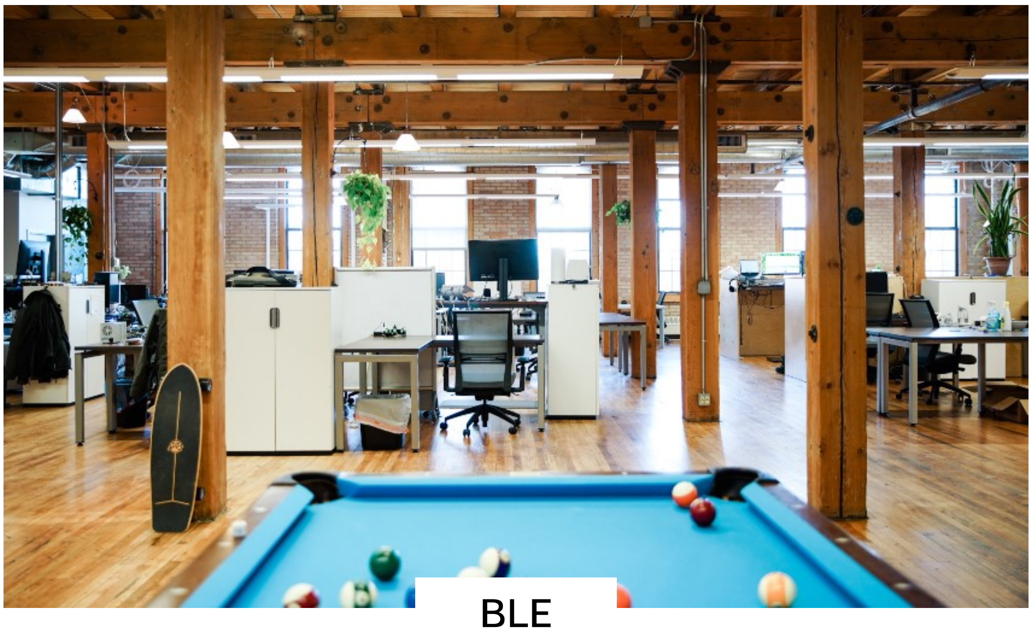
PunchThrough



BLE

Creating a BLE Peripheral with BlueZ

June 29, 2020 By: Andy Lee

While I'm fairly well-versed in working with BLE on an embedded device, I have so far had little experience with using BLE at a higher level. To help remedy this, I underwent the task of creating a BLE peripheral running a GATT server on a Raspberry Pi 3+.

This is not intended to be an introduction on BLE and assumes that the reader has some idea of what services, characteristics, and advertising are. If you don't know what those words mean, you should first check out Adafruit's article on BLE.

Julian, one of our Mobile Software Engineers, showed us how to use bleno to quickly spin up a BLE peripheral. While the official forks of Noble/bleno seem to have been abandoned, they have been picked up by abandonware. They're a great wrapper around Bluetooth things. However, if you're looking for more in-depth insights into directly working with BlueZ, or for those who don't want to use a third party library to manage their BLE connections, this guide is for you.

What is BlueZ?

BlueZ is the Bluetooth stack for Linux. It handles both Bluetooth BR/EDR as well as BLE. To communicate with BlueZ, we'll be using something called D-Bus.

What is D-Bus?

Before we get into BlueZ, we'll want to have a cursory understanding of d-bus and how we use it to talk to BlueZ. D-Bus is essentially a message bus that allows different processes running on a Linux system to talk to each other.

We're going to build our peripheral directly using BlueZ's D-Bus API. D-Bus is an Inter-process Communication (IPC) method that allows us to make Remote Procedure Calls (RPC) between different processes. In this case, the application we'll be creating will be talking to the BlueZ process to tell BlueZ what we need it to do.

There are two types of buses in D-Bus: system and session. We're only interested in talking on the *system* bus where the bluetooth service is perched on. There's only one system bus on your machine, but there can be multiple session buses running on a system at the same time.

Within D-Bus, there's the concept of `objects` and `interfaces`.

