Bluetooth for Linux Developers Study Guide

Developing LE Central Devices using Python

Release : 1.0.1

Document Version: 1.0.0

Last updated : 16th November 2021

Contents

1. Revision History

Version	Date	Author	Changes
1.0.0	16th November 2021	Martin Woolley Bluetooth SIG	**Release:** Initial release. **Document**: This document is new in this release.

2. Introduction

There is no strict correlation between the GAP roles *Peripheral* and *Central* and the GATT roles of client and server. Any of the four possible permutations is allowed by the Bluetooth Core Specification, so a GAP Peripheral could be either a GATT client or a GATT server. Typically though, a Peripheral is also a GATT server and that's the combination of roles that we're assuming in this module.

We'll learn how to develop applications that act as Bluetooth LE Central devices on Linux using Python. We'll examine and learn about the most important use cases for Central devices including device discovery, connecting and disconnecting, service discovery, reading and writing characteristics and enabling and handling notifications. You'll be given the opportunity to complete exercises where you will write code that exhibits specified functionality. Just like in the real world!

This module builds upon the knowledge gained in module 03. If you haven't gone through module 03 and are new or relatively new to using D-Bus then you really should before continuing with this module.

For the practical work in this module, you'll need a Linux computer with a BlueZ stack to run your code on of course but you'll also need a Bluetooth LE Peripheral device to test against. The choice is yours if you're prepared to customise some of the exercises to allow them to work with your test device but these are the details of the device used to create and test the exercises originally:

- A BBC micro:bit running the Temperature Service and the LED Service.

Source code for the GUI development tool for micro:bit, MakeCode is provided along with a binary hex file which can be directly installed into a micro:bit. Pairing is not required.

3. Device Discovery

Device discovery is the procedure whereby a Bluetooth LE Central device can identify the list of Bluetooth LE Peripheral devices that are in range, advertising and optionally, seem likely to be of a type which is relevant to the scanning device or application.

3.1 Device Discovery and BlueZ

```
signal time=1635330796.572439 sender=:1.15 -> destination=(null destination) serial=114 path=/;
interface=org.freedesktop.DBus.ObjectManager; member=InterfacesAdded
  object path "/org/bluez/hci0/dev_00_F2_26_94_4F_D4"
  array [
    dict entry(
      string "org.freedesktop.DBus.Introspectable"
      array [
      ]
    )
    dict entry(
      string "org.bluez.Device1"
      array [
        dict entry(
          string "Address"
```

```
      variant             string "00:F2:26:94:4F:D4"
    )
    dict entry(
      string "AddressType"
      variant             string "random"
    )
    dict entry(
      string "Alias"
      variant             string "00-F2-26-94-4F-D4"
    )
    dict entry(
      string "Paired"
      variant             boolean false
    )
    dict entry(
      string "Trusted"
      variant             boolean false
    )
    dict entry(
      string "Blocked"
      variant             boolean false
    )
    dict entry(
      string "LegacyPairing"
      variant             boolean false
    )
    dict entry(
      string "RSSI"
      variant             int16 -81
    )
    dict entry(
      string "Connected"
      variant             boolean false
    )
    dict entry(
      string "UUIDs"
      variant             array [
          string "0000fd6f-0000-1000-8000-00805f9b34fb"
        ]
    )
    dict entry(
      string "Adapter"
      variant             object path "/org/bluez/hci0"
    )
    dict entry(
      string "ServiceData"
      variant             array [
          dict entry(
            string "0000fd6f-0000-1000-8000-00805f9b34fb"
            variant                 array of bytes [
                1a 11 5d 6a 7d 1e 39 f3 ff 04 3e 25 23 d6 b2 00
```

```
                    e6 3c 74
                ]
            )
          ]
        )
        dict entry(
          string "ServicesResolved"
          variant             boolean false
        )
      ]
    )
    dict entry(
      string "org.freedesktop.DBus.Properties"
      array [
      ]
    )
  ]


signal time=1635330828.891484 sender=:1.15 -> destination=(null destination) serial=120 path=/;
interface=org.freedesktop.DBus.ObjectManager; member=InterfacesRemoved
  object path "/org/bluez/hci0/dev_2C_48_35_90_5D_6E"
  array [
    string "org.freedesktop.DBus.Properties"
    string "org.freedesktop.DBus.Introspectable"
    string "org.bluez.Device1"
  ]
```

when *exactly* is this being emitted by BlueZ?

```
------------------------------
NEW path  : /org/bluez/hci0/dev_79_7C_BE_4B_F7_C1
NEW bdaddr:  79:7C:BE:4B:F7:C1
NEW RSSI  :  -91
------------------------------
CHG path  : /org/bluez/hci0/dev_56_A3_39_A2_B4_F7
CHG bdaddr:  56:A3:39:A2:B4:F7
CHG RSSI  :  -90
------------------------------
DEL bdaddr:  5B:10:9C:9B:A2:66
DEL bdaddr:  76:12:F0:9F:3B:6B

pi@raspberrypi:~/projects/ldsg/solutions/python $ python3 client_discover_devices.py 60000
Listing devices already known to BlueZ:
EXI path  :  /org/bluez/hci0/dev_46_15_A5_70_80_D6
EXI bdaddr:  46:15:A5:70:80:D6
------------------------------
EXI path  :  /org/bluez/hci0/dev_60_6A_73_55_9D_57
EXI bdaddr:  60:6A:73:55:9D:57
------------------------------
EXI path  :  /org/bluez/hci0/dev_7C_77_22_C0_6E_82
EXI bdaddr:  7C:77:22:C0:6E:82
------------------------------
EXI path  :  /org/bluez/hci0/dev_40_49_0F_3F_A8_2A
EXI bdaddr:  40:49:0F:3F:A8:2A
------------------------------
EXI path  :  /org/bluez/hci0/dev_54_D6_41_02_E9_F1
EXI bdaddr:  54:D6:41:02:E9:F1
------------------------------
EXI path  :  /org/bluez/hci0/dev_7F_14_58_0C_3E_95
EXI bdaddr:  7F:14:58:0C:3E:95
------------------------------
EXI path  :  /org/bluez/hci0/dev_3A_2D_EE_D1_62_7C
```

```
EXI bdaddr:  3A:2D:EE:D1:62:7C
------------------------------
Scanning
CHG path  : /org/bluez/hci0/dev_60_6A_73_55_9D_57
CHG bdaddr:  60:6A:73:55:9D:57
CHG RSSI  :  -80
------------------------------
CHG path  : /org/bluez/hci0/dev_7C_77_22_C0_6E_82
CHG bdaddr:  7C:77:22:C0:6E:82
CHG RSSI  :  -79
------------------------------
CHG path  : /org/bluez/hci0/dev_3A_2D_EE_D1_62_7C
CHG bdaddr:  3A:2D:EE:D1:62:7C
CHG RSSI  :  -83
------------------------------
CHG path  : /org/bluez/hci0/dev_7F_14_58_0C_3E_95
CHG bdaddr:  7F:14:58:0C:3E:95
CHG name  :  [LG] webOS TV UJ750V
CHG RSSI  :  -81
------------------------------
CHG path  : /org/bluez/hci0/dev_54_D6_41_02_E9_F1
CHG bdaddr:  54:D6:41:02:E9:F1
CHG RSSI  :  -93
------------------------------
CHG path  : /org/bluez/hci0/dev_46_15_A5_70_80_D6
CHG bdaddr:  46:15:A5:70:80:D6
CHG RSSI  :  -95
------------------------------
```

In a Linux BlueZ environment, due to the overall architecture, device discovery is a little more complicated than it is in other environments. On platforms like Android and iOS, an API is used to initiate scanning, perhaps with some filtering criteria included. Callbacks or similar are used to deliver details of discovered devices in close to real time to the scanning device and those reported are those that are in range and advertising right now. This is as you'd expect. But with BlueZ, it's not like that in a number of ways.

Review Figure 2 in module 03 to remind yourself of the architecture and note the key fact that BlueZ supports one-to-many applications that use Bluetooth *concurrently*.

When BlueZ performs scanning, on discovering a device, an object representing it is created and retained by BlueZ and exported to the D-Bus system bus. Such objects are known as *managed* objects.

At the same time, a signal *InterfacesAdded* is emitted and this informs interested, connected D-Bus services of the newly discovered device.

```
signal time=1635330796.572439 sender=:1.15 -> destination=(null destination) serial=114
path=/; interface=org.freedesktop.DBus.ObjectManager; member=InterfacesAdded
   object path "/org/bluez/hci0/dev_00_F2_26_94_4F_D4"
   array [
      dict entry(
         string "org.freedesktop.DBus.Introspectable"
         array [
         ]
      )
      dict entry(
         string "org.bluez.Device1"
         array [
            dict entry(
               string "Address"
               variant                    string "00:F2:26:94:4F:D4"
            )
            dict entry(
               string "AddressType"
               variant                    string "random"
            )
            dict entry(
```

```
               string "Alias"
               variant                    string "00-F2-26-94-4F-D4"
            )
            dict entry(
               string "Paired"
               variant                    boolean false
            )
            dict entry(
               string "Trusted"
               variant                    boolean false
            )
            dict entry(
               string "Blocked"
               variant                    boolean false
            )
            dict entry(
               string "LegacyPairing"
               variant                    boolean false
            )
            dict entry(
               string "RSSI"
               variant                    int16 -81
            )
            dict entry(
               string "Connected"
               variant                    boolean false
            )
            dict entry(
               string "UUIDs"
               variant                    array [
                     string "0000fd6f-0000-1000-8000-00805f9b34fb"
                  ]
            )
            dict entry(
               string "Adapter"
               variant                    object path "/org/bluez/hci0"
            )
            dict entry(
               string "ServiceData"
               variant                    array [
                     dict entry(
                        string "0000fd6f-0000-1000-8000-00805f9b34fb"
                        variant                          array of bytes [
                              1a 11 5d 6a 7d 1e 39 f3 ff 04 3e 25 23 d6 b2 00
                              e6 e6 3c 74
                           ]
                     )
                  ]
            )
            dict entry(
               string "ServicesResolved"
               variant                    boolean false
            )
         ]
      )
      dict entry(
         string "org.freedesktop.DBus.Properties"
         array [
         ]
      )
   ]
```

Figure 1 - an example InterfacesAdded signal reporting a newly discovered device

But if this or another application requests scanning shortly afterwards, any devices already known to BlueZ will not be reported again by signalling. This can cause seemingly strange results with devices that are clearly in range and currently advertising not being signalled when BlueZ is scanning.

The answer to this strange seeming situation is this. The D-Bus org.bluez service exports a root (“/”) object which implements a number of standard interfaces, including one called org.freedesktop.DBus.ObjectManager or *ObjectManager* for short.

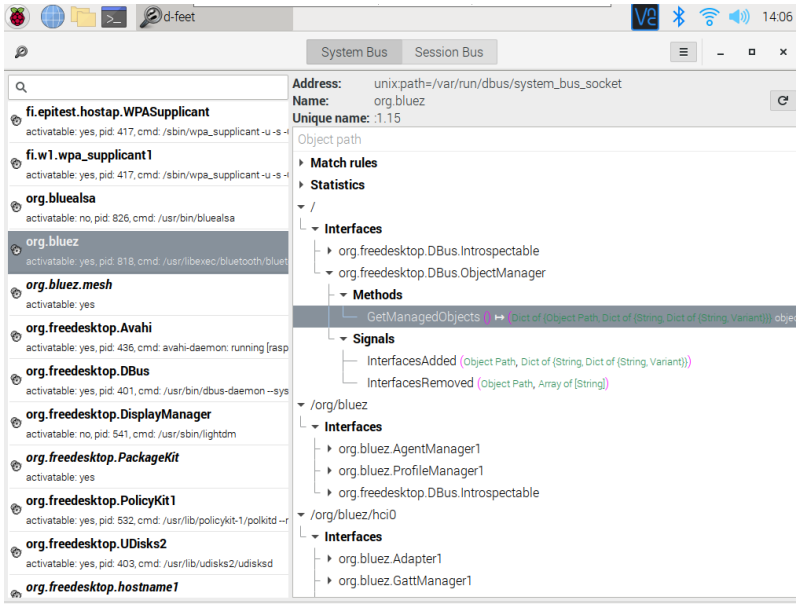

Figure 2 - The ObjectManager interface of the BlueZ root object

ObjectManager allows the list of objects currently known to BlueZ (*managed objects*) to be requested by calling the *GetManagedObjects* method. This returns a dictionary of objects, each with its D-Bus path as a key and another dictionary as its value.

There are a number of different types of *managed objects*, not just those which represent devices[1]. Objects which implement the org.bluez.Device1 interface (let's just call them *device objects*) represent Bluetooth devices. Objects which implement org.bluez.Adapter1 represent Bluetooth adapters that the system has. Therefore, to obtain a complete list of devices, an application must both obtain the current list of managed objects that implement the Device1 interface and start scanning, capturing details of additional devices as they are discovered and reported in InterfacesAdded signals. Of course this too can result in unexpected results. The complete list of devices could include devices that are no longer advertising (moved out of range since initial discovery, for example) but which have not yet been removed from the managed objects list. If such a device is selected by the user say and then an attempt made to connect to it, the attempt will fail because the device is no longer available. But that can happen on other platforms too, so it's not as unusual as it might seem at first.

A discovered device does not exist as a managed object forever. After a time, it is removed by BlueZ and this event is notified to interested applications by emitting a InterfacesRemoved signal. The specific time value for devices to be removed is defined in the BlueZ configuration file /etc/bluetooth/main.conf in a property called TemporaryTimeout which has a default value of 30 seconds.

```
signal time=1635330828.891484 sender=:1.15 -> destination=(null destination) serial=120
path=/; interface=org.freedesktop.DBus.ObjectManager; member=InterfacesRemoved
   object path "/org/bluez/hci0/dev_2C_48_35_90_5D_6E"
   array [
      string "org.freedesktop.DBus.Properties"
      string "org.freedesktop.DBus.Introspectable"
      string "org.bluez.Device1"
   ]
```

Figure 3 - example InterfacesRemoved signal

[1] The BlueZ API documentation lists the interfaces of all managed objects

Device objects include a series of properties, each of which has a name and value. Which particular properties the object has depends on which particular properties the physical device was found to have. In the context of device discovery this means the data items contained in advertising packets or other attribute values like the signal strength. The Bluetooth Core Specification defines advertising, including packet structure and header fields. The Core Specification Supplement defines the fields that advertising packets can contain in the payload part. Header fields are mandatory (e.g. a Bluetooth device address) whilst payload fields are all optional (e.g. a human friendly name).

Whenever one or more properties of a managed device object changes and this is detected by BlueZ, a signal called PropertiesChanged is emitted, and the modified property is reported with its name and new value. If multiple properties have changed, they are all reported in the same signal. For example, it's common to see RSSI property values varying and reported in PropertiesChanged signals.

```
signal time=1635335844.209568 sender=:1.15 -> destination=(null destination) serial=564
path=/org/bluez/hci0/dev_6C_48_28_1B_DB_00; interface=org.freedesktop.DBus.Properties;
member=PropertiesChanged
   string "org.bluez.Device1"
   array [
      dict entry(
         string "RSSI"
         variant             int16 -89
      )
      dict entry(
         string "TxPower"
         variant             int16 12
      )
   ]
   array [
   ]
```

Figure 4 - example PropertiesChanged signal

So in summary, the standard procedure for obtaining a list of devices to consider connecting to is therefore:

1. Obtain the list of known device objects by calling GetManagedObjects on the ObjectManager interface
2. Perform Bluetooth scanning to discover new devices that are advertising but not currently in the managed device objects list. Receive details of discovered devices in InterfacesAdded signals.
3. Concatenate the results of (1) and (2)
4. Keep the properties of the devices in your list up to date by handling PropertiesChanged signals.
5. Remove devices from your local list of discovered devices using information reported in InterfaceRemoved signals.

3.2 Implementing Device Discovery

Let's put the theory into practice.

Copy the files bluetooth_utils.py and bluetooth_constants.py from the study guide's code/solutions/python folder. Feel free to take a look at them.

As you have learned, device discovery is a surprisingly complicated process so we'll progress our coding in a number of bite-sized stages.

