Bluetooth for Linux Developers Study Guide

Orientation Guide

Release	:	1.0.1
Document Version:		1.0.0
Last updated	:	16th November 2021

Contents

1. REVISION HISTORY **3**

2. INTRODUCTION AND ORIENTATION **4**

3. GOALS **4**

4. MODULE SEQUENCE **5**

5. EQUIPMENT REQUIRED **5**

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.
1.0.1	11th January 2022	Martin Woolley Bluetooth SIG	**Release:** **Document**: Fixed typo in list of modules. Module 04 was listed as Module 04.

2. Introduction and Orientation

Welcome to the Bluetooth® for Linux Developers Study Guide.

This is a self-paced, educational resource for software developers. It aims to introduce the reader to the architecture, concepts and code required to exploit Bluetooth on a Linux device using the BlueZ Bluetooth stack.

The study guide is modular in design and this orientation guide will help you decide where to start and what path to follow, depending on your current knowledge and goals.

Module 02 - Bluetooth LE Primer introduces those aspects of Bluetooth Low Energy (LE) that must be understood for the remainder of this resource to make sense.

Module 03 - Linux and Bluetooth explains BlueZ, the Bluetooth stack for Linux in terms of its architecture, interfaces and tools commonly used. It also explains how to install and configure BlueZ ready for use in the modules which follow. Finally, languages and language bindings available for developing Bluetooth applications on Linux are considered and evaluated.

Module 04 - Mastering DBus Basics using Python covers implementing DBus applications using Python but does not consider Bluetooth. This allows the reader to master the basics of DBus programming without needing to also learn about those issues that relate specifically to Bluetooth.

Module 05 - Developing LE Central Devices using Python explains how to write Python code which acts in the Bluetooth GAP Central role and after connecting to a remote device, as a GATT client. If the terms *GAP Central* and *GATT client* don't mean anything to you, don't worry because they're explained in module 02. Example code is included and you'll also have the opportunity to write code of your own and test it on a suitable computer.

Module 06 - Developing LE Peripheral Devices using Python explains how to write code which acts in the Bluetooth GAP Peripheral role, advertises and after being connecting to by a remote device, as a GATT server. The terms *GAP Peripheral* and *GATT server* are also explained in module 02.

Module A1 - Installation and Configuration provides step by step instructions for installing and setting up BlueZ for use with either GAP/GATT applications or Bluetooth mesh node development. Offers some tips on troubleshooting.

3. Goals

After completing the work in this study guide, you should:

- Be able to explain basic Bluetooth LE concepts and terminology such as *GAP Central* and *GATT client*.
- Be able to explain what BlueZ is, how applications use BlueZ in terms of architecture, services and communication.
- Understand the fundamentals of developing applications which use DBus inter-process communication.
- Be able to implement key functionality, typically required by GAP Central/GATT client Bluetooth devices.

4. Module Sequence

The modules are designed to be followed in a largely linear sequence but obviously if you are already familiar with the subject of a module you should skip it and dive straight into the first module that you think will offer something of value to you. If you're only interested in LE Peripheral development then it's not strictly necessary to study module 05 on Central device development but it's still recommended. If you're only interested in Central device development then it's not necessary to study Peripheral development but once again, it's recommended.

Note that module A1 acts as an appendix to the main study guide modules and contains information about setting up your Linux environment and some tips on tools that might be useful in troubleshooting.

Good luck with your studies!