- **Objects** are concrete things that you can act on. Objects can implement one or more interfaces. Objects owned by their respective process and are referenced like file paths. They look something like this: `/com/punchthrough/office/desk/andy`.
- **Interfaces** are what an object adheres to. Interfaces are composed of `methods` (methods other processes can call), `signals` (events other processes can listen for), and `properties` (variables other processes can read/write from/to). An interface looks something like this: `com.punchthrough.desk`.

Namespacing with your domain will help prevent any possible collisions with external code.

Getting Familiar with BlueZ and D-Bus

The tool we'll be using to familiarize ourselves BlueZ is called `bluetoothctl`. It comes bundled with BlueZ. If you want to see what's happening on the system bus, you can run `sudo dbus-monitor --system "destination='org.bluez'" "sender='org.bluez'"` in another terminal window. This allows you to watch as calls are made to/from the BlueZ daemon.

Let's start by running `bluetoothctl` and typing `help`. This will show the commands available to you. The first thing we need to do is power on our adapter. You can do this with the `power on` command.With the adapter powered up, let's set some advertising data so we can see our BLE peripheral. To do this, access the `advertise` menu of `bluetoothctl` by typing `menu advertise`. This should show the `advertise` help menu. We're going to want to set up manufacturing data, and service uuids in our advertising packet.

Set manufacturer data:

```
[bluetooth]# manufacturer 0xffff 0x12 0x34
```

Set the device's local name:

```
[bluetooth]# name myDevice
```

At this point you could also set other pieces of data in the advertising packet with `uuids`, service, or data menu items.

Now we can back out of the advertise menu and start advertising:

```
[bluetooth]# back[bluetooth]# advertise on
```

Open up your BLE testing app of choice and you should see your device advertising with the info you set.

Cool, but if we connect to it, we only have the generic access and generic attribute services available to us. To make things a little more interesting, we can use `bluetoothctl` to add our own service/characteristic.

To do that, we want to type `menu gatt` to get to the gatt submenu. We'll set up a single service with two characteristics. One read only and one read/write.

First, create the service. I used a full 128bit uuid here, but for the services or characteristics, you can use either the full value or 16/32 bit shortened ones.

```
[bluetooth]# register-service e2d36f99-8909-4136-9a49-
d825508b297b
```

When prompted if this should be the primary service, say yes.

Then create our characteristics:

```
[bluetooth]# register-characteristic 0x1234 read
```

Enter the initial value you want for that characteristic:

```
[bluetooth]# register-characteristic 0x4567 read,write
```

The above calls are just creating objects on the system bus. The objects are owned by `bluetoothctl`. We now want to tell BlueZ about the service and characteristics we just created. To do that, we run:

```
[bluetooth]# register-application
```

Once the application is registered, you'll notice that when you connect to the device, you can now see the newly added services. You can read and write to them just as you would on any other BLE characteristic. If you have the `dbus-monitor` terminal open, you can see the messages flying back and forth for each read/write.

While the above exercise is helpful to see the individual steps that are needed to tell BlueZ to do what we want, it's not very scalable to an actual application. To do that, we're going to look at what it takes to make a python application that uses the same d-bus apis that `bluetoothctl` does.

An Application for a "Real" Problem

Instead of coming up with a "Hello world" style application that just shows you all that we can dump some info on the screen, I instead decided to come up with an application that did something *slightly* more useful.

Backstory

I recently upgraded my espresso machine. While the machine was an upgrade in many regards, it lacked the ability to turn itself on via a timer. That trusty auto-on feature of my previous one was sorely missed. At 5:30 AM, every extra minute I get to stay in bed is one that I want. And, when it takes ~20 minutes for the machine to warm up, I needed to do something to get those minutes back.

So, I did what anyone else in my shoes would do. I made an interface board, hooked up some wires, and attached a Raspberry Pi to give it a web interface. I could now control the machine from anywhere — as well as set some timers on it to turn on when I tell it to.

Now, this works fine and dandy if I happen to have an HTTP client laying around. And what kind of self respecting software engineer wouldn't? But what if my wifi goes down or I only have BLE available to me?…

This "problem" ended up being a great excuse to write up how a user can use BlueZ to turn your linux machine into a full blown BLE peripheral running a GATT server. This code was written and created on a Raspberry Pi 3+ running BlueZ 5.53, but it should be mostly portable to any recent-ish BlueZ stack.