3.2.1 StartDiscovery and InterfacesAdded

The first goal will be to initiate device scanning and track details of devices discovered and reported in InterfacesAdded signals. The BlueZ method we need to use to start scanning (*StartDiscovery*) is owned by the Adapter1 interface and documented here: https://git.kernel.org/pub/scm/bluetooth/bluez.git/tree/doc/adapter-api.txt

You can also see it using D-Feet:

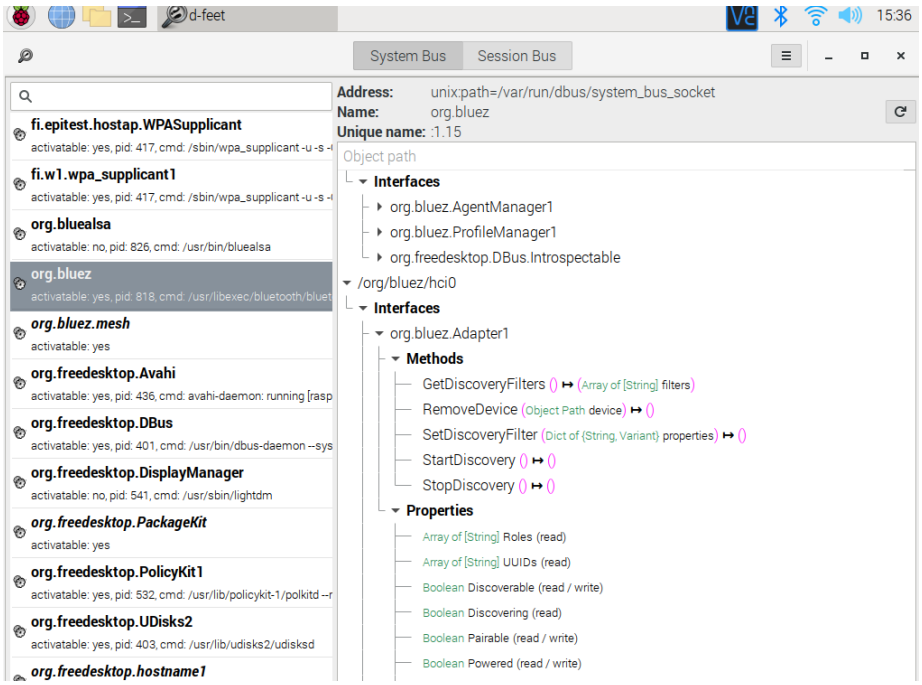

Figure 5 - The Adapter1 interface

Create a file called client_discover_devices.py and add the following code:

```
#!/usr/bin/python3
from gi.repository import GLib
import bluetooth_utils
import bluetooth_constants
import dbus
import dbus.mainloop.glib
import sys
sys.path.insert(0, '.')

adapter_interface = None
mainloop = None
timer_id = None

devices = {}

def interfaces_added(path, interfaces):
    # interfaces is an array of dictionary entries
    if not bluetooth_constants.DEVICE_INTERFACE in interfaces:
        return
```

```
    device_properties = interfaces[bluetooth_constants.DEVICE_INTERFACE]
    if path not in devices:
        print("NEW path  :", path)
        devices[path] = device_properties
        dev = devices[path]
        if 'Address' in dev:
            print("NEW bdaddr: ",
bluetooth_utils.dbus_to_python(dev['Address']))
        if 'Name' in dev:
            print("NEW name  : ",
bluetooth_utils.dbus_to_python(dev['Name']))
        if 'RSSI' in dev:
            print("NEW RSSI  : ",
bluetooth_utils.dbus_to_python(dev['RSSI']))
        print("------------------------------")

def discover_devices(bus):
    global adapter_interface
    global mainloop
    global timer_id
    adapter_path = bluetooth_constants.BLUEZ_NAMESPACE +
bluetooth_constants.ADAPTER_NAME

    # acquire an adapter proxy object and its Adapter1 interface so we can
call its methods
    adapter_object = bus.get_object(bluetooth_constants.BLUEZ_SERVICE_NAME,
adapter_path)
    adapter_interface=dbus.Interface(adapter_object,
bluetooth_constants.ADAPTER_INTERFACE)

    # register signal handler functions so we can asynchronously report
discovered devices

    # InterfacesAdded signal is emitted by BlueZ when an advertising packet
from a device it doesn't
    # already know about is received
    bus.add_signal_receiver(interfaces_added,
            dbus_interface = bluetooth_constants.DBUS_OM_IFACE,
            signal_name = "InterfacesAdded")

    mainloop = GLib.MainLoop()

    adapter_interface.StartDiscovery(byte_arrays=True)

    mainloop.run()

# dbus initialisation steps
dbus.mainloop.glib.DBusGMainLoop(set_as_default=True)
```

```
bus = dbus.SystemBus()

print("Scanning")
discover_devices(bus)
```

Run this code. You'll get results like these:

```
Scanning
NEW path  : /org/bluez/hci0/dev_1E_E2_2E_BC_12_F2
NEW bdaddr:  1E:E2:2E:BC:12:F2
NEW RSSI  :  -85
------------------------------
NEW path  : /org/bluez/hci0/dev_54_55_AA_4C_DA_06
NEW bdaddr:  54:55:AA:4C:DA:06
NEW RSSI  :  -77
------------------------------
NEW path  : /org/bluez/hci0/dev_75_DC_5F_26_E5_42
NEW bdaddr:  75:DC:5F:26:E5:42
NEW RSSI  :  -85
------------------------------
NEW path  : /org/bluez/hci0/dev_43_71_75_4A_B9_EA
NEW bdaddr:  43:71:75:4A:B9:EA
NEW RSSI  :  -95
------------------------------
NEW path  : /org/bluez/hci0/dev_65_1B_FD_FF_15_42
NEW bdaddr:  65:1B:FD:FF:15:42
NEW RSSI  :  -99
------------------------------
NEW path  : /org/bluez/hci0/dev_40_49_0F_3F_A8_2A
NEW bdaddr:  40:49:0F:3F:A8:2A
NEW name  :  Living Room TV
NEW RSSI  :  -81
------------------------------
NEW path  : /org/bluez/hci0/dev_08_D9_5A_BC_21_3F
NEW bdaddr:  08:D9:5A:BC:21:3F
NEW RSSI  :  -95
```

As it stands, this code will block in the mainloop and receive signals until you interrupt it. We'll add a timer to limit how long scanning is performed for next. But before we do, look at the code and consider the following points:

- We're using code from other Python files, namely the bluetooth_constants.py and bluetooth_utils.py files. We make them available to be imported by having the files in the same directory as our client_discovery_devices.py file and by executing the statement sys.path.insert(0, '.')
- We want to call the remote BlueZ method *StartDiscovery* which belongs to the Adapter1 interface so in the *discover_*devices function we obtain a proxy to the machine's main adapter object using *bus.get_object* and then acquire the Adapter1 interface from it. We've cheated slightly here and hard-coded the name of the Adapter object as /org/bluez/hci0. This is commonly the name used for the primary (and typically, only) adapter in a system. But Adapter objects are managed objects and so a list of one or more known to BlueZ on this system can also be obtained using GetManagedObjects. This is more elegant and safer. But we'll allow ourselves one little shortcut this time!
- We then use *add_signal_receiver* to indicate to D-Bus that we want to receive InterfacesAdded signals. Note that this signal belongs to the *ObjectManager*.
- Next, we call StartDiscovery on the adapter proxy and block in the main loop so that signals can be received.

- We implement a function which will handle InterfacesAdded signals when received. Remind yourself what this signal looks like by reviewing Figure 1. In the interface_added callback function we start by checking that the array of interface names received in the call includes the Device1 interface. If it doesn't then this general purpose signal does not relate to the discovery of a device and so we're not interested. Note that we use a function that is already implemented in the bluetooth_utils.py source file called dbus_to_python. Take a look at this function. As you'll see, it converts DBus data types into Python data types.
- Obviously, the discover_devices function is called from our main code block at the bottom.

3.2.3 Adding a time limit

Next we'll add a limit to the time to be spent scanning and allow this to be specified from the command line. Update your code as highlighted in red:

```
#!/usr/bin/python3
from gi.repository import GLib
import bluetooth_utils
import bluetooth_constants
import dbus
import dbus.mainloop.glib
import sys
sys.path.insert(0, '.')

adapter_interface = None
mainloop = None
timer_id = None

devices = {}

def interfaces_added(path, interfaces):
    # interfaces is an array of dictionary entries
    if not bluetooth_constants.DEVICE_INTERFACE in interfaces:
        return
    device_properties = interfaces[bluetooth_constants.DEVICE_INTERFACE]
    if path not in devices:
        print("NEW path  :", path)
        devices[path] = device_properties
        dev = devices[path]
        if 'Address' in dev:
            print("NEW bdaddr: ",
bluetooth_utils.dbus_to_python(dev['Address']))
        if 'Name' in dev:
            print("NEW name  : ",
bluetooth_utils.dbus_to_python(dev['Name']))
        if 'RSSI' in dev:
            print("NEW RSSI  : ",
bluetooth_utils.dbus_to_python(dev['RSSI']))
        print("------------------------------")
```

```
def discovery_timeout():
    global adapter_interface
    global mainloop
    global timer_id
    GLib.source_remove(timer_id)
    mainloop.quit()
    adapter_interface.StopDiscovery()
    bus = dbus.SystemBus()
    bus.remove_signal_receiver(interfaces_added,"InterfacesAdded")
    return True

def discover_devices(bus,timeout):
    global adapter_interface
    global mainloop
    global timer_id
    adapter_path = bluetooth_constants.BLUEZ_NAMESPACE +
bluetooth_constants.ADAPTER_NAME

    # acquire an adapter proxy object and its Adapter1 interface so we can
call its methods
    adapter_object = bus.get_object(bluetooth_constants.BLUEZ_SERVICE_NAME,
adapter_path)
    adapter_interface=dbus.Interface(adapter_object,
bluetooth_constants.ADAPTER_INTERFACE)

    # register signal handler functions so we can asynchronously report
discovered devices

    # InterfacesAdded signal is emitted by BlueZ when an advertising packet
from a device it doesn't
    # already know about is received
    bus.add_signal_receiver(interfaces_added,
            dbus_interface = bluetooth_constants.DBUS_OM_IFACE,
            signal_name = "InterfacesAdded")

    mainloop = GLib.MainLoop()
    timer_id = GLib.timeout_add(timeout, discovery_timeout)
    adapter_interface.StartDiscovery(byte_arrays=True)

    mainloop.run()

if (len(sys.argv) != 2):
    print("usage: python3 client_discover_devices.py [scantime (secs)]")
    sys.exit(1)

scantime = int(sys.argv[1]) * 1000

# dbus initialisation steps
```

```
dbus.mainloop.glib.DBusGMainLoop(set_as_default=True)
bus = dbus.SystemBus()

print("Scanning")
discover_devices(bus, scantime)
```

Run your updated code with an argument of 10, indicating that scanning should be performed for 10 seconds and then stopped.

Notes regarding the most recent code updates:

- The discover_devices function now has a timeout parameter that we pass a value in milliseconds to
- We create a timer using GLib.timeout_add which causes the function *discovery_timeout* to be called after the timeout period has elapsed.
- In *discovery_timeout* we remove the timer, stop the event loop, call the BlueZ Adapter method StopDiscovery to stop scanning and then unregister the InterfacesAdded signal receiver.

3.2.3 InterfacesRemoved

Next we'll handle InterfacesRemoved signals. Update your code as shown:

```
#!/usr/bin/python3
from gi.repository import GLib
import bluetooth_utils
import bluetooth_constants
import dbus
import dbus.mainloop.glib
import sys
sys.path.insert(0, '.')

adapter_interface = None
mainloop = None
timer_id = None

devices = {}

def interfaces_removed(path, interfaces):
    # interfaces is an array of dictionary strings in this signal
    if not bluetooth_constants.DEVICE_INTERFACE in interfaces:
        return
    if path in devices:
        dev = devices[path]
        if 'Address' in dev:
            print("DEL bdaddr: ",
bluetooth_utils.dbus_to_python(dev['Address']))
        else:
            print("DEL path  : ", path)
```

```
            print("------------------------------")
        del devices[path]

def interfaces_added(path, interfaces):
    # interfaces is an array of dictionary entries
    if not bluetooth_constants.DEVICE_INTERFACE in interfaces:
        return
    device_properties = interfaces[bluetooth_constants.DEVICE_INTERFACE]
    if path not in devices:
        print("NEW path  :", path)
        devices[path] = device_properties
        dev = devices[path]
        if 'Address' in dev:
            print("NEW bdaddr: ",
bluetooth_utils.dbus_to_python(dev['Address']))
        if 'Name' in dev:
            print("NEW name  : ",
bluetooth_utils.dbus_to_python(dev['Name']))
        if 'RSSI' in dev:
            print("NEW RSSI  : ",
bluetooth_utils.dbus_to_python(dev['RSSI']))
        print("------------------------------")

def discovery_timeout():
    global adapter_interface
    global mainloop
    global timer_id
    GLib.source_remove(timer_id)
    mainloop.quit()
    adapter_interface.StopDiscovery()
    bus = dbus.SystemBus()
    bus.remove_signal_receiver(interfaces_added,"InterfacesAdded")
    bus.remove_signal_receiver(interfaces_added,"InterfacesRemoved")
    return True

def discover_devices(bus,timeout):
    global adapter_interface
    global mainloop
    global timer_id
    adapter_path = bluetooth_constants.BLUEZ_NAMESPACE +
bluetooth_constants.ADAPTER_NAME

    # acquire an adapter proxy object and its Adapter1 interface so we can
call its methods
    adapter_object = bus.get_object(bluetooth_constants.BLUEZ_SERVICE_NAME,
adapter_path)
    adapter_interface=dbus.Interface(adapter_object,
bluetooth_constants.ADAPTER_INTERFACE)
```

```
    # register signal handler functions so we can asynchronously report
discovered devices

    # InterfacesAdded signal is emitted by BlueZ when an advertising packet
from a device it doesn't
    # already know about is received
    bus.add_signal_receiver(interfaces_added,
            dbus_interface = bluetooth_constants.DBUS_OM_IFACE,
            signal_name = "InterfacesAdded")

    # InterfacesRemoved signal is emitted by BlueZ when a device "goes away"
    bus.add_signal_receiver(interfaces_removed,
            dbus_interface = bluetooth_constants.DBUS_OM_IFACE,
            signal_name = "InterfacesRemoved")

    mainloop = GLib.MainLoop()
    timer_id = GLib.timeout_add(timeout, discovery_timeout)
    adapter_interface.StartDiscovery(byte_arrays=True)

    mainloop.run()

if (len(sys.argv) != 2):
    print("usage: python3 client_discover_devices.py [scantime (secs)]")
    sys.exit(1)

scantime = int(sys.argv[1]) * 1000

# dbus initialisation steps
dbus.mainloop.glib.DBusGMainLoop(set_as_default=True)
bus = dbus.SystemBus()

print("Scanning")
discover_devices(bus, scantime)
```

The changes:

- We register for the InterfacesRemoved signal and have provide the function *interfaces_removed* to be called when this signal is received.
- In *interfaces_removed* we report the fact and remove the corresponding entry from our list (dictionary in fact) of discovered devices.
- When our timer expires, we unregister the InterfacesRemoved signal handler.

Test the new code as follows. Specify that scanning should be performed for 60 seconds from the command line. Have a device to hand which is advertising. You should see it reported in the list of

NEW devices. Stop that device from advertising after about 10 seconds. You should see it reported as having been removed with a DEL message to the console.

In testing this exercise, a BBC micro:bit was used. It advertises when powered on. To stop it advertising, either removing its power source or connecting to it from a smartphone app could be used to stop it advertising. Here are the test results.

```
python3 client_discover_devices.py 60
Scanning
NEW path  : /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D
NEW bdaddr:  EB:EE:7B:08:EC:3D
NEW name  :  BBC micro:bit [vutoz]
NEW RSSI  :  -74
------------------------------
NEW path  : /org/bluez/hci0/dev_78_79_F6_CE_79_C5
NEW bdaddr:  78:79:F6:CE:79:C5
NEW RSSI  :  -78
------------------------------
NEW path  : /org/bluez/hci0/dev_2A_77_DF_62_62_94
NEW bdaddr:  2A:77:DF:62:62:94
NEW RSSI  :  -92
------------------------------
NEW path  : /org/bluez/hci0/dev_7A_76_46_58_4F_6C
NEW bdaddr:  7A:76:46:58:4F:6C
NEW RSSI  :  -94
------------------------------
NEW path  : /org/bluez/hci0/dev_5E_E4_E7_D9_36_5E
NEW bdaddr:  5E:E4:E7:D9:36:5E
NEW RSSI  :  -80
------------------------------
NEW path  : /org/bluez/hci0/dev_61_88_50_19_C1_E9
NEW bdaddr:  61:88:50:19:C1:E9
NEW RSSI  :  -85
------------------------------
NEW path  : /org/bluez/hci0/dev_2C_48_35_90_5D_6E
NEW bdaddr:  2C:48:35:90:5D:6E
NEW RSSI  :  -83
------------------------------
NEW path  : /org/bluez/hci0/dev_40_49_0F_3F_A8_2A
NEW bdaddr:  40:49:0F:3F:A8:2A
NEW name  :  Living Room TV
NEW RSSI  :  -96
------------------------------
DEL bdaddr:  EB:EE:7B:08:EC:3D
```

3.2.4 PropertiesChanged

Whilst scanning for devices it can be useful or just plain interesting to also keep an eye on properties of devices like the signal strength and to report changes in them. This is not strictly necessary for the purposes of device discovery but it's commonly done so that's the next task.