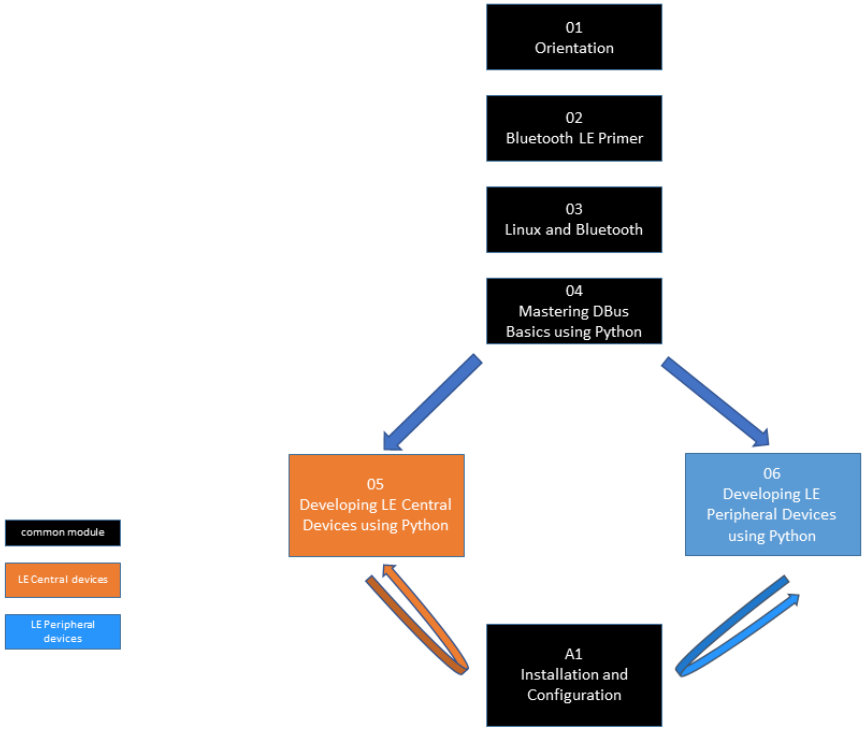


Figure 1 - Module order and dependencies

5. Equipment Required

To get the most out of this resource you should go *hands-on* whenever the opportunity arises and execute suggested commands, run all provided code examples and complete the coding exercises. To do so you will need a Linux computer which has a Bluetooth adapter either on-board or plugged into it, perhaps in the form of a Bluetooth USB dongle.

This study guide was developed using a Raspberry Pi 4.

What you use to edit code is up to you. All that's required is a text editor and we all have our favourites.

Bluetooth for Linux Developers Study Guide

Bluetooth LE Primer

Release : 1.0.1

Document Version: 1.0.1

Last updated : 18th October 2022

Contents

1. REVISION HISTORY 3

2. INTRODUCTION 4

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.
1.0.1	18th October 2022	Martin Woolley Bluetooth SIG	Replaced material on basic Bluetooth LE concepts with a recommendation to download and review the Bluetooth LE Primer resource.

2. Introduction

The Bluetooth Low Energy Primer is a separate resource which explains every layer of the Bluetooth LE stack, starting with the physical layer at the bottom and ending with the generic access profile (GAP) at the top. This is the place to start if you're new to Bluetooth LE. Topics of most relevance to this study guide include the generic access profile (GAP), generic attribute profile (GATT) and the attribute protocol (ATT).

You can download the Bluetooth Low Energy Primer from https://www.bluetooth.com/bluetooth-resources/the-bluetooth-low-energy-primer/

When you're happy you understand GAP, GATT and ATT, move on to the next module of this study guide.

Bluetooth for Linux Developers Study Guide

Linux and Bluetooth

Release : 1.0.1

Document Version: 1.0.0

Last updated : 16th November 2021

Contents

1. REVISION HISTORY 3

2. INTRODUCTION 4

3. LINUX AND BLUETOOTH ARCHITECTURE 4

4. BLUEZ AND D-BUS 6

4.1 D-Bus Concepts 6

4.1.1 Message Buses 6

4.1.2 Clients, Servers and Connections 6

4.1.3 Objects, Interfaces, Methods, Signals and Properties 7

4.1.3 Proxy Objects 7

4.1.4 Well-known Names 8

4.1.5 Standard Interfaces 8

4.1.6 Data Types 8

4.1.7 Introspection 8

4.1.8 Security Policies 9

4.2 D-Bus Tools 9

4.2.1 D-feet 9

4.2.2 dbus-monitor 10

4.2.3 dbus-send 11

4.3 Examples 11

4.3.1 A D-Bus Method Call Message 11

4.3.2 A D-Bus Signal Message 11

5. PROGRAMMING LANGUAGES AND APIS 12

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

In this module we'll look at Bluetooth within Linux from an architectural point of view and how applications that use Bluetooth communicate with the stack. We'll also review programming language and API options.

3. Linux and Bluetooth Architecture

The Bluetooth Low Energy stack is split into two major architectural blocks known as the Host and the Controller. The stack and the distribution of layers across the host and controller parts is depicted in Figure 1.