Creating a Python Application

The full code for this project is available here. The `ble.py` file is mostly based off of the BlueZ examples.

Starting at `main`, the first thing that we need to do is get a reference to the system dbus so that we can start talking/listening on it

Note: There are some main-loop things that need to be set up throughout the codebase, but we'll consider those out of scope for this. You can just copy what's in the sample code.

```
# get the system busbus = dbus.SystemBus()
```

Now that we have a reference to that bus, we can listen in on it and talk with other processes that are sitting on the bus. But before we start talking to another process, we need to know *how* to talk to them. To do that, we'll need a little understanding of how D-Bus works.

The first thing we'll want to do is power on our actual BLE adapter. Find a BlueZ object that implements the `GattManager1` interface. That interface is described here and is the interface that we'll use to set up our GATT server.

This is what `find_adapter()` is doing in our code. It asks BlueZ for all of the objects that it's managing and then looks for one that implements the `org.bluez.GattManager1` interface. On my system, that object is located at `/org/bluez/hci0`. Find that object and use the `bus.get_object(BLUEZ_SERVICE_NAME, "/org/bluez/hci0")` to get what's called a ProxyObject. The proxy object is our local reference to the remote BlueZ object that we can use to call methods on and receive signals from the remote object.

At this point, we could use a tool like `gdbus` on the command line to look at that object and see what actual interfaces it implements. This can be done with `gdbus introspect -y -d "org.bluez" -o "/org/bluez/hci0"`. This tells gdbus that we want to look at our adapter object `-o "/org/bluez/hci0"` that is managed by the BlueZ service `-d "org.bluez"` which is on the system bus -y. I trimmed the output of some things that we don't care about, but the output should look something like this:

```
node /org/bluez/hci0 {
  interface org.bluez.Adapter1 {
    methods:
      StartDiscovery();
      SetDiscoveryFilter(in  a{sv} properties);
      StopDiscovery();
      RemoveDevice(in  o device);
      GetDiscoveryFilters(out as filters);
      ConnectDevice(in  a{sv} properties);
    signals:
    properties:
      readonly s Address = 'B8:27:EB:FD:26:69';
      readonly s AddressType = 'public';
      readonly s Name = 'raspberrypi';
      readwrite s Alias = 'raspberrypi';
      readonly u Class = 0;
      readwrite b Powered = true;
      readwrite b Discoverable = false;
      readwrite u DiscoverableTimeout = 180;
      readwrite b Pairable = true;
      readwrite u PairableTimeout = 0;
      readonly b Discovering = false;
      readonly as UUIDs =
['00001801-0000-1000-8000-00805f9b34fb', '0000110e-
0000-1000-8000-00805f9b34fb',
'00001200-0000-1000-8000-00805f9b34fb', '0000110c-
0000-1000-8000-00805f9b34fb',
'00001800-0000-1000-8000-00805f9b34fb'];
      readonly s Modalias = 'usb:v1D6Bp0246d0532';
  };
  interface org.freedesktop.DBus.Properties {
    methods:
      Get(in  s interface,
          in  s name,
          out v value);
      Set(in  s interface,
          in  s name,
          in  v value);
      GetAll(in  s interface,
             out a{sv} properties);
    signals:
      PropertiesChanged(s interface,
                        a{sv} changed_properties,
                        as invalidated_properties);
    properties:
  };
  interface org.bluez.GattManager1 {
    methods:
      RegisterApplication(in  o application,
                          in  a{sv} options);
      UnregisterApplication(in  o application);
    signals:
    properties:
  };
  interface org.bluez.LEAdvertisingManager1 {
    methods:
          RegisterAdvertisement(in  o advertisement,
                                in  a{sv} options);
          UnregisterAdvertisement(in  o service);
        signals:
        properties:
          readonly y ActiveInstances = 0x03;
          readonly y SupportedInstances = 0x02;
          readonly as SupportedIncludes = ['tx-power',
'appearance', 'local-name'];
      };
    };
```

This tells us that our adapter object implements these interfaces:

1. `org.bluez.Adapter1`
2. `org.freedesktop.DBus.Properties`
3. `org.bluez.GattManager1`
4. `org.bluez.LEAdvertisingManager1`

To power the adapter on, access the `Powered` property of the `org.bluez.Adapter1` interface. To do that, take the adapter object that we created and use the `org.freedesktop.DBus.Properties` interface on the properties of the `org.bluez.Adapter1` interface and set its value to `true`.

```
 adapter_props = dbus.Interface(adapter_obj,
"org.freedesktop.DBus.Properties")
 adapter_props.Set("org.bluez.Adapter1", "Powered",
dbus.Boolean(1))
```

With our adapter on, let's set up our advertising data. We must first get an interface for our adapter proxy object that implements the adapters `org.bluez.LEAdvertisingManager1` interface.

```
ad_manager = dbus.Interface(adapter_obj,
LE_ADVERTISING_MANAGER_IFACE)
```

At this point, we need to create our advertisement. Use the `Advertisement` class in the `ble` module to create a

VivaldiAdvertisement.

Note: the model of the espresso machine is a Vivaldi.

```python
class VivaldiAdvertisement(Advertisement):
    def __init__(self, bus, index):
        Advertisement.__init__(
            self, bus, index, "peripheral", base_path = "/com/punchthrough/advertisement/espresso"
        )
        self.add_manufacturer_data(
            0xFFFF, [0x70, 0x74],
        )
self.add_service_uuid(VivaldiS1Service.ESPRESSO_SVC_UUID) # Our service uuid we'll advertise

        self.add_local_name("Vivaldi")
        self.include_tx_power = True
```

The `Advertisement` class in `ble.py` takes care of creating our object on the d-bus that implements the `org.bluez.LEAdvertisement1`. We then tell register our advertising object with BlueZ.

```python
advertisement = VivaldiAdvertisement(bus, 0)

ad_manager.RegisterAdvertisement(
    advertisement.get_path(),
    {},
    reply_handler=register_ad_cb,
    error_handler=register_ad_error_cb,
)
```

This tells BlueZ where our advertising object is (`advertisement.get_path()`) and sets up our reply or error handlers to get the response of the advertisement registration.

Next, let's set up our services and characteristics. Use the `Application` class in `ble.py`. This creates an object, owned by our process, on the system dbus that implements the `org.bluez.GattApplication1 interface`.

We also need to create a service for this application. To do that, we'll first create our characteristics. This is where the actual business logic for read/writes to characteristics comes into play. For this service, I have three things I want to control on my espresso machine.

1. Power state of the machine
2. Power state of the steam boiler
3. How many minutes the machine will stay on before it automatically shuts off.

These three features are already supported by the existing web API.

To create the characteristics, use the `Characteristic` class provided by `ble.py`. This takes care of creating the objects that represent the characteristics on the system dbus that implement the `org.bluez.GattCharacteristic1` interface.

We'll look in-depth at just the characteristic for powering the machine on and off as the others are mostly the same. First, we have some setup stuff. This code is setting the uuid for our characteristic as well as the description. The description will be set as the Characteristic User Description Descriptor. The `State` enum is used to enumerate the states the machine can be in. `__init__` creates our actual object. It's also where we set the permissions for the characteristic and its initial value.

```python
class PowerControlCharacteristic(Characteristic):
    uuid = "4116f8d2-9f66-4f58-a53d-fc7440e7c14e"
    description = b"Get/set machine power state {'ON', 'OFF', 'UNKNOWN'}"

    class State(Enum):
        on = "ON"
        off = "OFF"
        unknown = "UNKNOWN"

        @classmethod
        def has_value(cls, value):
            return value in cls._value2member_map_

    def __init__(self, bus, index, service):
        Characteristic.__init__(
            self, bus, index, self.uuid, ["encrypt-read", "encrypt-write"], service,
        )

        self.value = bytearray(self.State.unknown.value, "utf-8")
        self.add_descriptor(CharacteristicUserDescriptionDescriptor(bus, 1, self))
```