Update your code as shown in red:

```
#!/usr/bin/python3
from gi.repository import GLib
import bluetooth_utils
import bluetooth_constants
import dbus
import dbus.mainloop.glib
import sys
sys.path.insert(0, '.')

adapter_interface = None
```

```
mainloop = None
timer_id = None

devices = {}

def properties_changed(interface, changed, invalidated, path):
    if interface != bluetooth_constants.DEVICE_INTERFACE:
        return
    if path in devices:
        devices[path] = dict(devices[path].items())
        devices[path].update(changed.items())
    else:
        devices[path] = changed

    dev = devices[path]
    print("CHG path  :", path)
    if 'Address' in dev:
        print("CHG bdaddr: ",
bluetooth_utils.dbus_to_python(dev['Address']))
    if 'Name' in dev:
        print("CHG name  : ", bluetooth_utils.dbus_to_python(dev['Name']))
    if 'RSSI' in dev:
        print("CHG RSSI  : ", bluetooth_utils.dbus_to_python(dev['RSSI']))
    print("------------------------------")

def interfaces_removed(path, interfaces):
    # interfaces is an array of dictionary strings in this signal
    if not bluetooth_constants.DEVICE_INTERFACE in interfaces:
        return
    if path in devices:
        dev = devices[path]
        if 'Address' in dev:
            print("DEL bdaddr: ",
bluetooth_utils.dbus_to_python(dev['Address']))
        else:
            print("DEL path  : ", path)
            print("------------------------------")
        del devices[path]

def interfaces_added(path, interfaces):
    # interfaces is an array of dictionary entries
    if not bluetooth_constants.DEVICE_INTERFACE in interfaces:
        return
    device_properties = interfaces[bluetooth_constants.DEVICE_INTERFACE]
    if path not in devices:
        print("NEW path  :", path)
        devices[path] = device_properties
        dev = devices[path]
```

```
        if 'Address' in dev:
            print("NEW bdaddr: ",
bluetooth_utils.dbus_to_python(dev['Address']))
        if 'Name' in dev:
            print("NEW name  : ",
bluetooth_utils.dbus_to_python(dev['Name']))
        if 'RSSI' in dev:
            print("NEW RSSI  : ",
bluetooth_utils.dbus_to_python(dev['RSSI']))
        print("------------------------------")

def discovery_timeout():
    global adapter_interface
    global mainloop
    global timer_id
    GLib.source_remove(timer_id)
    mainloop.quit()
    adapter_interface.StopDiscovery()
    bus = dbus.SystemBus()
    bus.remove_signal_receiver(interfaces_added,"InterfacesAdded")
    bus.remove_signal_receiver(interfaces_added,"InterfacesRemoved")
    bus.remove_signal_receiver(properties_changed,"PropertiesChanged")
    return True

def discover_devices(bus,timeout):
    global adapter_interface
    global mainloop
    global timer_id
    adapter_path = bluetooth_constants.BLUEZ_NAMESPACE +
bluetooth_constants.ADAPTER_NAME

    # acquire an adapter proxy object and its Adapter1 interface so we can
call its methods
    adapter_object = bus.get_object(bluetooth_constants.BLUEZ_SERVICE_NAME,
adapter_path)
    adapter_interface=dbus.Interface(adapter_object,
bluetooth_constants.ADAPTER_INTERFACE)

    # register signal handler functions so we can asynchronously report
discovered devices

    # InterfacesAdded signal is emitted by BlueZ when an advertising packet
from a device it doesn't
    # already know about is received
    bus.add_signal_receiver(interfaces_added,
            dbus_interface = bluetooth_constants.DBUS_OM_IFACE,
            signal_name = "InterfacesAdded")
```

```
    # InterfacesRemoved signal is emitted by BlueZ when a device "goes away"
    bus.add_signal_receiver(interfaces_removed,
            dbus_interface = bluetooth_constants.DBUS_OM_IFACE,
            signal_name = "InterfacesRemoved")

    # PropertiesChanged signal is emitted by BlueZ when something re: a
device already encountered
    # changes e.g. the RSSI value
    bus.add_signal_receiver(properties_changed,
           dbus_interface = bluetooth_constants.DBUS_PROPERTIES,
            signal_name = "PropertiesChanged",
            path_keyword = "path")

    mainloop = GLib.MainLoop()
    timer_id = GLib.timeout_add(timeout, discovery_timeout)
    adapter_interface.StartDiscovery(byte_arrays=True)

    mainloop.run()

if (len(sys.argv) != 2):
    print("usage: python3 client_discover_devices.py [scantime (secs)]")
    sys.exit(1)

scantime = int(sys.argv[1]) * 1000

# dbus initialisation steps
dbus.mainloop.glib.DBusGMainLoop(set_as_default=True)
bus = dbus.SystemBus()

print("Scanning")
discover_devices(bus, scantime)
```

About the changes:

- We have a new signal handler, *properties_changed*. In this function we either update the properties of the device object stored in our *devices* list if we've already discovered this device or we add it to the list. We report the signal with the label CHG.
- We register for the PropertiesChanged signal in *discover_devices* and unregister in the timeout handler.

Test your code and move a device around so that its RSSI value changes. Results will look something like this:

```
Scanning
NEW path  : /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D
NEW bdaddr:  EB:EE:7B:08:EC:3D
NEW name  :  BBC micro:bit [vutoz]
NEW RSSI  :  -71
------------------------------
```

```
NEW path  : /org/bluez/hci0/dev_54_37_93_3A_09_BE
NEW bdaddr:  54:37:93:3A:09:BE
NEW RSSI  :  -88
------------------------------
NEW path  : /org/bluez/hci0/dev_42_4A_8B_09_10_A7
NEW bdaddr:  42:4A:8B:09:10:A7
NEW RSSI  :  -83
------------------------------
CHG path  : /org/bluez/hci0/dev_40_49_0F_3F_A8_2A
CHG bdaddr:  40:49:0F:3F:A8:2A
CHG name  :  Living Room TV
CHG RSSI  :  -80
------------------------------
CHG path  : /org/bluez/hci0/dev_69_FF_62_6E_83_CF
CHG bdaddr:  69:FF:62:6E:83:CF
CHG RSSI  :  -95
------------------------------
CHG path  : /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D
CHG bdaddr:  EB:EE:7B:08:EC:3D
CHG name  :  BBC micro:bit [vutoz]
CHG RSSI  :  -88
------------------------------
CHG path  : /org/bluez/hci0/dev_69_FF_62_6E_83_CF
CHG bdaddr:  69:FF:62:6E:83:CF
CHG RSSI  :  -95
------------------------------
CHG path  : /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D
CHG bdaddr:  EB:EE:7B:08:EC:3D
CHG name  :  BBC micro:bit [vutoz]
CHG RSSI  :  -74
------------------------------
NEW path  : /org/bluez/hci0/dev_4A_A0_65_CD_3E_7F
NEW bdaddr:  4A:A0:65:CD:3E:7F
NEW RSSI  :  -96
------------------------------
CHG path  : /org/bluez/hci0/dev_40_49_0F_3F_A8_2A
CHG bdaddr:  40:49:0F:3F:A8:2A
CHG name  :  Living Room TV
CHG RSSI  :  -88
------------------------------
NEW path  : /org/bluez/hci0/dev_21_07_CF_8E_CC_70
NEW bdaddr:  21:07:CF:8E:CC:70
NEW RSSI  :  -98
------------------------------
CHG path  : /org/bluez/hci0/dev_4A_A0_65_CD_3E_7F
CHG bdaddr:  4A:A0:65:CD:3E:7F
CHG RSSI  :  -96
------------------------------
CHG path  : /org/bluez/hci0/dev_5D_E7_5E_43_79_FB
CHG bdaddr:  5D:E7:5E:43:79:FB
CHG RSSI  :  -96
------------------------------
CHG path  : /org/bluez/hci0/dev_40_49_0F_3F_A8_2A
CHG bdaddr:  40:49:0F:3F:A8:2A
CHG name  :  Living Room TV
CHG RSSI  :  -74
------------------------------
```

You can clearly see RSSI values varying and being reported in PropertiesChanged signals here.

3.2.5 GetManagedObjects

Our final task is to find the details of all those devices which are already known to BlueZ and therefore not being reported using InterfacesAdded signals. For good measure we'll also report a summary of all devices found by the various methods at the end of the process.

Update your code as shown:

```
#!/usr/bin/python3
from gi.repository import GLib
```

```
import bluetooth_utils
import bluetooth_constants
import dbus
import dbus.mainloop.glib
import sys
sys.path.insert(0, '.')

adapter_interface = None
mainloop = None
timer_id = None

devices = {}
managed_objects_found = 0

def get_known_devices(bus):
    global managed_objects_found
    object_manager =
dbus.Interface(bus.get_object(bluetooth_constants.BLUEZ_SERVICE_NAME, "/"),
bluetooth_constants.DBUS_OM_IFACE)
    managed_objects=object_manager.GetManagedObjects()

    for path, ifaces in managed_objects.items():
        for iface_name in ifaces:
            if iface_name == bluetooth_constants.DEVICE_INTERFACE:
                managed_objects_found += 1
                print("EXI path  : ", path)
                device_properties =
ifaces[bluetooth_constants.DEVICE_INTERFACE]
                devices[path] = device_properties
                if 'Address' in device_properties:
                    print("EXI bdaddr: ",
bluetooth_utils.dbus_to_python(device_properties['Address']))
                    print("-----------------------------")

def properties_changed(interface, changed, invalidated, path):
    if interface != bluetooth_constants.DEVICE_INTERFACE:
        return
    if path in devices:
        devices[path] = dict(devices[path].items())
        devices[path].update(changed.items())
    else:
        devices[path] = changed

    dev = devices[path]
    print("CHG path  :", path)
    if 'Address' in dev:
        print("CHG bdaddr: ",
bluetooth_utils.dbus_to_python(dev['Address']))
```

```
    if 'Name' in dev:
        print("CHG name  : ", bluetooth_utils.dbus_to_python(dev['Name']))
    if 'RSSI' in dev:
        print("CHG RSSI  : ", bluetooth_utils.dbus_to_python(dev['RSSI']))
    print("------------------------------")

def interfaces_removed(path, interfaces):
    # interfaces is an array of dictionary strings in this signal
    if not bluetooth_constants.DEVICE_INTERFACE in interfaces:
        return
    if path in devices:
        dev = devices[path]
        if 'Address' in dev:
            print("DEL bdaddr: ",
bluetooth_utils.dbus_to_python(dev['Address']))
        else:
            print("DEL path  : ", path)
            print("------------------------------")
        del devices[path]

def interfaces_added(path, interfaces):
    # interfaces is an array of dictionary entries
    if not bluetooth_constants.DEVICE_INTERFACE in interfaces:
        return
    device_properties = interfaces[bluetooth_constants.DEVICE_INTERFACE]
    if path not in devices:
        print("NEW path  :", path)
        devices[path] = device_properties
        dev = devices[path]
        if 'Address' in dev:
            print("NEW bdaddr: ",
bluetooth_utils.dbus_to_python(dev['Address']))
        if 'Name' in dev:
            print("NEW name  : ",
bluetooth_utils.dbus_to_python(dev['Name']))
        if 'RSSI' in dev:
            print("NEW RSSI  : ",
bluetooth_utils.dbus_to_python(dev['RSSI']))
        print("------------------------------")

def list_devices_found():
    print("Full list of devices",len(devices),"discovered:")
    print("------------------------------")
    for path in devices:
        dev = devices[path]
        print(bluetooth_utils.dbus_to_python(dev['Address']))

def discovery_timeout():
```

```
    global adapter_interface
    global mainloop
    global timer_id
    GLib.source_remove(timer_id)
    mainloop.quit()
    adapter_interface.StopDiscovery()
    bus = dbus.SystemBus()
    bus.remove_signal_receiver(interfaces_added,"InterfacesAdded")
    bus.remove_signal_receiver(interfaces_added,"InterfacesRemoved")
    bus.remove_signal_receiver(properties_changed,"PropertiesChanged")
    list_devices_found()
    return True

def discover_devices(bus,timeout):
    global adapter_interface
    global mainloop
    global timer_id
    adapter_path = bluetooth_constants.BLUEZ_NAMESPACE +
bluetooth_constants.ADAPTER_NAME

    # acquire an adapter proxy object and its Adapter1 interface so we can
call its methods
    adapter_object = bus.get_object(bluetooth_constants.BLUEZ_SERVICE_NAME,
adapter_path)
    adapter_interface=dbus.Interface(adapter_object,
bluetooth_constants.ADAPTER_INTERFACE)

    # register signal handler functions so we can asynchronously report
discovered devices

    # InterfacesAdded signal is emitted by BlueZ when an advertising packet
from a device it doesn't
    # already know about is received
    bus.add_signal_receiver(interfaces_added,
            dbus_interface = bluetooth_constants.DBUS_OM_IFACE,
            signal_name = "InterfacesAdded")

    # InterfacesRemoved signal is emitted by BlueZ when a device "goes away"
    bus.add_signal_receiver(interfaces_removed,
            dbus_interface = bluetooth_constants.DBUS_OM_IFACE,
            signal_name = "InterfacesRemoved")

    # PropertiesChanged signal is emitted by BlueZ when something re: a
device already encountered
    # changes e.g. the RSSI value
    bus.add_signal_receiver(properties_changed,
           dbus_interface = bluetooth_constants.DBUS_PROPERTIES,
            signal_name = "PropertiesChanged",
```

```
            path_keyword = "path")

    mainloop = GLib.MainLoop()
    timer_id = GLib.timeout_add(timeout, discovery_timeout)
    adapter_interface.StartDiscovery(byte_arrays=True)

    mainloop.run()

if (len(sys.argv) != 2):
    print("usage: python3 client_discover_devices.py [scantime (secs)]")
    sys.exit(1)

scantime = int(sys.argv[1]) * 1000

# dbus initialisation steps
dbus.mainloop.glib.DBusGMainLoop(set_as_default=True)
bus = dbus.SystemBus()

# ask for a list of devices already known to the BlueZ daemon
print("Listing devices already known to BlueZ:")
get_known_devices(bus)
print("Found ",managed_objects_found," managed device objects")
print("Scanning")
discover_devices(bus, scantime)
```

Note on the changes made:

- We've added two new functions; *get_known_devices* and *list_devices_found*
- *get_known_devices* calls the ObjectManager method GetManagedObjects and iterates through the results, selecting those objects that implement the Device1 interface. The instance of ObjectManager used is of course one which the BlueZ D-Bus service owns and which is attached to the exported root object. Devices found are reported on with the label EXI (for *existing*) and added to the *devices* dictionary which is where we collect details of all devices discovered using the various methods. The new function is called before we start scanning.
- When scanning completes and its timer expires, in the *discovery_timeout* function we call the other new function, *list_devices_found* where we iterate through the full list and print the Bluetooth device address of each member.

That's it for device discovery! You now have a pretty thorough knowledge of device discovery in BlueZ using D-Bus.

4. Connecting and Disconnecting

In a typical application flow, after device discovery the next step is usually to connect to a selected device. The user normally makes that selection. In this section we'll implement as simple a script as possible which will attempt to connect to a device, specified by its Bluetooth device address as a command line argument. We'll modify it later on to disconnect after a period of time.

4.1 Connecting

Device objects have a Connect method which can be called. But for the device object to exist in the first place, it must have first been discovered through scanning and exist in BlueZ as a managed object.

