb shows pure white it turns on the outer white LEDs to have a more pleasing color as the red, green, and blue LEDs would be colder and less pleasant for indoor lighting.

Also interesting is the inclusion of only one blue LED vs. two of the other primary color LEDs. I suspect only one blue LED is necessary because humans perceive the color blue with more intensity than other colors. This is why blue LEDs on electronic gadets are so bright and annoying in a dark room!

Going further I unscrewed the board from the bulb and pried it off with a screwdriver. It turns there's a row of headers that are soldered to the board, but the solder snapped off with a bit of force.

Once unscrewed the top of the bulb base lifts off to reveal:



Solid silicone gunk, yuck! The edges of a couple circuit boards are visible poking out near the left side, but unfortunately the silicone makes it impossible to get to the boards easily. Removing the silicone could get quite messy so this is as far as the bulb diassembly will go.

Overall for the price these 'Colorific!' smart light bulbs are quite interesting. From what I found during the reverse engineering it looks like they're probably using a common TI CC2540 BLE SoC and perhaps even based the bulb's code on some of TI's examples like a proximity alert sensor. If you're looking for an inexpensive light bulb that you can control yourself, check out these bulbs!