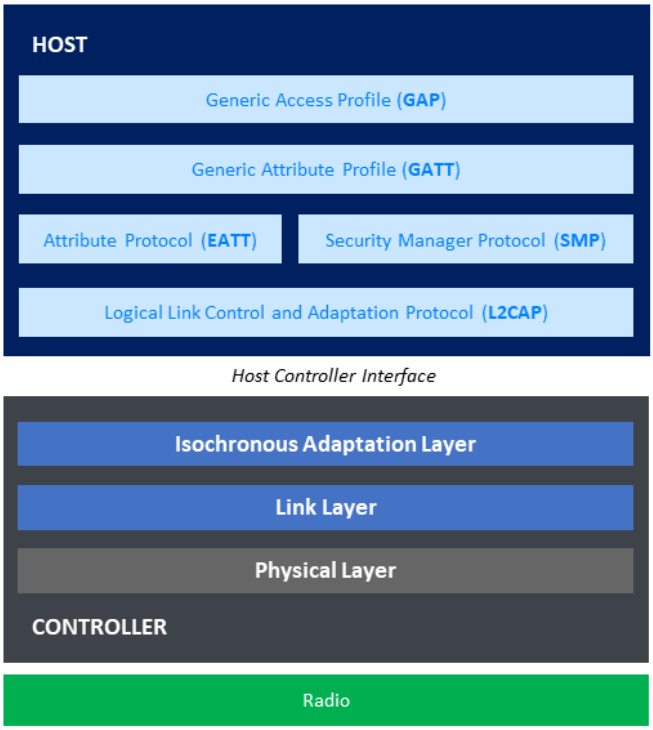


Figure 1 - The Bluetooth Low Energy Stack

Linux computers typically use a component known as *BlueZ* which the associated web site describes as the *official Linux Bluetooth Protocol Stack*.

Figure 2 depicts the architecture of Bluetooth on Linux when using BlueZ. As we can see, BlueZ implements the host layers of the Bluetooth LE stack and the controller typically resides within a chip which is either an integral part of the computer, as is the case with devices like the Raspberry Pi or is implemented within a peripheral device like a USB Bluetooth dongle. In the BlueZ documentation and code, the Bluetooth controller is referred to as an *adapter*.