You may notice that the Adapter1 interface has a method called ConnectDevice. If you consult the API documentation it says "This method connects to device without need of performing General Discovery". However if you look more closely you will see that ConnectDevice is tagged as [experimental]. This usually means that the code concerned is relatively untested or incomplete. In this case however, in carrying out research for this study guide, the author was informed that this method should not be used and only exists to support Bluetooth SIG qualification testing at some point in the past. It should not be used in other contexts.

So, your new Python script will assume that scanning has been carried out separately and within 30 seconds so that devices discovered have not yet been removed from the managed objects list. You can of course use the script you developed in section 3 for this.

Connecting to a device is easy and takes a lot less code than device discovery. Here's the complete code which you should insert in a file ready for testing:

```
#!/usr/bin/python3
#
# Connects to a specified device
# Run from the command line with a bluetooth device address argument

import bluetooth_constants
import bluetooth_utils
import dbus
import sys
sys.path.insert(0, '.')

bus = None
device_interface = None

def connect(device_path):
    global bus
    global device_interface
    try:
        device_interface.Connect()
    except Exception as e:
        print("Failed to connect")
        print(e.get_dbus_name())
        print(e.get_dbus_message())
```

```
        if ("UnknownObject" in e.get_dbus_name()):
            print("Try scanning first to resolve this problem")
        return bluetooth_constants.RESULT_EXCEPTION
    else:
        print("Connected OK")
        return bluetooth_constants.RESULT_OK

if (len(sys.argv) != 2):
    print("usage: python3 client_connect_disconnect.py [bdaddr]")
    sys.exit(1)

bdaddr = sys.argv[1]
bus = dbus.SystemBus()
adapter_path = bluetooth_constants.BLUEZ_NAMESPACE +
bluetooth_constants.ADAPTER_NAME
device_path = bluetooth_utils.device_address_to_path(bdaddr, adapter_path)
device_proxy =
bus.get_object(bluetooth_constants.BLUEZ_SERVICE_NAME,device_path)
device_interface = dbus.Interface(device_proxy,
bluetooth_constants.DEVICE_INTERFACE)

print("Connecting to " + bdaddr)
connect(device_path)
```

Notes on this code:

- As a convenience we allow a Bluetooth device address in the usual format to be supplied as an argument. But D-Bus identifies objects by path and so we had to convert the address into a path that conforms to the BlueZ conventions first. This was done in a function in the bluetooth_utils.py source file.
- We acquire a proxy object corresponding to the device object with this path and a reference to the Device1 interface from it.
- In our connect function we call the Connect method on the Device1 interface. This is done in a try/except/else block so that we can handle any exceptions which might be thrown.
- Any exception is a failure but we know that the special case of the UnknownObject exception occurs when there is no Device object with the specified path identifier known to BlueZ and that this may be the case because scanning has not recently been performed (or the device genuinely is not present or is not advertising) so we provide the user with some useful feedback on this possibility.
- We use a selection of return codes to indicate the outcome of the operation (although we don't test for them in this example)

With a device you know you can connect to (it's advertising and either doesn't require pairing or does and your Linux machine has been paired with it) present and advertising, run your device discovery script and take note of the Bluetooth device address of this device. Then run your new

client_connect.py script, passing this device address as an argument. Here's the type of output you should see from the two scripts.

```
pi@raspberrypi:~/projects/ldsg/solutions/python/bluetooth $ python3
client_discover_devices.py 10
Listing devices already known to BlueZ:
Found  0  managed device objects
Scanning
NEW path  : /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D
NEW bdaddr:  EB:EE:7B:08:EC:3D
NEW name  :  BBC micro:bit [vutoz]
NEW RSSI  :  -64
------------------------------
NEW path  : /org/bluez/hci0/dev_45_85_98_8F_E1_65
NEW bdaddr:  45:85:98:8F:E1:65
NEW RSSI  :  -87
------------------------------
NEW path  : /org/bluez/hci0/dev_74_96_38_37_1E_8F
NEW bdaddr:  74:96:38:37:1E:8F
NEW RSSI  :  -92
------------------------------
NEW path  : /org/bluez/hci0/dev_62_AF_E3_A4_59_98
NEW bdaddr:  62:AF:E3:A4:59:98
NEW RSSI  :  -81
------------------------------
NEW path  : /org/bluez/hci0/dev_23_92_60_3D_82_B8
NEW bdaddr:  23:92:60:3D:82:B8
NEW RSSI  :  -81
------------------------------
NEW path  : /org/bluez/hci0/dev_6F_EC_3B_59_A7_E2
NEW bdaddr:  6F:EC:3B:59:A7:E2
NEW RSSI  :  -95
------------------------------
NEW path  : /org/bluez/hci0/dev_2C_48_35_90_5D_6E
NEW bdaddr:  2C:48:35:90:5D:6E
NEW RSSI  :  -94
------------------------------
NEW path  : /org/bluez/hci0/dev_40_49_0F_3F_A8_2A
NEW bdaddr:  40:49:0F:3F:A8:2A
NEW name  :  Living Room TV
NEW RSSI  :  -76
------------------------------
CHG path  : /org/bluez/hci0/dev_40_49_0F_3F_A8_2A
CHG bdaddr:  40:49:0F:3F:A8:2A
CHG name  :  Living Room TV
CHG RSSI  :  -76
------------------------------
Full list of devices 8 discovered:
------------------------------
EB:EE:7B:08:EC:3D
45:85:98:8F:E1:65
74:96:38:37:1E:8F
62:AF:E3:A4:59:98
23:92:60:3D:82:B8
6F:EC:3B:59:A7:E2
2C:48:35:90:5D:6E
40:49:0F:3F:A8:2A
pi@raspberrypi:~/projects/ldsg/solutions/python/bluetooth $
pi@raspberrypi:~/projects/ldsg/solutions/python/bluetooth $ python3
client_connect_disconnect.py EB:EE:7B:08:EC:3D
Connecting to EB:EE:7B:08:EC:3D
Connected OK
```

If you're curious, run `sudo dbus-montor --system` when executing client_connect.py in another window. You'll see messages such as these:

```
method call time=1635430645.890082 sender=:1.151 -> destination=:1.14 serial=4
path=/org/bluez/hci0/dev_EB_EE_7B_08_EC_3D; interface=org.bluez.Device1; member=Connect

signal time=1635430646.333806 sender=:1.14 -> destination=(null destination) serial=942
path=/org/bluez/hci0/dev_EB_EE_7B_08_EC_3D; interface=org.freedesktop.DBus.Properties;
member=PropertiesChanged
   string "org.bluez.Device1"
```

```
   array [
      dict entry(
         string "Connected"
         variant             boolean true
      )
   ]
   array [
   ]
```

Here you can see the call to the Connect method being made against a Device1 interface within the object with path /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D. The connection succeeds and so the Connected property of that object is modified and a PropertiesChanged signal emitted to communicate this. We could have listened for this signal of course.

4.2 Disconnecting

Disconnecting is very similar to connecting using the BlueZ APIs. The Device1 interface has a Disconnect method sitting right alongside the Connect method we already used. The same rules apply in that the device must be known to BlueZ and exist as a managed object before we can invoke its Disconnect method.

Update your code as shown in red:

```
#!/usr/bin/python3
#
# Connects to a specified device
# Run from the command line with a bluetooth device address argument

import bluetooth_constants
import bluetooth_utils
import dbus
import sys
import time
sys.path.insert(0, '.')

bus = None
device_interface = None

def connect():
    global bus
    global device_interface
    try:
        device_interface.Connect()
    except Exception as e:
        print("Failed to connect")
        print(e.get_dbus_name())
        print(e.get_dbus_message())
        if ("UnknownObject" in e.get_dbus_name()):
            print("Try scanning first to resolve this problem")
        return bluetooth_constants.RESULT_EXCEPTION
    else:
        print("Connected OK")
```

```
        return bluetooth_constants.RESULT_OK

def disconnect():
    global bus
    global device_interface
    try:
        device_interface.Disconnect()
    except Exception as e:
        print("Failed to disconnect")
        print(e.get_dbus_name())
        print(e.get_dbus_message())
        return bluetooth_constants.RESULT_EXCEPTION
    else:
        print("Disconnected OK")
        return bluetooth_constants.RESULT_OK

if (len(sys.argv) != 2):
    print("usage: python3 client_connect_disconnect.py [bdaddr]")
    sys.exit(1)

bdaddr = sys.argv[1]
bus = dbus.SystemBus()
adapter_path = bluetooth_constants.BLUEZ_NAMESPACE +
bluetooth_constants.ADAPTER_NAME
device_path = bluetooth_utils.device_address_to_path(bdaddr, adapter_path)
device_proxy =
bus.get_object(bluetooth_constants.BLUEZ_SERVICE_NAME,device_path)
device_interface = dbus.Interface(device_proxy,
bluetooth_constants.DEVICE_INTERFACE)

print("Connecting to " + bdaddr)
connect()
time.sleep(5)
print("Disconnecting from " + bdaddr)
disconnect()
```

Notes on these changes:

- There's nothing new to learn here in reality
- We added a new function called *disconnect*
- It uses the same bus and Device1 interface object we used to connect and proceeds in the same way as our *connect* function except that it calls Disconnect rather than Connect on the remote device object.

Test your code now.

4.3 Extra Credit

For an extra gold star, modify your code to read some of the properties of the target device and in particular, check whether it is already in a Connected state before attempting to connect to it. You'll need to use the Get method of the org.freedesktop.Dbus.Properties interface with the name of a property that a device object has. D-Feet will be useful in exploring this. Obviously BlueZ will only know that the target device has already been connected to if it is connected to the same Linux machine, so connect in one window and then try to connect again from another to test your change.

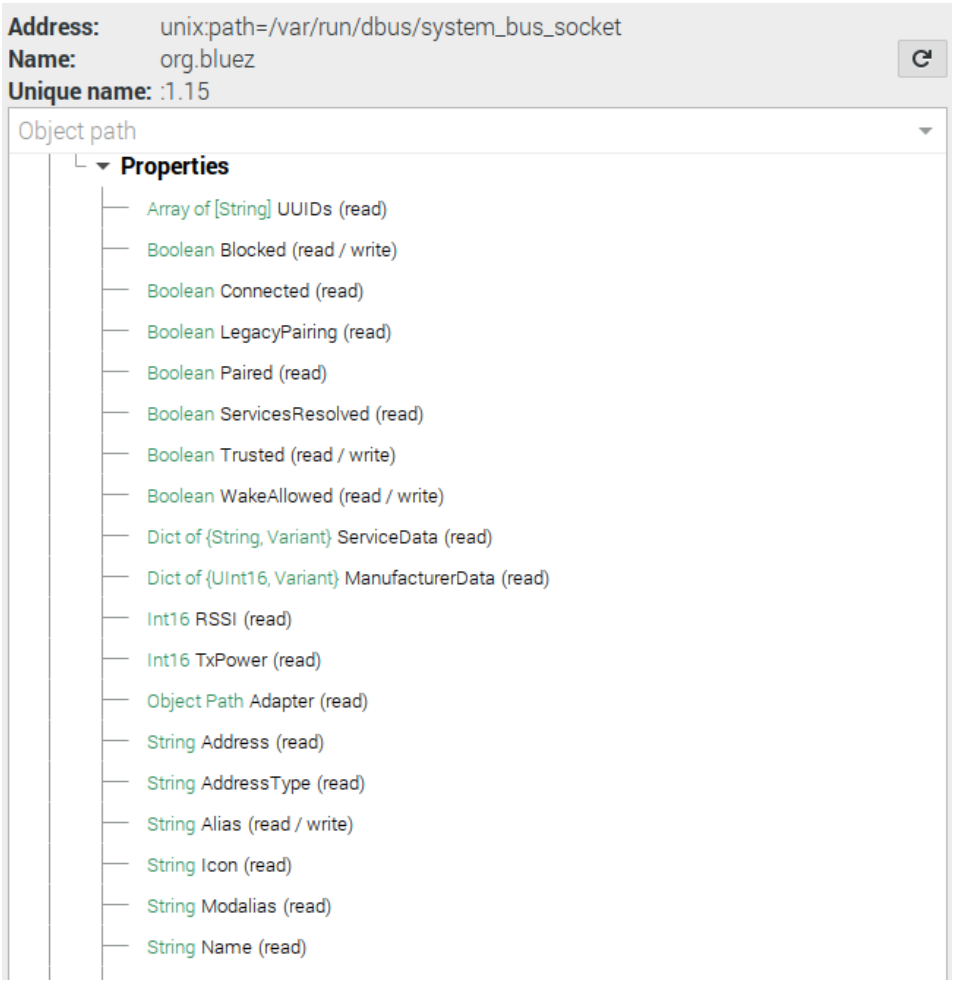

Figure 6 - some of the properties of a Device object

5. Service Discovery

5.1 BlueZ and Service Discovery

After connecting to a device which is known or expected to be a GATT server, it is typical to want to perform *service discovery* next.

Let's define what we mean by *service discovery* in this context.

A GATT server contains a series of GATT services, each of which contains one or more GATT characteristics. Each GATT characteristic has zero or more GATT descriptors attached to it. Figure 02 in module 02 depicts this. Services, characteristics and descriptors all have an associated UUID which indicates their type plus various other attributes which we may or may not be interested in for the purposes of our application.

When we perform service discovery, we are intending to discover this entire hierarchy of services, characteristics and descriptors, not just the services although that would also be legitimate. It depends on your requirements. From an application perspective we tend to be interested in accomplishing the following:

- Verifying that the connected device has the GATT services we expect and need. This is a good way to verify that the device we connected to is definitely of the right type for our application. A heart rate monitor application will only work if connected to a device which has the GATT heart rate service, for example. And don't forget, sometimes a profile will indicate that some services are optional so a device might be the right type of device but that doesn't necessarily mean it supports all possible functionality given it might not be mandatory that all services associated with the device type are implemented in a particular case.
- Verifying that each service has the characteristics and descriptors needed. Characteristics and descriptors can also sometimes be optional, so this is a good check.
- We will have expectations and requirements regarding the properties of characteristics which indicate which GATT operations are supported. A specification should tell us this and in the main, we can trust this information. We might choose to validate the device further though by checking the properties of each characteristic and descriptor to ensure they match expectations. And if you're developing a generalised application which can handle *any* device then you will definitely need to determine and make use of the properties of each characteristic (etc) when building your user interface dynamically.
- Ultimately (and details vary depending on the programming language, API and so on) we probably want to acquire some kind of reference to or identifier for each of the services, characteristics and descriptors that our application needs to work with.

BlueZ performs service discovery automatically on connecting to a device if it has not been paired and by default, caches the discovered details from paired devices so that service discovery doesn't need to be done every time a connection is established.

The discovery of services, characteristics and descriptors generates InterfacesAdded signals, so this can be used to allow an application to track the discovery process. And for each service, characteristic and descriptor discovered, an object is created and exported so that it is available on the system bus.