Next up, we have our ReadValue method. This is the actual method that is called when something reads your characteristic value. Any exceptions thrown in here will bubble back up to the calling device as an error. So, you'll want to be explicit with how you handle them. What we're doing, in this case, is using requests to make an HTTP request to our existing web app to get the state of the machine. We then return that state as a bytearray. This is the actual binary data that is sent back over BLE as the response to the read request.

```
def ReadValue(self, options):
    try:
        res = requests.get(VivaldiBaseUrl + "/vivaldi")
        self.value = bytearray(res.json()["machine"], encoding="utf8")
    except Exception as e:
        self.value = bytearray(self.State.unknown.value, encoding="utf8")

    return self.value
```

Last up for this characteristic is implementing the `WriteValue` method. This is what is called when something attempts to write to the characteristic over BLE.

```
def WriteValue(self, value, options):
    cmd = bytes(value).decode("utf-8")
    if self.State.has_value(cmd):
        # write it to machine
        data = {"cmd": cmd.lower()}
        try:
            res = requests.post(VivaldiBaseUrl + "/vivaldi/cmds", json=data)
            self.value = value
        except Exceptions as e:
            logger.error(f"Error updating machine state: {e}")
            raise Exception
    else:
        logger.info(f"invalid state written {cmd}")
        raise NotPermittedException
```

In this code, we use requests to post some data to our local webserver that sets the actual power state of the espresso machine.

The other characteristics that control the boiler and auto off time are mostly the same with just some different HTTP endpoints called. We can now use the Service class provided by `ble.py` that will take care of creating our actual service and characteristic objects on the system dbus.

```
class VivaldiService(Service):
    ESPRESSO_SVC_UUID = "12634d89-d598-4874-8e86-7d042ee07ba7"

    def __init__(self, bus, index):
        Service.__init__(self, bus, index, self.ESPRESSO_SVC_UUID, True)
        self.add_characteristic(PowerControlCharacteristic(bus, 0, self))
        self.add_characteristic(BoilerControlCharacteristic(bus, 1, self))
        self.add_characteristic(AutoOffCharacteristic(bus, 2, self))
```

We can now create the application, and add our new service to it.

```
app = Application(bus)
app.add_service(VivaldiS1Service(bus, 2))
service_manager = dbus.Interface(adapter_obj, GATT_MANAGER_IFACE)
service_manager.RegisterApplication(
    app.get_path(),
    {},
    reply_handler=register_app_cb,
    error_handler=[register_app_error_cb],
)
```

At this point, BlueZ now knows about our advertising data, and our services and characteristics.

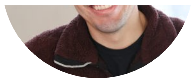
You can also create custom agents if you want more control over the pairing/bonding process. Creating agents works the same way as most other things we've done. You create the custom agent object, and then register it with BlueZ.

Additional Resources

Debugging things when they're not working or you're trying to learn how they work can be pretty hard without visibility into the underlying factors of the system. I mentioned `gdbus` and `dbus-monitor` above for inspecting and monitoring the dbus. Here are some other tools/resources you may find relevant.

- If you're more comfortable with a GUI, d-feet is another great GUI tool for exploring D-Bus.
- A good resource for just D-Bus things is the d-bus tutorial. It has a lot of great information.
- For exploring dbus within python, the dbus-python docs are a good starting point.
- Another tool that's more BLE specific is btmon. It can show you the lower level HCI activity related to the bluetooth module.

All of the source code for the python application can be found here.



Andy Lee

Principal Firmware Engineer at Punch Through. A knowledgable and kind mentor to other team members. While off the job, he enjoys mountain biking, finding new places for lunch, and spending time with his wife and two children.

RECENT POSTS

Bluetooth Module vs Chip-Down: What is Best for Your Design?

Matt Dunham

Architecting iOS and Android Apps for IoT BLE Systems

Gretchen Walker

Meet the Team – Mike Waddick

Erin Moore

The Key to Creating a Growth Culture: Our Approach to Personal Development

Punch Through

HOME

PORTFOLIO

ABOUT

CAREERS

BLOG

CONTACT

CONTACT

MINNEAPOLIS

123 N. 3rd St,
Suite 500
Minneapolis, MN
55401

info@punchthrough.com