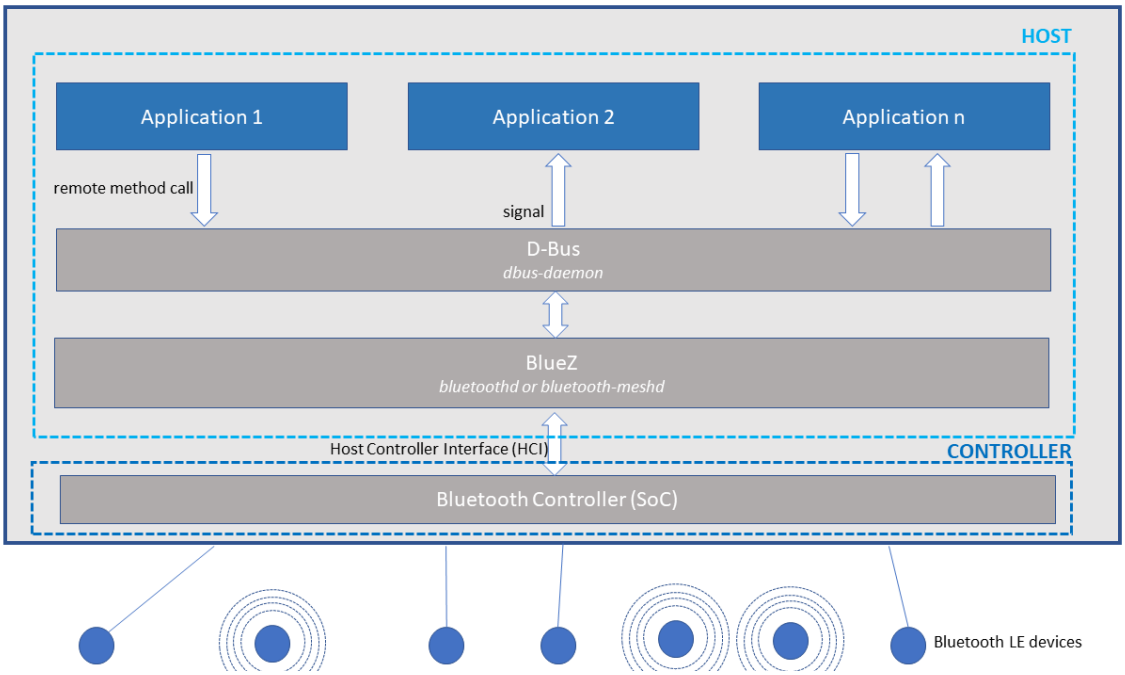


Figure 2 - Architecture

Communication between BlueZ in the host and the lower layers of the Bluetooth stack in the controller takes place via a standard logical interface which is called the Host Controller Interface or HCI for short. HCI is defined in the Bluetooth Core Specification. Underpinning HCI and allowing HCI commands to be passed from the host to the controller and HCI events to be passed from the controller to the host is one of the standard HCI transports, also defined in the core specification. Choices of HCI transport are UART, USB, Secure Digital (SD) and three-wire UART.

Applications either implement profiles and/or services based on GAP and GATT or they implement Bluetooth mesh models and act as a node in a Bluetooth mesh network.

Depending on whether a Linux computer is to host GAP/GATT applications or Bluetooth mesh nodes, one of two BlueZ daemon processes must be running, either *bluetoothd* for GAP/GATT or *bluetooth-meshd* for Bluetooth mesh. Note that a single computer cannot be used for both GAP/GATT and for Bluetooth mesh nodes at the same time. The selected BlueZ daemon serialises and handles all HCI communication on behalf of host applications.

That leaves one more component of the architecture as shown in Figure 2. That component is the dbus-daemon, a key part of an interprocess communication (IPC) system on Linux called D-Bus. D-Bus was created originally as a standard message-based communication system to facilitate interoperability between components in X-Windows desktop environments on personal computers. But D-Bus has wider applicability than communication involving desktop GUI components and since version 5.52 of BlueZ it has been the standard interface between Bluetooth applications running on Linux and BlueZ itself.

Understanding how to use D-Bus from application code is key to understanding how to use Bluetooth within applications that run on Linux computers.

4. BlueZ and D-Bus

BlueZ provides an API which is documented in a series of text files that are part of the BlueZ distribution. You can see them in the BlueZ source code repository here:

https://git.kernel.org/pub/scm/bluetooth/bluez.git/tree/doc

Applications do not make direct calls to BlueZ functions however and do not receive direct callbacks from BlueZ either. There is no need to compile application code against BlueZ header files. D-Bus decouples applications from BlueZ completely and consequently, Bluetooth application development work is predominantly concerned with using D-Bus APIs to send or receive messages.

4.1 D-Bus Concepts

The following D-Bus concepts are important to Bluetooth developers on Linux.

4.1.1 Message Buses

D-Bus inter-process communication centres around a message bus. Messages are placed on the bus by one process and travel along the bus to be delivered to one or more other processes connected to the same bus. There are two types of message bus. There is a single instance of the *system bus* and BlueZ uses this bus. In addition, one *session bus* exists per user login session within Linux.