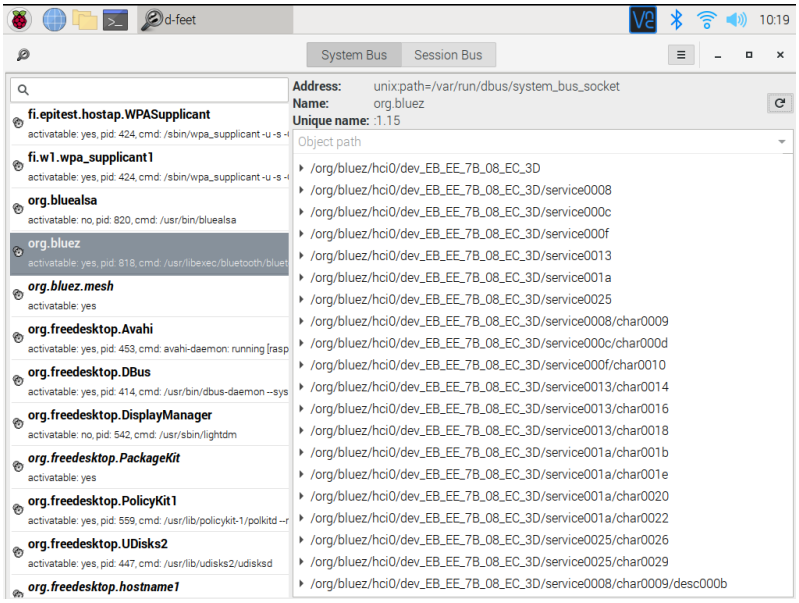

Figure 7 - a device, its services, characteristics and one descriptor as D-Bus objects

Service objects are identifiable through them implementing the GattService1 interface. This owns a number of properties including the UUID which tells us which type of service the object represents. UUIDs issued by the Bluetooth SIG are listed at https://www.bluetooth.com/specifications/assigned-numbers/

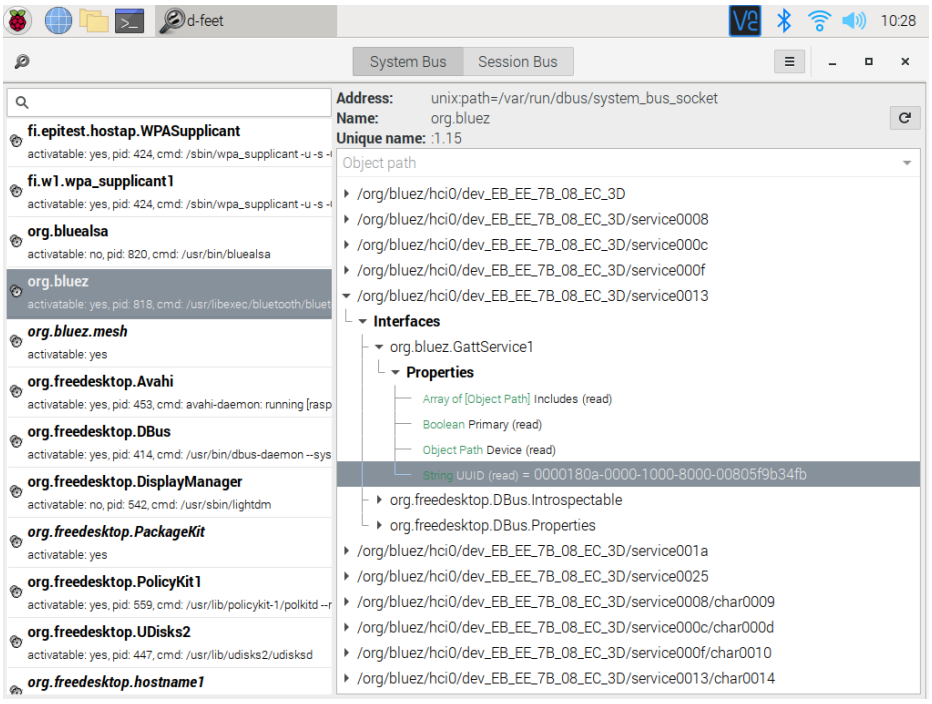

Figure 8 - A service object whose UUID property identifies it as representing the Device Information Service

As you can see in Figures 7 and 8, as always each of these D-Bus objects has a path which we can use to reference it from our code so we can execute methods of specific characteristics etc. You can also see how paths reflect the service/characteristic/descriptor hierarchy.

In this section, we'll write a Python script which will:

1. Report on the service discovery process by tracking InterfacesAdded signals

2. Verify that the connected device has a particular service

It's informative to examine the D-Bus signals that are emitted when service discovery is taking place. Here's an extract which shows a service being discovered and then one of its characteristics and descriptors:

```
signal time=1633330888.218003 sender=:1.15 -> destination=(null destination) serial=300
path=/; interface=org.freedesktop.DBus.ObjectManager; member=InterfacesAdded
   object path "/org/bluez/hci0/dev_C1_6D_4A_98_A0_36/service0008"
   array [
      dict entry(
         string "org.freedesktop.DBus.Introspectable"
         array [
         ]
      )
      dict entry(
         string "org.bluez.GattService1"
         array [
            dict entry(
               string "UUID"
               variant                    string "00001801-0000-1000-8000-00805f9b34fb"
            )
            dict entry(
               string "Device"
               variant                    object path
"/org/bluez/hci0/dev_C1_6D_4A_98_A0_36"
            )
            dict entry(
               string "Primary"
               variant                    boolean true
            )
            dict entry(
               string "Includes"
               variant                    array [
                  ]
            )
         ]
      )
      dict entry(
         string "org.freedesktop.DBus.Properties"
         array [
         ]
      )
   ]
signal time=1633330888.219080 sender=:1.15 -> destination=(null destination) serial=301
path=/; interface=org.freedesktop.DBus.ObjectManager; member=InterfacesAdded
   object path "/org/bluez/hci0/dev_C1_6D_4A_98_A0_36/service0008/char0009"
   array [
      dict entry(
         string "org.freedesktop.DBus.Introspectable"
         array [
         ]
      )
      dict entry(
         string "org.bluez.GattCharacteristic1"
         array [
            dict entry(
               string "UUID"
               variant                    string "00002a05-0000-1000-8000-00805f9b34fb"
            )
            dict entry(
               string "Service"
               variant                    object path
"/org/bluez/hci0/dev_C1_6D_4A_98_A0_36/service0008"
            )
            dict entry(
               string "Value"
               variant                    array [
                  ]
            )
            dict entry(
               string "Notifying"
               variant                    boolean false
            )
```

```
            dict entry(
               string "Flags"
               variant                    array [
                     string "indicate"
                  ]
            )
         ]
      )
      dict entry(
         string "org.freedesktop.DBus.Properties"
         array [
         ]
      )
   ]
signal time=1633330888.220140 sender=:1.15 -> destination=(null destination) serial=302
path=/; interface=org.freedesktop.DBus.ObjectManager; member=InterfacesAdded
   object path "/org/bluez/hci0/dev_C1_6D_4A_98_A0_36/service0008/char0009/desc000b"
   array [
      dict entry(
         string "org.freedesktop.DBus.Introspectable"
         array [
         ]
      )
      dict entry(
         string "org.bluez.GattDescriptor1"
         array [
            dict entry(
               string "UUID"
               variant                    string "00002902-0000-1000-8000-00805f9b34fb"
            )
            dict entry(
               string "Characteristic"
               variant                    object path
"/org/bluez/hci0/dev_C1_6D_4A_98_A0_36/service0008/char0009"
            )
            dict entry(
               string "Value"
               variant                    array [
                  ]
            )
         ]
      )
      dict entry(
         string "org.freedesktop.DBus.Properties"
         array [
         ]
      )
   ]
```

As you can see, service discovery yields a wealth of information.

5.2 Getting Started

Copy the code you wrote which connects to a specified device into a new file. You do not need the code concerned with disconnecting.

The first coding task will be to track the service discovery process and report on it as it progresses. Your code should also make a note of any significant services and characteristics found and save their paths to variables. Which services is your choice and depends on your test device. For the purposes of this exercise, we'll look for the Device Information Service (see https://www.bluetooth.com/specifications/specs/) which has a 16-bit UUID of 0x180A and the Model Number String characteristic within that service which has a UUID of 0x2A24.

First check that your starter code allows you to connect to a device specified by Bluetooth device address.

```
#!/usr/bin/python3
```

```
#
# Connects to a specified device
# Run from the command line with a bluetooth device address argument

import bluetooth_constants
import bluetooth_utils
import dbus
import sys
import time
sys.path.insert(0, '.')

bus = None
device_interface = None

def connect():
    global bus
    global device_interface
    try:
        device_interface.Connect()
    except Exception as e:
        print("Failed to connect")
        print(e.get_dbus_name())
        print(e.get_dbus_message())
        if ("UnknownObject" in e.get_dbus_name()):
            print("Try scanning first to resolve this problem")
        return bluetooth_constants.RESULT_EXCEPTION
    else:
        print("Connected OK")
        return bluetooth_constants.RESULT_OK

if (len(sys.argv) != 2):
    print("usage: python3 client_discover_services.py [bdaddr]")
    sys.exit(1)

bdaddr = sys.argv[1]
bus = dbus.SystemBus()
adapter_path = bluetooth_constants.BLUEZ_NAMESPACE +
bluetooth_constants.ADAPTER_NAME
device_path = bluetooth_utils.device_address_to_path(bdaddr, adapter_path)
device_proxy =
bus.get_object(bluetooth_constants.BLUEZ_SERVICE_NAME,device_path)
device_interface = dbus.Interface(device_proxy,
bluetooth_constants.DEVICE_INTERFACE)

print("Connecting to " + bdaddr)
connect()
```

```
pi@raspberrypi:~/projects/ldsg/solutions/python/bluetooth $ python3 client_connect_only.py
EB:EE:7B:08:EC:3D
Connecting to EB:EE:7B:08:EC:3D
Connected OK
```

5.3 Tracking Service Discovery

Now we'll modify the code so that the discovery of services, characteristics and descriptors is reported on.

To track the service discovery process, we need to register for and handle InterfacesAdded signals and process those that relate to objects that implement the org.bluez.GattService1, org.bluez.GattCharacteristic1 or org.bluez.GattDescriptor1 interface.

Have a go at implementing this in your code on your own now. You already know all you need to know to be able to do this.

If you need assistance or want to compare your code with another implementation, here's one possible solution:

```
#!/usr/bin/python3
#
# Connects to a specified device
# Run from the command line with a bluetooth device address argument

import bluetooth_constants
import bluetooth_utils
import dbus
import dbus.mainloop.glib
import sys
import time
from gi.repository import GLib
sys.path.insert(0, '.')

bus = None
device_interface = None

def interfaces_added(path, interfaces):

    if bluetooth_constants.GATT_SERVICE_INTERFACE in interfaces:
        properties = interfaces[bluetooth_constants.GATT_SERVICE_INTERFACE]
        print("------------------------------------------------------------
-------------------")
        print("SVC path   :", path)
        if 'UUID' in properties:
            uuid = properties['UUID']
            print("SVC UUID   : ", bluetooth_utils.dbus_to_python(uuid))
            print("SVC name   : ", bluetooth_utils.get_name_from_uuid(uuid))
        return

    if bluetooth_constants.GATT_CHARACTERISTIC_INTERFACE in interfaces:
```

```
        properties =
interfaces[bluetooth_constants.GATT_CHARACTERISTIC_INTERFACE]
        print("  CHR path   :", path)
        if 'UUID' in properties:
            uuid = properties['UUID']
            print("  CHR UUID   : ", bluetooth_utils.dbus_to_python(uuid))
            print("  CHR name   : ",
bluetooth_utils.get_name_from_uuid(uuid))
            flags  = ""
            for flag in properties['Flags']:
                flags = flags + flag + ","
            print("  CHR flags  : ", flags)
        return

    if bluetooth_constants.GATT_DESCRIPTOR_INTERFACE in interfaces:
        properties =
interfaces[bluetooth_constants.GATT_DESCRIPTOR_INTERFACE]
        print("    DSC path   :", path)
        if 'UUID' in properties:
            uuid = properties['UUID']
            print("    DSC UUID   : ", bluetooth_utils.dbus_to_python(uuid))
            print("    DSC name   : ",
bluetooth_utils.get_name_from_uuid(uuid))
        return

def connect():
    global bus
    global device_interface
    try:
        device_interface.Connect()
    except Exception as e:
        print("Failed to connect")
        print(e.get_dbus_name())
        print(e.get_dbus_message())
        if ("UnknownObject" in e.get_dbus_name()):
            print("Try scanning first to resolve this problem")
        return bluetooth_constants.RESULT_EXCEPTION
    else:
        print("Connected OK")
        return bluetooth_constants.RESULT_OK

if (len(sys.argv) != 2):
    print("usage: python3 client_discover_services.py [bdaddr]")
    sys.exit(1)

bdaddr = sys.argv[1]
dbus.mainloop.glib.DBusGMainLoop(set_as_default=True)
bus = dbus.SystemBus()
```

```
adapter_path = bluetooth_constants.BLUEZ_NAMESPACE +
bluetooth_constants.ADAPTER_NAME
device_path = bluetooth_utils.device_address_to_path(bdaddr, adapter_path)
device_proxy =
bus.get_object(bluetooth_constants.BLUEZ_SERVICE_NAME,device_path)
device_interface = dbus.Interface(device_proxy,
bluetooth_constants.DEVICE_INTERFACE)

print("Connecting to " + bdaddr)
connect()
print("Discovering services++")
print("Registering to receive InterfacesAdded signals")
bus.add_signal_receiver(interfaces_added,
        dbus_interface = bluetooth_constants.DBUS_OM_IFACE,
        signal_name = "InterfacesAdded")
mainloop = GLib.MainLoop()
mainloop.run()
```

5.4 Validating Services Found

Next, we'll do two things to improve our service discovery code. We'll check for the discovery of one particular service and one particular characteristic when service discovery has finished and we'll exit completely after that.

How do we know when service discovery has finished? The answer is that BlueZ will tell us if we ask it to. When service discovery completes, BlueZ emits a PropertiesChanged signal from the device object for a property called ServicesResolved which has a boolean value. So, all we need to do is to register for PropertiesChanged signals and watch for this property being signalled. Once we've received it we can validate the results and then exit.

Here's an example of a ServicesResolved property change signal at the end of service discovery:

```
signal time=1633330888.262144 sender=:1.15 -> destination=(null destination) serial=327
path=/org/bluez/hci0/dev_C1_6D_4A_98_A0_36; interface=org.freedesktop.DBus.Properties;
member=PropertiesChanged
   string "org.bluez.Device1"
   array [
      dict entry(
         string "UUIDs"
         variant             array [
               string "00001800-0000-1000-8000-00805f9b34fb"
               string "00001801-0000-1000-8000-00805f9b34fb"
               string "0000180a-0000-1000-8000-00805f9b34fb"
               string "e95d6100-251d-470a-a062-fa1922dfa9a8"
               string "e95d93af-251d-470a-a062-fa1922dfa9a8"
               string "e95d93b0-251d-470a-a062-fa1922dfa9a8"
               string "e95dd91d-251d-470a-a062-fa1922dfa9a8"
               string "e97dd91d-251d-470a-a062-fa1922dfa9a8"
            ]
      )
      dict entry(
         string "ServicesResolved"
         variant             boolean true
      )
   ]
   array [
   ]
```

Note that for good measure, we're also provided with an array of the UUIDs of all discovered GATT services.

Change your code to note the discovery of a service and characteristic of your choice (a mobile app called nRF Connect for iOS or Android may help you get to know your test device) and store the associated path values and handle PropertiesChanged signals, watching for ServicesResolved=true being signalled. When this happens, check that your chosen service and characteristic was found on the connected device and exit.

Here's one possible solution. Note that this code watches for the discovery of the Device Information Service and its Model Number String characteristic.

```
#!/usr/bin/python3
#
# Connects to a specified device
# Run from the command line with a bluetooth device address argument

import bluetooth_constants
import bluetooth_utils
import dbus
import dbus.mainloop.glib
import sys
import time
from gi.repository import GLib
sys.path.insert(0, '.')

bus = None
device_interface = None
device_path = None
found_dis = False
found_mn  = False
dis_path = None
mn_path  = None

def service_discovery_completed():
    global found_dis
    global found_mn
    global dis_path
    global mn_path
    global bus

    if found_dis and found_mn:
        print("Required service and characteristic found - device is OK")
        print("Device Information service path: ",dis_path)
        print("Model Number String characteristic path: ",mn_path)
    else:
        print("Required service and characteristic were not found - device
is NOK")
        print("Device Information service found: ",str(found_dis))
        print("Device Name characteristic found: ",str(found_mn))
```

```
    bus.remove_signal_receiver(interfaces_added,"InterfacesAdded")
    bus.remove_signal_receiver(properties_changed,"PropertiesChanged")
    mainloop.quit()

def properties_changed(interface, changed, invalidated, path):
    global device_path
    if path != device_path:
        return

    if 'ServicesResolved' in changed:
        sr = bluetooth_utils.dbus_to_python(changed['ServicesResolved'])
        print("ServicesResolved  : ", sr)
        if sr == True:
            service_discovery_completed()


def interfaces_added(path, interfaces):
    global found_dis
    global found_mn
    global dis_path
    global mn_path
    if bluetooth_constants.GATT_SERVICE_INTERFACE in interfaces:
        properties = interfaces[bluetooth_constants.GATT_SERVICE_INTERFACE]
        print("------------------------------------------------------------
------------------")
        print("SVC path   :", path)
        if 'UUID' in properties:
            uuid = properties['UUID']
            if uuid == bluetooth_constants.DEVICE_INF_SVC_UUID:
                found_dis = True
                dis_path = path
            print("SVC UUID   : ", bluetooth_utils.dbus_to_python(uuid))
            print("SVC name   : ", bluetooth_utils.get_name_from_uuid(uuid))
        return

    if bluetooth_constants.GATT_CHARACTERISTIC_INTERFACE in interfaces:
        properties =
interfaces[bluetooth_constants.GATT_CHARACTERISTIC_INTERFACE]
        print("  CHR path   :", path)
        if 'UUID' in properties:
            uuid = properties['UUID']
            if uuid == bluetooth_constants.MODEL_NUMBER_UUID:
                found_mn = True
                mn_path = path
            print("  CHR UUID   : ", bluetooth_utils.dbus_to_python(uuid))
            print("  CHR name   : ",
bluetooth_utils.get_name_from_uuid(uuid))
            flags  = ""
```

```
            for flag in properties['Flags']:
                flags = flags + flag + ","
            print("  CHR flags  : ", flags)
        return

    if bluetooth_constants.GATT_DESCRIPTOR_INTERFACE in interfaces:
        properties =
interfaces[bluetooth_constants.GATT_DESCRIPTOR_INTERFACE]
        print("    DSC path   :", path)
        if 'UUID' in properties:
            uuid = properties['UUID']
            print("    DSC UUID   : ", bluetooth_utils.dbus_to_python(uuid))
            print("    DSC name   : ",
bluetooth_utils.get_name_from_uuid(uuid))
        return

def connect():
    global bus
    global device_interface
    try:
        device_interface.Connect()
    except Exception as e:
        print("Failed to connect")
        print(e.get_dbus_name())
        print(e.get_dbus_message())
        if ("UnknownObject" in e.get_dbus_name()):
            print("Try scanning first to resolve this problem")
        return bluetooth_constants.RESULT_EXCEPTION
    else:
        print("Connected OK")
        return bluetooth_constants.RESULT_OK

if (len(sys.argv) != 2):
    print("usage: python3 client_discover_services.py [bdaddr]")
    sys.exit(1)

bdaddr = sys.argv[1]
dbus.mainloop.glib.DBusGMainLoop(set_as_default=True)
bus = dbus.SystemBus()
adapter_path = bluetooth_constants.BLUEZ_NAMESPACE +
bluetooth_constants.ADAPTER_NAME
device_path = bluetooth_utils.device_address_to_path(bdaddr, adapter_path)
device_proxy =
bus.get_object(bluetooth_constants.BLUEZ_SERVICE_NAME,device_path)
device_interface = dbus.Interface(device_proxy,
bluetooth_constants.DEVICE_INTERFACE)

print("Connecting to " + bdaddr)
```

```
connect()
print("Discovering services++")
print("Registering to receive InterfacesAdded signals")
bus.add_signal_receiver(interfaces_added,
        dbus_interface = bluetooth_constants.DBUS_OM_IFACE,
        signal_name = "InterfacesAdded")
print("Registering to receive PropertiesChanged signals")
bus.add_signal_receiver(properties_changed,
        dbus_interface = bluetooth_constants.DBUS_PROPERTIES,
        signal_name = "PropertiesChanged",
        path_keyword = "path")
mainloop = GLib.MainLoop()
mainloop.run()
print("Finished")
```

Running this code (after separately performing device discovery) produced these results:

```
pi@raspberrypi:~/projects/ldsg/solutions/python/bluetooth $ python3
client_discover_services.py EB:EE:7B:08:EC:3D
Connecting to EB:EE:7B:08:EC:3D
Connected OK
Discovering services++
Registering to receive InterfacesAdded signals
Registering to receive PropertiesChanged signals
--------------------------------------------------------------------------------
SVC path   : /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D/service0008
SVC UUID   :  00001801-0000-1000-8000-00805f9b34fb
SVC name   :  Generic Attribute Service
  CHR path   : /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D/service0008/char0009
  CHR UUID   :  00002a05-0000-1000-8000-00805f9b34fb
  CHR name   :  Service Changed
  CHR flags  :  indicate,
    DSC path   : /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D/service0008/char0009/desc000b
    DSC UUID   :  00002902-0000-1000-8000-00805f9b34fb
    DSC name   :  Client Characteristic Configuration
--------------------------------------------------------------------------------
SVC path   : /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D/service000c
SVC UUID   :  e95d93b0-251d-470a-a062-fa1922dfa9a8
SVC name   :  DFU Control Service
  CHR path   : /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D/service000c/char000d
  CHR UUID   :  e95d93b1-251d-470a-a062-fa1922dfa9a8
  CHR name   :  DFU Control
  CHR flags  :  read,write,
--------------------------------------------------------------------------------
SVC path   : /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D/service000f
SVC UUID   :  e97dd91d-251d-470a-a062-fa1922dfa9a8
SVC name   :  Unknown
  CHR path   : /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D/service000f/char0010
  CHR UUID   :  e97d3b10-251d-470a-a062-fa1922dfa9a8
  CHR name   :  Unknown
  CHR flags  :  write-without-response,notify,
    DSC path   : /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D/service000f/char0010/desc0012
    DSC UUID   :  00002902-0000-1000-8000-00805f9b34fb
    DSC name   :  Client Characteristic Configuration
--------------------------------------------------------------------------------
SVC path   : /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D/service0013
SVC UUID   :  0000180a-0000-1000-8000-00805f9b34fb
SVC name   :  Device Information Service
  CHR path   : /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D/service0013/char0014
  CHR UUID   :  00002a24-0000-1000-8000-00805f9b34fb
  CHR name   :  Model Number String
  CHR flags  :  read,
  CHR path   : /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D/service0013/char0016
  CHR UUID   :  00002a25-0000-1000-8000-00805f9b34fb
  CHR name   :  Serial Number String
  CHR flags  :  read,
```

```
  CHR path   : /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D/service0013/char0018
  CHR UUID   :  00002a26-0000-1000-8000-00805f9b34fb
  CHR name   :  Firmware Revision String
  CHR flags  :  read,
--------------------------------------------------------------------------------
SVC path   : /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D/service001a
SVC UUID   :  e95d93af-251d-470a-a062-fa1922dfa9a8
SVC name   :  Event Service
  CHR path   : /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D/service001a/char001b
  CHR UUID   :  e95d9775-251d-470a-a062-fa1922dfa9a8
  CHR name   :  micro:bit Event
  CHR flags  :  read,notify,
    DSC path   : /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D/service001a/char001b/desc001d
    DSC UUID   :  00002902-0000-1000-8000-00805f9b34fb
    DSC name   :  Client Characteristic Configuration
  CHR path   : /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D/service001a/char001e
  CHR UUID   :  e95d5404-251d-470a-a062-fa1922dfa9a8
  CHR name   :  Client Event
  CHR flags  :  write-without-response,write,
  CHR path   : /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D/service001a/char0020
  CHR UUID   :  e95d23c4-251d-470a-a062-fa1922dfa9a8
  CHR name   :  Client Requirements
  CHR flags  :  write,
  CHR path   : /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D/service001a/char0022
  CHR UUID   :  e95db84c-251d-470a-a062-fa1922dfa9a8
  CHR name   :  micro:bit Requirements
  CHR flags  :  read,notify,
    DSC path   : /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D/service001a/char0022/desc0024
    DSC UUID   :  00002902-0000-1000-8000-00805f9b34fb
    DSC name   :  Client Characteristic Configuration
--------------------------------------------------------------------------------
SVC path   : /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D/service0025
SVC UUID   :  e95d9882-251d-470a-a062-fa1922dfa9a8
SVC name   :  Button Service
  CHR path   : /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D/service0025/char0026
  CHR UUID   :  e95dda90-251d-470a-a062-fa1922dfa9a8
  CHR name   :  Button A State
  CHR flags  :  read,notify,
    DSC path   : /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D/service0025/char0026/desc0028
    DSC UUID   :  00002902-0000-1000-8000-00805f9b34fb
    DSC name   :  Client Characteristic Configuration
  CHR path   : /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D/service0025/char0029
  CHR UUID   :  e95dda91-251d-470a-a062-fa1922dfa9a8
  CHR name   :  Button B State
  CHR flags  :  read,notify,
    DSC path   : /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D/service0025/char0029/desc002b
    DSC UUID   :  00002902-0000-1000-8000-00805f9b34fb
    DSC name   :  Client Characteristic Configuration
ServicesResolved  :  True
Required service and characteristic found - device is OK
Device Information service path:  /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D/service0013
Model Number String characteristic path:
/org/bluez/hci0/dev_EB_EE_7B_08_EC_3D/service0013/char0014
Finished
```

6. Reading Characteristics

In this section, we'll create a script which connects to a specified device and then reads and displays the value of a particular characteristic. Device discovery must be performed first in a separate step, for example by running the client_discover_devices.py script developed in section 3 in a separate terminal window.

When creating this exercise and as explained in the Introduction, a BBC micro:bit running the Temperature service was used. It includes the Temperature characteristic whose value is a signed 8-bit field which contains the current temperature (in fact the value is derived from the core CPU temperature so it may not be an accurate reflection of the ambient temperature in your room). You can use a different service and characteristic if you want to. Make sure the characteristic you choose supports the GATT Read procedure. From now on, we assume the Temperature service and characteristic are in use.

The UUID of the Temperature service is e95d6100-251d-470a-a062-fa1922dfa9a8.

The UUID of the Temperature characteristic is e95d9250-251d-470a-a062-fa1922dfa9a8.

With your device advertising, perform device discovery and connect to the device using your scripts or D-Feet. In D-Feet explore the service and characteristic objects associated with your test device. By double clicking on the UUID property of a service or characteristic, you can cause it to be read and displayed. In this way, locate the service and characteristic you intend to use for testing.

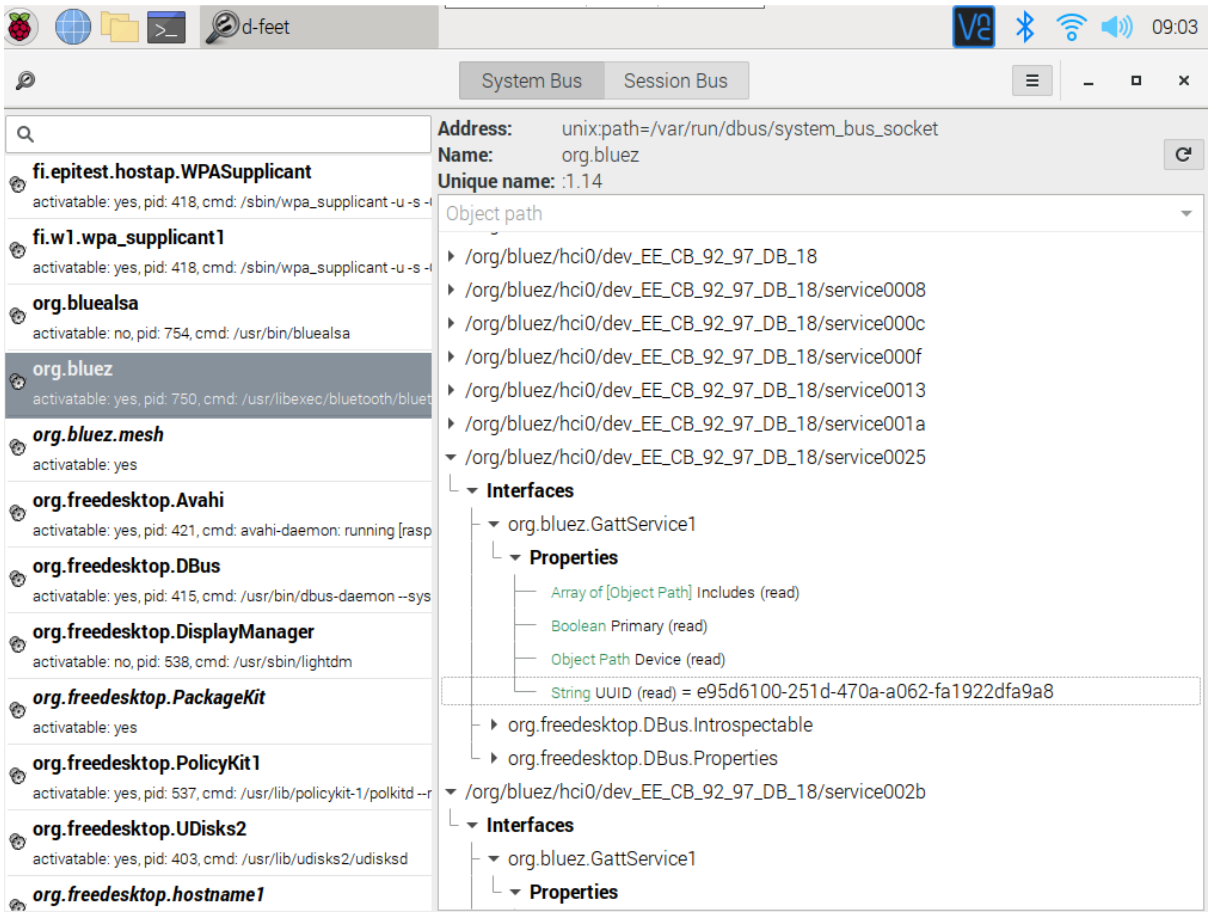

Figure 9 - service0025 has been found to represent the temperature service on this device

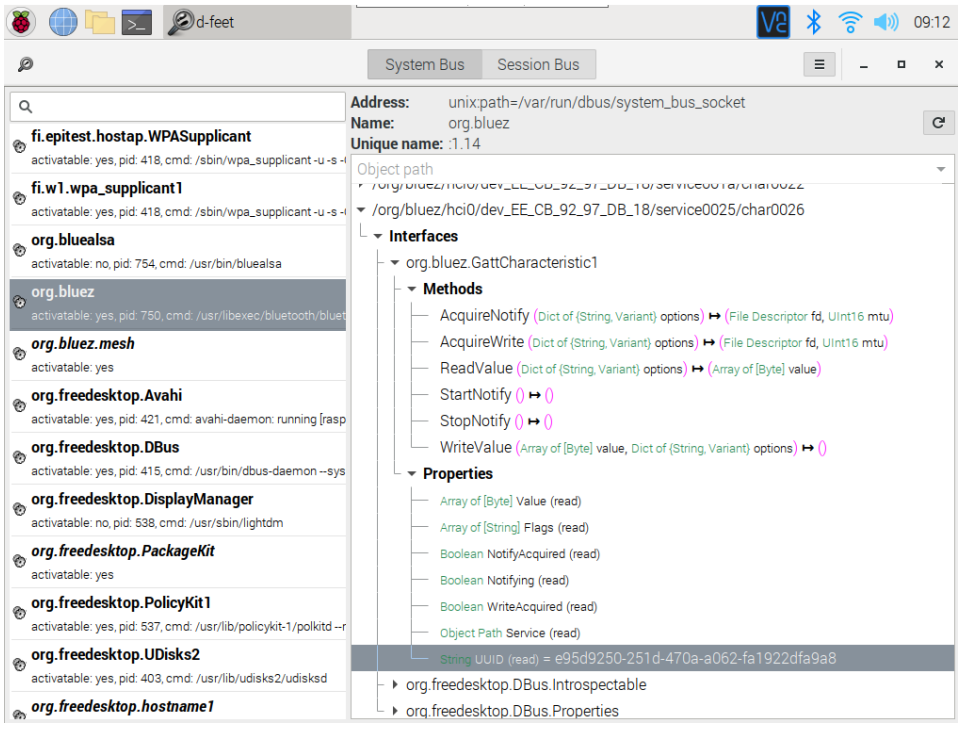

Figure 10 - char0026 within service0025 is the temperature characteristic on this device

Within the characteristic, double click the ReadValue method. Supply an input argument of {} signifying an empty dictionary of options and click Execute.

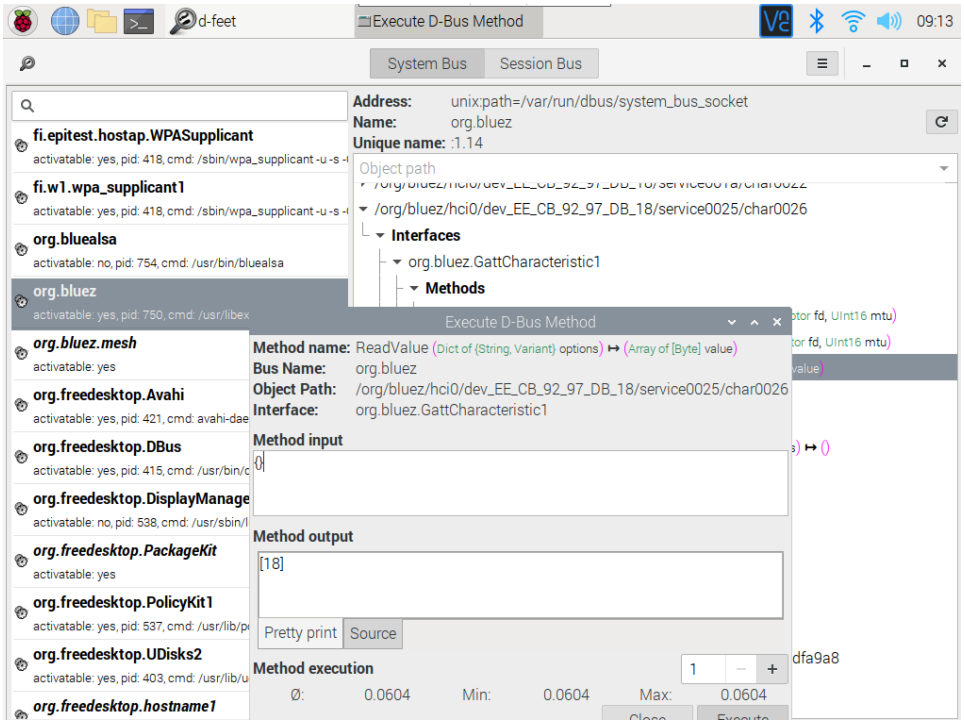

Figure 11 - Executing ReadValue within D-Feet

Output in the form of a byte array containing a single byte value will be displayed. In Figure 11 we can see 18 displayed, suggesting the temperature in the room is around 18 degrees Celsius.