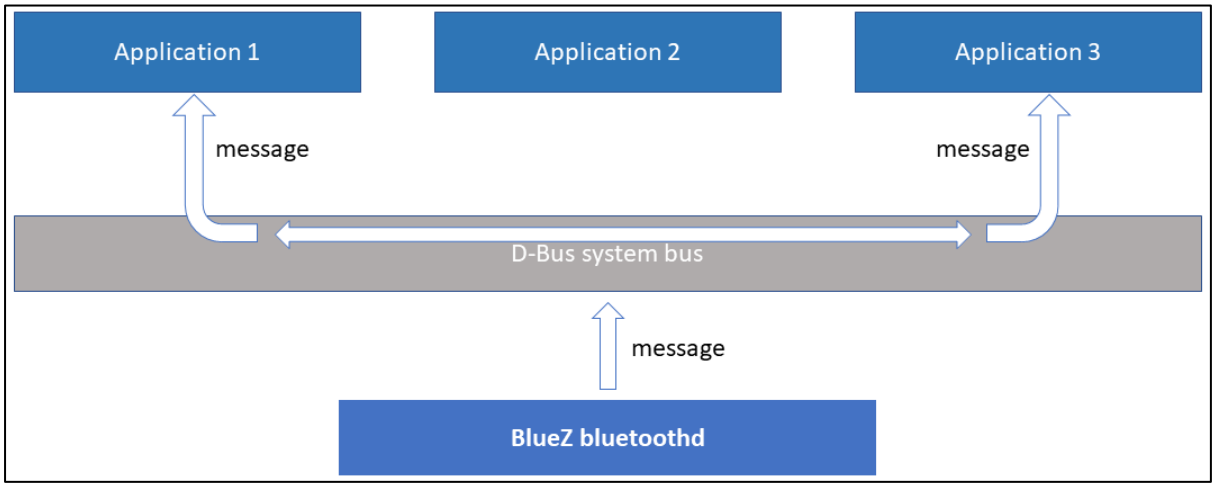


Figure 1 - messages delivered using the D-Bus system bus

4.1.2 Clients, Servers and Connections

D-Bus communication requires processes to connect to a message bus. A process which connects to a bus is called a client. A process which listens for and accepts connections is called a server. After a connection has been established though, the distinction between a client and a server ceases to exist.

When an application connects to a bus it is allocated a unique connection name which starts with a colon, for example *:1.16*.

Applications communicate with each other by sending and receiving various types of *D-Bus message* via the bus to which they are connected.

4.1.3 Objects, Interfaces, Methods, Signals and Properties

D-Bus uses an object-oriented paradigm for some of its concepts. Applications that use D-Bus are deemed to contain various ***objects***.

Objects implement ***interfaces*** which consist of a series of one or more functions or ***methods***. Interfaces have dot-separated names similar to domain names. For example, *org.freedesktop.DBus.Introspectable* and *org.bluez.GattManager1*.

An application can call a method of an object owned by another application by sending a special message over its connection to the bus. The message is passed along the bus and over a connection to the application which owns the target object.

Methods may or may not return a result. If a result is produced by a method, it is returned to the original, calling application as another type of D-Bus message via the bus.

Objects must be registered with the D-Bus daemon to allow their methods to be called by other applications. The act of registering an object with the bus is known as *exporting*. Each object has a unique identifier which takes the form of a ***path***. For example, an object that represents a Bluetooth device might have a path identifier of */org/bluez/hci0/dev_4C_D7_64_CD_22_0A*. The registration of an object with its path enables the D-Bus daemon to route messages addressed to the object's identifying path over the appropriate connection to the owning application.

Servers often expose objects and are sometimes then referred to as D-Bus **services**.

Note that paths have a hierarchical structure with parts earlier in the path containing or owning those towards the end of the path. In our example path (***/org/bluez/hci0/****dev_4C_D7_64_CD_22_0A)*, the device *dev_4C_D7_64_CD_22_0A* is owned by the Bluetooth adapter object which is known to the D-Bus daemon by the path identifier ***/org/bluez/hci0/***.

An object or more precisely, an interface of an object may emit ***signals***. A signal is a message that an object can send unprompted and can be likened to an *event*. Applications may subscribe to or register an interest in receiving particular signals. More than one application can register for a given signal and a copy of the signal will be delivered to each registered application (as shown in Figure 1). Some signals are delivered to all applications connected to the bus.

An object can have ***properties***. A property is an attribute whose value can be retrieved using a *get* operation or changed using a *set* operation. Properties are referenced by name and are accessible via an interface that the object implements.

4.1.3 Proxy Objects

Some D-Bus APIs support the concept of *proxy objects*. A proxy object represents a remote object in another application. But a proxy object is instantiated in the local process and its methods, which look the same as those of the remote object it represents, can be called directly by the application code. The proxy object then takes care of turning these local method calls into the sending and receiving of D-Bus messages. Proxy objects make D-Bus programming easier and relieve the application developer of the burden of having to work directly with D-Bus messages which can require some fairly low level programming.

4.1.4 Well-known Names

Applications can register a name by which they can be addressed instead of using a system allocated connection name (e.g. *:1.16*). Any application can do this but the practice is more common amongst system services of which BlueZ is an example. The Bluetooth daemon *bluetoothd* is a D-Bus server which owns the well known name ***org.bluez*** whilst the *bluetooth-meshd* daemon owns the name ***org.bluez.mesh***.