We'll now create a script which reads the temperature characteristic and displays the result in much the same way.

Connecting and performing service discovery and validation are needed before we can read the characteristic so make a copy of the context of your client_discover_services.py script, calling the copy client_read_temperature.py.

6.1 Obtaining service and characteristic paths

Modify your new script so that it finds the paths for the temperature service and characteristic. Validate that they were found. This should be easy and result in code looking like this:

```
#!/usr/bin/python3
#
# Connects to a specified device
# Run from the command line with a bluetooth device address argument

import bluetooth_constants
import bluetooth_utils
import dbus
import dbus.mainloop.glib
import sys
import time
from gi.repository import GLib
sys.path.insert(0, '.')
```

```
bus = None
device_interface = None
device_path = None
found_ts = False
found_tc  = False
ts_path = None
tc_path  = None

def service_discovery_completed():
    global found_ts
    global found_tc
    global ts_path
    global tc_path
    global bus

    if found_ts and found_tc:
        print("Required service and characteristic found - device is OK")
        print("Temperature service path: ",ts_path)
        print("Temperature characteristic path: ",tc_path)
    else:
        print("Required service and characteristic were not found - device
is NOK")
        print("Temperature service found: ",str(found_ts))
        print("Temperature characteristic found: ",str(found_tc))
    bus.remove_signal_receiver(interfaces_added,"InterfacesAdded")
    bus.remove_signal_receiver(properties_changed,"PropertiesChanged")
    mainloop.quit()

def properties_changed(interface, changed, invalidated, path):
    global device_path
    if path != device_path:
        return

    if 'ServicesResolved' in changed:
        sr = bluetooth_utils.dbus_to_python(changed['ServicesResolved'])
        print("ServicesResolved  : ", sr)
        if sr == True:
            service_discovery_completed()


def interfaces_added(path, interfaces):
    global found_ts
    global found_tc
    global ts_path
    global tc_path
    if bluetooth_constants.GATT_SERVICE_INTERFACE in interfaces:
        properties = interfaces[bluetooth_constants.GATT_SERVICE_INTERFACE]
```

```
        print("------------------------------------------------------------
------------------")
        print("SVC path   :", path)
        if 'UUID' in properties:
            uuid = properties['UUID']
            if uuid == bluetooth_constants.TEMPERATURE_SVC_UUID:
                found_ts = True
                ts_path = path
            print("SVC UUID   : ", bluetooth_utils.dbus_to_python(uuid))
            print("SVC name   : ", bluetooth_utils.get_name_from_uuid(uuid))
        return

    if bluetooth_constants.GATT_CHARACTERISTIC_INTERFACE in interfaces:
        properties =
interfaces[bluetooth_constants.GATT_CHARACTERISTIC_INTERFACE]
        print("  CHR path   :", path)
        if 'UUID' in properties:
            uuid = properties['UUID']
            if uuid == bluetooth_constants.TEMPERATURE_CHR_UUID:
                found_tc = True
                tc_path = path
            print("  CHR UUID   : ", bluetooth_utils.dbus_to_python(uuid))
            print("  CHR name   : ",
bluetooth_utils.get_name_from_uuid(uuid))
            flags  = ""
            for flag in properties['Flags']:
                flags = flags + flag + ","
            print("  CHR flags  : ", flags)
        return

    if bluetooth_constants.GATT_DESCRIPTOR_INTERFACE in interfaces:
        properties =
interfaces[bluetooth_constants.GATT_DESCRIPTOR_INTERFACE]
        print("    DSC path   :", path)
        if 'UUID' in properties:
            uuid = properties['UUID']
            print("    DSC UUID   : ", bluetooth_utils.dbus_to_python(uuid))
            print("    DSC name   : ",
bluetooth_utils.get_name_from_uuid(uuid))
        return

def connect():
    global bus
    global device_interface
    try:
        device_interface.Connect()
    except Exception as e:
        print("Failed to connect")
```

```
        print(e.get_dbus_name())
        print(e.get_dbus_message())
        if ("UnknownObject" in e.get_dbus_name()):
            print("Try scanning first to resolve this problem")
        return bluetooth_constants.RESULT_EXCEPTION
    else:
        print("Connected OK")
        return bluetooth_constants.RESULT_OK

if (len(sys.argv) != 2):
    print("usage: python3 client_read_temperature.py [bdaddr]")
    sys.exit(1)

bdaddr = sys.argv[1]
dbus.mainloop.glib.DBusGMainLoop(set_as_default=True)
bus = dbus.SystemBus()
adapter_path = bluetooth_constants.BLUEZ_NAMESPACE +
bluetooth_constants.ADAPTER_NAME
device_path = bluetooth_utils.device_address_to_path(bdaddr, adapter_path)
device_proxy =
bus.get_object(bluetooth_constants.BLUEZ_SERVICE_NAME,device_path)
device_interface = dbus.Interface(device_proxy,
bluetooth_constants.DEVICE_INTERFACE)

print("Connecting to " + bdaddr)
connect()
print("Discovering services++")
print("Registering to receive InterfacesAdded signals")
bus.add_signal_receiver(interfaces_added,
        dbus_interface = bluetooth_constants.DBUS_OM_IFACE,
        signal_name = "InterfacesAdded")
print("Registering to receive PropertiesChanged signals")
bus.add_signal_receiver(properties_changed,
        dbus_interface = bluetooth_constants.DBUS_PROPERTIES,
        signal_name = "PropertiesChanged",
        path_keyword = "path")
mainloop = GLib.MainLoop()
mainloop.run()
print("Finished")
```

Test the modified code. It should output lines like these at the end:

```
ServicesResolved  :  True
Required service and characteristic found - device is OK
Temperature service path:  /org/bluez/hci0/dev_EE_CB_92_97_DB_18/service0025
Temperature characteristic path:
/org/bluez/hci0/dev_EE_CB_92_97_DB_18/service0025/char0026
Finished
```

6.2 Reading the temperature characteristic value

Next, modify the code so that after service discovery and validation has completed, execute the ReadValue method on the characteristic object (identified by its path) and display the returned value in decimal. You must pass an empty dictionary to the ReadValue method which in Python is indicated with *{}*.

Try this now, making sure you call ReadValue on the GattService1 interface of the characteristic object whose path you acquired during service discovery.

Here's the solution and test results:

```
#!/usr/bin/python3
#
# Connects to a specified device
# Run from the command line with a bluetooth device address argument

import bluetooth_constants
import bluetooth_utils
import dbus
import dbus.mainloop.glib
import sys
import time
from gi.repository import GLib
sys.path.insert(0, '.')

bus = None
device_interface = None
device_path = None
found_ts = False
found_tc  = False
ts_path = None
tc_path  = None

def read_temperature():
    global tc_path
    char_proxy =
bus.get_object(bluetooth_constants.BLUEZ_SERVICE_NAME,tc_path)
    char_interface = dbus.Interface(char_proxy,
bluetooth_constants.GATT_CHARACTERISTIC_INTERFACE)
    try:
        value = char_interface.ReadValue({})
    except Exception as e:
        print("Failed to read temperature")
        print(e.get_dbus_name())
        print(e.get_dbus_message())
        return bluetooth_constants.RESULT_EXCEPTION
    else:
        temperature = bluetooth_utils.dbus_to_python(value[0])
        print("Temperature="+str(temperature)+"C")
```

```
        return bluetooth_constants.RESULT_OK

def service_discovery_completed():
    global found_ts
    global found_tc
    global ts_path
    global tc_path
    global bus

    if found_ts and found_tc:
        print("Required service and characteristic found - device is OK")
        print("Temperature service path: ",ts_path)
        print("Temperature characteristic path: ",tc_path)
        read_temperature()
    else:
        print("Required service and characteristic were not found - device
is NOK")
        print("Temperature service found: ",str(found_ts))
        print("Temperature characteristic found: ",str(found_tc))
    bus.remove_signal_receiver(interfaces_added,"InterfacesAdded")
    bus.remove_signal_receiver(properties_changed,"PropertiesChanged")
    mainloop.quit()

def properties_changed(interface, changed, invalidated, path):
    global device_path
    if path != device_path:
        return

    if 'ServicesResolved' in changed:
        sr = bluetooth_utils.dbus_to_python(changed['ServicesResolved'])
        print("ServicesResolved  : ", sr)
        if sr == True:
            service_discovery_completed()


def interfaces_added(path, interfaces):
    global found_ts
    global found_tc
    global ts_path
    global tc_path
    if bluetooth_constants.GATT_SERVICE_INTERFACE in interfaces:
        properties = interfaces[bluetooth_constants.GATT_SERVICE_INTERFACE]
        print("----------------------------------------------------------------
-------------------")
        print("SVC path   :", path)
        if 'UUID' in properties:
            uuid = properties['UUID']
            if uuid == bluetooth_constants.TEMPERATURE_SVC_UUID:
```

```
                found_ts = True
                ts_path = path
            print("SVC UUID    : ", bluetooth_utils.dbus_to_python(uuid))
            print("SVC name    : ", bluetooth_utils.get_name_from_uuid(uuid))
        return

    if bluetooth_constants.GATT_CHARACTERISTIC_INTERFACE in interfaces:
        properties =
interfaces[bluetooth_constants.GATT_CHARACTERISTIC_INTERFACE]
        print("  CHR path    :", path)
        if 'UUID' in properties:
            uuid = properties['UUID']
            if uuid == bluetooth_constants.TEMPERATURE_CHR_UUID:
                found_tc = True
                tc_path = path
            print("  CHR UUID    : ", bluetooth_utils.dbus_to_python(uuid))
            print("  CHR name    : ",
bluetooth_utils.get_name_from_uuid(uuid))
            flags  = ""
            for flag in properties['Flags']:
                flags = flags + flag + ","
            print("  CHR flags   : ", flags)
        return

    if bluetooth_constants.GATT_DESCRIPTOR_INTERFACE in interfaces:
        properties =
interfaces[bluetooth_constants.GATT_DESCRIPTOR_INTERFACE]
        print("    DSC path    :", path)
        if 'UUID' in properties:
            uuid = properties['UUID']
            print("    DSC UUID    : ", bluetooth_utils.dbus_to_python(uuid))
            print("    DSC name    : ",
bluetooth_utils.get_name_from_uuid(uuid))
        return

def connect():
    global bus
    global device_interface
    try:
        device_interface.Connect()
    except Exception as e:
        print("Failed to connect")
        print(e.get_dbus_name())
        print(e.get_dbus_message())
        if ("UnknownObject" in e.get_dbus_name()):
            print("Try scanning first to resolve this problem")
        return bluetooth_constants.RESULT_EXCEPTION
    else:
```

```
        print("Connected OK")
        return bluetooth_constants.RESULT_OK

if (len(sys.argv) != 2):
    print("usage: python3 client_read_temperature.py [bdaddr]")
    sys.exit(1)

bdaddr = sys.argv[1]
dbus.mainloop.glib.DBusGMainLoop(set_as_default=True)
bus = dbus.SystemBus()
adapter_path = bluetooth_constants.BLUEZ_NAMESPACE +
bluetooth_constants.ADAPTER_NAME
device_path = bluetooth_utils.device_address_to_path(bdaddr, adapter_path)
device_proxy =
bus.get_object(bluetooth_constants.BLUEZ_SERVICE_NAME,device_path)
device_interface = dbus.Interface(device_proxy,
bluetooth_constants.DEVICE_INTERFACE)

print("Connecting to " + bdaddr)
connect()
print("Discovering services++")
print("Registering to receive InterfacesAdded signals")
bus.add_signal_receiver(interfaces_added,
        dbus_interface = bluetooth_constants.DBUS_OM_IFACE,
        signal_name = "InterfacesAdded")
print("Registering to receive PropertiesChanged signals")
bus.add_signal_receiver(properties_changed,
        dbus_interface = bluetooth_constants.DBUS_PROPERTIES,
        signal_name = "PropertiesChanged",
        path_keyword = "path")
mainloop = GLib.MainLoop()
mainloop.run()
print("Finished")
```

```
ServicesResolved  :  True
Required service and characteristic found - device is OK
Temperature service path:  /org/bluez/hci0/dev_EE_CB_92_97_DB_18/service0025
Temperature characteristic path:
/org/bluez/hci0/dev_EE_CB_92_97_DB_18/service0025/char0026
Temperature=21C
Finished
```

7. Writing Characteristics

In this section, we'll create a script which connects to a specified device and then writes a value to a characteristic. Device discovery must be performed first in a separate step, for example by running the client_discover_devices.py script developed in section 3 in a separate terminal window.

When creating this exercise and as explained in the Introduction, a BBC micro:bit running the LED service was used. It includes the LED Text characteristic whose value is an array of ASCII character codes which when written to will cause the corresponding characters to scroll across the LED matrix on the micro:bit. You can use a different service and characteristic if you want to. Make sure the characteristic you choose supports Write Requests. From now on, we assume the LED service and LED Text characteristic are in use.

The UUID of the LED service is e95dd91d-251d-470a-a062-fa1922dfa9a8.

The UUID of the LED Text characteristic is e95d93ee-251d-470a-a062-fa1922dfa9a8.

Either review the BlueZ documentation for the characteristic WriteValue method or use D-Feet to explore your device, find the characteristic to be written to and review the WriteValue method signature. You will see that it takes two input arguments, the first the value as a byte array and the second a dictionary of options. Using D-Feet, connect to your device and test writing to a suitable characteristic. Using a micro:bit and writing value of [72, 101, 108, 108, 111] with an empty dictionary of options to the LED Text characteristic you will see the message "Hello" scroll across the display.

7.1 Writing to the LED Text characteristic from Python

Copy your client_read_temperature.py script to a new file, client_write_text.py. Modify it to search for, save and validate the presence of the LED service and LED Text characteristic. Then replace the function which reads the temperature with one which will write a short string of at most 20 ASCII characters to the LED Text characteristic. Hard code the value or allow it to be passed from the command line as an argument. The only issues you need to watch out for here is to ensure you pass an array of integer ASCII codes as the first argument to WriteValue rather than a string.

You know everything you need to know to complete this exercise already.

Here's the solution:

```
#!/usr/bin/python3
#
# Connects to a specified device and writes a short string to the LED Text
characteristic of a BBC micro:bit.
# Run from the command line with a bluetooth device address argument

import bluetooth_constants
import bluetooth_utils
import dbus
import dbus.mainloop.glib
import sys
import time
from gi.repository import GLib
sys.path.insert(0, '.')
```

```
bus = None
device_interface = None
device_path = None
found_ls = False
found_lc  = False
ls_path = None
lc_path  = None
text = None

def write_text(text):
    global lc_path
    char_proxy =
bus.get_object(bluetooth_constants.BLUEZ_SERVICE_NAME,lc_path)
    char_interface = dbus.Interface(char_proxy,
bluetooth_constants.GATT_CHARACTERISTIC_INTERFACE)
    try:
        ascii = bluetooth_utils.text_to_ascii_array(text)
        value = char_interface.WriteValue(ascii, {})
    except Exception as e:
        print("Failed to write to LED Text")
        print(e.get_dbus_name())
        print(e.get_dbus_message())
        return bluetooth_constants.RESULT_EXCEPTION
    else:
        print("LED Text written OK")
        return bluetooth_constants.RESULT_OK

def service_discovery_completed():
    global found_ls
    global found_lc
    global ls_path
    global lc_path
    global bus
    global text

    if found_ls and found_lc:
        print("Required service and characteristic found - device is OK")
        print("LED service path: ",ls_path)
        print("LED characteristic path: ",lc_path)
        write_text(text)
    else:
        print("Required service and characteristic were not found - device
is NOK")
        print("LED service found: ",str(found_ls))
        print("LED Text characteristic found: ",str(found_lc))
    bus.remove_signal_receiver(interfaces_added,"InterfacesAdded")
    bus.remove_signal_receiver(properties_changed,"PropertiesChanged")
```

```
    mainloop.quit()

def properties_changed(interface, changed, invalidated, path):
    global device_path
    if path != device_path:
        return

    if 'ServicesResolved' in changed:
        sr = bluetooth_utils.dbus_to_python(changed['ServicesResolved'])
        print("ServicesResolved  : ", sr)
        if sr == True:
            service_discovery_completed()


def interfaces_added(path, interfaces):
    global found_ls
    global found_lc
    global ls_path
    global lc_path
    if bluetooth_constants.GATT_SERVICE_INTERFACE in interfaces:
        properties = interfaces[bluetooth_constants.GATT_SERVICE_INTERFACE]
        print("----------------------------------------------------------------
------------------")
        print("SVC path   :", path)
        if 'UUID' in properties:
            uuid = properties['UUID']
            if uuid == bluetooth_constants.LED_SVC_UUID:
                found_ls = True
                ls_path = path
            print("SVC UUID   : ", bluetooth_utils.dbus_to_python(uuid))
            print("SVC name   : ", bluetooth_utils.get_name_from_uuid(uuid))
        return

    if bluetooth_constants.GATT_CHARACTERISTIC_INTERFACE in interfaces:
        properties =
interfaces[bluetooth_constants.GATT_CHARACTERISTIC_INTERFACE]
        print("  CHR path   :", path)
        if 'UUID' in properties:
            uuid = properties['UUID']
            if uuid == bluetooth_constants.LED_TEXT_CHR_UUID:
                found_lc = True
                lc_path = path
            print("  CHR UUID   : ", bluetooth_utils.dbus_to_python(uuid))
            print("  CHR name   : ",
bluetooth_utils.get_name_from_uuid(uuid))
            flags  = ""
            for flag in properties['Flags']:
                flags = flags + flag + ","
```

```
            print("  CHR flags  : ", flags)
        return

    if bluetooth_constants.GATT_DESCRIPTOR_INTERFACE in interfaces:
        properties =
interfaces[bluetooth_constants.GATT_DESCRIPTOR_INTERFACE]
        print("    DSC path   :", path)
        if 'UUID' in properties:
            uuid = properties['UUID']
            print("    DSC UUID   : ", bluetooth_utils.dbus_to_python(uuid))
            print("    DSC name   : ",
bluetooth_utils.get_name_from_uuid(uuid))
        return

def connect():
    global bus
    global device_interface
    try:
        device_interface.Connect()
    except Exception as e:
        print("Failed to connect")
        print(e.get_dbus_name())
        print(e.get_dbus_message())
        if ("UnknownObject" in e.get_dbus_name()):
            print("Try scanning first to resolve this problem")
        return bluetooth_constants.RESULT_EXCEPTION
    else:
        print("Connected OK")
        return bluetooth_constants.RESULT_OK

if (len(sys.argv) != 3):
    print("usage: python3 client_write_text.py [bdaddr] [text]")
    sys.exit(1)

bdaddr = sys.argv[1]
text = sys.argv[2]
dbus.mainloop.glib.DBusGMainLoop(set_as_default=True)
bus = dbus.SystemBus()
adapter_path = bluetooth_constants.BLUEZ_NAMESPACE +
bluetooth_constants.ADAPTER_NAME
device_path = bluetooth_utils.device_address_to_path(bdaddr, adapter_path)
device_proxy =
bus.get_object(bluetooth_constants.BLUEZ_SERVICE_NAME,device_path)
device_interface = dbus.Interface(device_proxy,
bluetooth_constants.DEVICE_INTERFACE)

print("Connecting to " + bdaddr)
connect()
```

```
print("Discovering services++")
print("Registering to receive InterfacesAdded signals")
bus.add_signal_receiver(interfaces_added,
        dbus_interface = bluetooth_constants.DBUS_OM_IFACE,
        signal_name = "InterfacesAdded")
print("Registering to receive PropertiesChanged signals")
bus.add_signal_receiver(properties_changed,
        dbus_interface = bluetooth_constants.DBUS_PROPERTIES,
        signal_name = "PropertiesChanged",
        path_keyword = "path")
mainloop = GLib.MainLoop()
mainloop.run()
print("Finished")
```

Test results:

```
pi@raspberrypi:~/projects/ldsg/solutions/python/bluetooth $ python3 client_write_text.py
EE:CB:92:97:DB:18 Hello
Connecting to EE:CB:92:97:DB:18
Connected OK
Discovering services++
Registering to receive InterfacesAdded signals
Registering to receive PropertiesChanged signals
....
....
....
ServicesResolved  :  True
Required service and characteristic found - device is OK
LED service path:  /org/bluez/hci0/dev_EE_CB_92_97_DB_18/service002b
LED characteristic path:  /org/bluez/hci0/dev_EE_CB_92_97_DB_18/service002b/char002e
LED Text written OK
Finished
```

And most importantly, the text supplied on the command line scrolls across the micro:bit display.

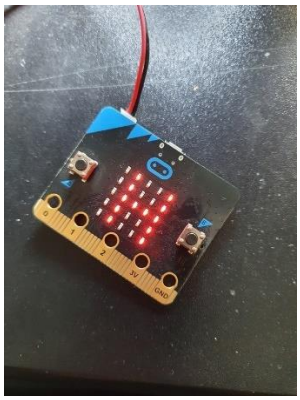

Figure 12 - Text scrolling across the micro:bit display

7.2 Write Requests vs Write Commands

Two variants of write operation are defined in the Bluetooth Attribute Protocol namely the *write request* and the *write command.* The write request PDU (ATT_WRITE_REQ) should be responded to by the GATT server that receives it with an ATT_WRITE_RSP PDU. The ATT_WRITE_CMD PDU on the other hand has no corresponding response PDU. Request/response pairs are more reliable than commands since the response acts as an acknowledgement and confirms that not only did the

request PDU make it to the other device over the link (which is already confirmed by an ACK at the link layer) but that it made it up the stack to the application too. An issue like buffer overflow could have caused the PDU to be discarded by the receiving device but the response PDU provides reassurance that this did not happen. Unfortunately, what we gain in reliability, we lose in throughput and a series of write requests will be relatively slow to process compared to a similar series of write commands, which do not require responses to be sent back in reply.

The BlueZ WriteValue API allows the type of write operation to be specified in its options argument which is of type dictionary. By including an option with key="type" and value="command", you can indicate that a write command must be performed. A value of "request" stipulates that a request must be performed.

You should only need to explicitly specify the type of write to perform when a characteristic supports both operations.

8. Using Notifications

In this section, we'll create a script which subscribes to temperature characteristic notifications and displays each temperature measurement to the console as it arrives. Naturally, we'll use the client_read_temperature.py script as a start-point.

The UUID of the Temperature service is e95d6100-251d-470a-a062-fa1922dfa9a8.

The UUID of the Temperature characteristic is e95d9250-251d-470a-a062-fa1922dfa9a8.

Review Figure 10 and you'll notice the methods StartNotify and StopNotify that belong to the GattCharacteristic1 interface. These methods do what their names suggest. When BlueZ receives a characteristic notification from a connected device, it sends a PropertiesChanged signal from the associated characteristic, identified by its path. All connected D-Bus services that have subscribed to this signal will receive it.

Therefore, to enable and process notifications, after finding the path for the relevant characteristic, we need to subscribe to its PropertiesChanged signal and then call its StartNotify method.

8.1 Creating the client_monitor_temperature.py script

Create a new script called client_monitor_temperature.py and copy the content of client_read_temperature.py into it to start things off. Change this code so that after connecting and completing service discovery, it registers to receive the PropertiesChanged signal when emitted by the temperature characteristic. Then call the StartNotify method of the characteristic. There are no arguments to this method and it does not return a value.

Signals containing temperature notifications look like this:

```
signal time=1635773453.656211 sender=:1.14 -> destination=(null destination) serial=3523
path=/org/bluez/hci0/dev_EE_CB_92_97_DB_18/service0025/char0026;
interface=org.freedesktop.DBus.Properties; member=PropertiesChanged
   string "org.bluez.GattCharacteristic1"
   array [
      dict entry(
         string "Value"
         variant             array of bytes [
               13
            ]
      )
   ]
```

```
    array [
    ]
```

As you can see, the signal sends an array of dictionary items. The one we want has a key of “Value” and a value which is a variant which wraps an array of bytes. dbus-monitor shows this value in hex and D-Feet shows in in decimal.

Implement a handler for the characteristic’s PropertiesChanged signal and in that function, check for an item with key equal to “Value”. If found, extract the byte array value part and display the first byte value from that array as the current temperature.

Here’s a solution with changes made to the client_read_temperature.py script in red:

```
#!/usr/bin/python3
#
# Connects to a specified device, starts temperature characteristic
notifications and logs values
# to the console as they are received in PropertiesChanged signals.
#
# Run from the command line with a bluetooth device address argument

import bluetooth_constants
import bluetooth_utils
import dbus
import dbus.mainloop.glib
import sys
import time
from gi.repository import GLib
sys.path.insert(0, '.')

bus = None
device_interface = None
device_path = None
found_ts = False
found_tc  = False
ts_path = None
tc_path  = None

def temperature_received(interface, changed, invalidated, path):
    if 'Value' in changed:
        temperature = bluetooth_utils.dbus_to_python(changed['Value'])
        print("temperature: "+ str(temperature[0]) + "C")

def start_notifications():
    global tc_path
    global bus
    char_proxy =
bus.get_object(bluetooth_constants.BLUEZ_SERVICE_NAME,tc_path)
    char_interface = dbus.Interface(char_proxy,
bluetooth_constants.GATT_CHARACTERISTIC_INTERFACE)
```

```
    bus.add_signal_receiver(temperature_received,
        dbus_interface = bluetooth_constants.DBUS_PROPERTIES,
        signal_name = "PropertiesChanged",
        path = tc_path,
        path_keyword = "path")

    try:
        print("Starting notifications")
        char_interface.StartNotify()
        print("Done starting notifications")
    except Exception as e:
        print("Failed to start temperature notifications")
        print(e.get_dbus_name())
        print(e.get_dbus_message())
        return bluetooth_constants.RESULT_EXCEPTION
    else:
        return bluetooth_constants.RESULT_OK

def service_discovery_completed():
    global found_ts
    global found_tc
    global ts_path
    global tc_path
    global bus

    if found_ts and found_tc:
        print("Required service and characteristic found - device is OK")
        print("Temperature service path: ",ts_path)
        print("Temperature characteristic path: ",tc_path)
        start_notifications()
    else:
        print("Required service and characteristic were not found - device
is NOK")
        print("Temperature service found: ",str(found_ts))
        print("Temperature characteristic found: ",str(found_tc))
    bus.remove_signal_receiver(interfaces_added,"InterfacesAdded")
    bus.remove_signal_receiver(properties_changed,"PropertiesChanged")
#    mainloop.quit()

def properties_changed(interface, changed, invalidated, path):
    global device_path
    if path != device_path:
        return

    if 'ServicesResolved' in changed:
        sr = bluetooth_utils.dbus_to_python(changed['ServicesResolved'])
        print("ServicesResolved  : ", sr)
```

```
        if sr == True:
            service_discovery_completed()


def interfaces_added(path, interfaces):
    global found_ts
    global found_tc
    global ts_path
    global tc_path
    if bluetooth_constants.GATT_SERVICE_INTERFACE in interfaces:
        properties = interfaces[bluetooth_constants.GATT_SERVICE_INTERFACE]
        print("------------------------------------------------------------
-------------------")
        print("SVC path   :", path)
        if 'UUID' in properties:
            uuid = properties['UUID']
            if uuid == bluetooth_constants.TEMPERATURE_SVC_UUID:
                found_ts = True
                ts_path = path
            print("SVC UUID   : ", bluetooth_utils.dbus_to_python(uuid))
            print("SVC name   : ", bluetooth_utils.get_name_from_uuid(uuid))
        return

    if bluetooth_constants.GATT_CHARACTERISTIC_INTERFACE in interfaces:
        properties =
interfaces[bluetooth_constants.GATT_CHARACTERISTIC_INTERFACE]
        print("  CHR path   :", path)
        if 'UUID' in properties:
            uuid = properties['UUID']
            if uuid == bluetooth_constants.TEMPERATURE_CHR_UUID:
                found_tc = True
                tc_path = path
            print("  CHR UUID   : ", bluetooth_utils.dbus_to_python(uuid))
            print("  CHR name   : ",
bluetooth_utils.get_name_from_uuid(uuid))
            flags  = ""
            for flag in properties['Flags']:
                flags = flags + flag + ","
            print("  CHR flags  : ", flags)
        return

    if bluetooth_constants.GATT_DESCRIPTOR_INTERFACE in interfaces:
        properties =
interfaces[bluetooth_constants.GATT_DESCRIPTOR_INTERFACE]
        print("    DSC path   :", path)
        if 'UUID' in properties:
            uuid = properties['UUID']
            print("    DSC UUID   : ", bluetooth_utils.dbus_to_python(uuid))
```

```
            print("    DSC name   : ",
bluetooth_utils.get_name_from_uuid(uuid))
        return

def connect():
    global bus
    global device_interface
    try:
        device_interface.Connect()
    except Exception as e:
        print("Failed to connect")
        print(e.get_dbus_name())
        print(e.get_dbus_message())
        if ("UnknownObject" in e.get_dbus_name()):
            print("Try scanning first to resolve this problem")
        return bluetooth_constants.RESULT_EXCEPTION
    else:
        print("Connected OK")
        return bluetooth_constants.RESULT_OK

if (len(sys.argv) != 2):
    print("usage: python3 client_monitor_temperature.py [bdaddr]")
    sys.exit(1)

bdaddr = sys.argv[1]
dbus.mainloop.glib.DBusGMainLoop(set_as_default=True)
bus = dbus.SystemBus()
adapter_path = bluetooth_constants.BLUEZ_NAMESPACE +
bluetooth_constants.ADAPTER_NAME
device_path = bluetooth_utils.device_address_to_path(bdaddr, adapter_path)
device_proxy =
bus.get_object(bluetooth_constants.BLUEZ_SERVICE_NAME,device_path)
device_interface = dbus.Interface(device_proxy,
bluetooth_constants.DEVICE_INTERFACE)

print("Connecting to " + bdaddr)
connect()
print("Discovering services++")
print("Registering to receive InterfacesAdded signals")
bus.add_signal_receiver(interfaces_added,
        dbus_interface = bluetooth_constants.DBUS_OM_IFACE,
        signal_name = "InterfacesAdded")
print("Registering to receive PropertiesChanged signals")
bus.add_signal_receiver(properties_changed,
        dbus_interface = bluetooth_constants.DBUS_PROPERTIES,
        signal_name = "PropertiesChanged",
        path_keyword = "path")
mainloop = GLib.MainLoop()
```

```
mainloop.run()
```

And test results should look like this:

```
pi@raspberrypi:~/projects/ldsg/solutions/python/bluetooth $ python3
client_monitor_temperature.py EE:CB:92:97:DB:18
Connecting to EE:CB:92:97:DB:18
Connected OK
Discovering services++
Registering to receive InterfacesAdded signals
Registering to receive PropertiesChanged signals
...
...
...
...
ServicesResolved  :  True
Required service and characteristic found - device is OK
Temperature service path:  /org/bluez/hci0/dev_EE_CB_92_97_DB_18/service0025
Temperature characteristic path:
/org/bluez/hci0/dev_EE_CB_92_97_DB_18/service0025/char0026
Starting notifications
Done starting notifications
temperature: 20C
temperature: 20C
temperature: 19C
temperature: 19C
temperature: 19C
temperature: 19C
temperature: 19C
...
...
...
```

The script will continue to display notification values as long as it is running.

9. Summary

That's the end of this module. You've learned about and had an opportunity to write code relating to device discovery, connecting and disconnecting from a device, performing service discovery, reading and writing characteristics and enabling and handling characteristic notifications.

Hopefully you are now confident that you can apply the basics of D-Bus programming learned in module 04 to the development of Bluetooth LE Central code using BlueZ.