4.1.5 Standard Interfaces

A number of standard interfaces exist and are often used when working with BlueZ. For example:

org.freedesktop.DBus.ObjectManager

This interface defines the signals *InterfacesAdded* and *InterfacesRemoved*. InterfacesAdded is emitted when BlueZ discovers a new device. InterfacesRemoved is emitted when the device is no longer known to BlueZ.

It also defines the method *GetManagedObjects*. This allows an application to discover all of the objects that a D-Bus connected process possesses.

org.freedesktop.DBus.Properties

This interface defines methods which allow property values to be retrieved or set and a signal, *PropertiesChanged* which is emitted when a property of an object changes. For example, BlueZ device objects (formally these are D-Bus objects that implement the org.bluez.Device1 interface) implement the Properties interface and emit PropertiesChanged signals when properties such as the signal strength (RSSI) changes.

4.1.6 Data Types

D-Bus is not bound to any particular programming language and given different programming languages and platform architectures have varying approaches to and definitions of data types has its own language-agnostic data typing and format specifier system. This supports all the usual data type concepts including numeric types and strings and container types such as arrays and dictionaries.

Data types are explicitly indicated in a *type signature* which appears in the header fields of messages exchanged by D-Bus using its system of format specifiers. For example, the type specifier *a(ii)* means the message contains an array (a) of structs, each containing a 32-bit integer (i) and another 32-bit integer (i). Programming language D-Bus *bindings* (implementations of D-Bus which provide APIs) often handle type conversions automatically but sometimes it's necessary to deal with them directly from application code.

Note that an important and commonly used type within DBus communication is that of the *variant*. The variant type acts as a generic type wrapper around another concrete type which is ultimately only determined at runtime.

4.1.7 Introspection

D-Bus supports *introspection* which is a technique that allows the dynamic disclosure of the objects supported, their methods, signals and properties. An object may be described using XML and this

XML description provided to a process which requests it using the standard org.freedesktop.DBus.Introspectable.Introspect method. Figure 2 contains a simple example of introspection XML in an application written in C. It exposes details of an interface called com.bluetooth.mwoolley.DbusMultiplier which has a single method named *Multiply*. This method takes two 32-bit integer arguments and returns a 32-bit integer value in associated response messages.

```
    const char *introspection_data =
        " <!DOCTYPE node PUBLIC \"-//freedesktop//DTD D-BUS Object Introspection 1.0//EN\"
"
        "\"http://www.freedesktop.org/standards/dbus/1.0/introspect.dtd\">"
        " <!-- dbus-sharp 0.8.1 -->"
        " <node>"
        "   <interface name=\"org.freedesktop.DBus.Introspectable\">"
        "     <method name=\"Introspect\">"
        "       <arg name=\"data\" direction=\"out\" type=\"s\" />"
        "     </method>"
        "   </interface>"
        "   <interface name=\"com.bluetooth.mwoolley.DBusMultiplier\">"
        "     <method name=\"Multiply\">"
        "       <arg name=\"a\" direction=\"in\" type=\"i\" />"
        "       <arg name=\"b\" direction=\"in\" type=\"i\" />"
        "       <arg name=\"ret\" direction=\"out\" type=\"i\" />"
        "     </method>"
        "   </interface>"
        " </node>";
```

Figure 2 - Example introspection XML

4.1.8 Security Policies

D-Bus includes a security policy framework that requires communication between services and their objects and methods to be explicitly allowed or denied in a configuration file which is used by the dbus-daemon.

4.2 D-Bus Tools

Key tools for the D-Bus developer include *d-feet* and *dbus-monitor*.

4.2.1 D-feet

D-feet is a GUI application which allows applications connected to the system or session bus to be viewed and the objects and interfaces they export to be browsed. It also allows methods to be executed which is useful for testing purposes. Figure 3 shows a screenshot of d-feet in use on a Raspberry Pi and accessed using the remote desktop tool, VNC. The org.bluez service has been selected in the left-hand pane and this caused the objects the service contains, each identified by a hierarchical path, to be listed. Objects can be expanded to show their supported interfaces and within interfaces we can see their methods.

d-feet is an excellent tool for testing and for exploring and developing an understanding of D-bus.

Install d-feet on a Raspberry Pi by running

```
sudo apt-get install d-feet
```

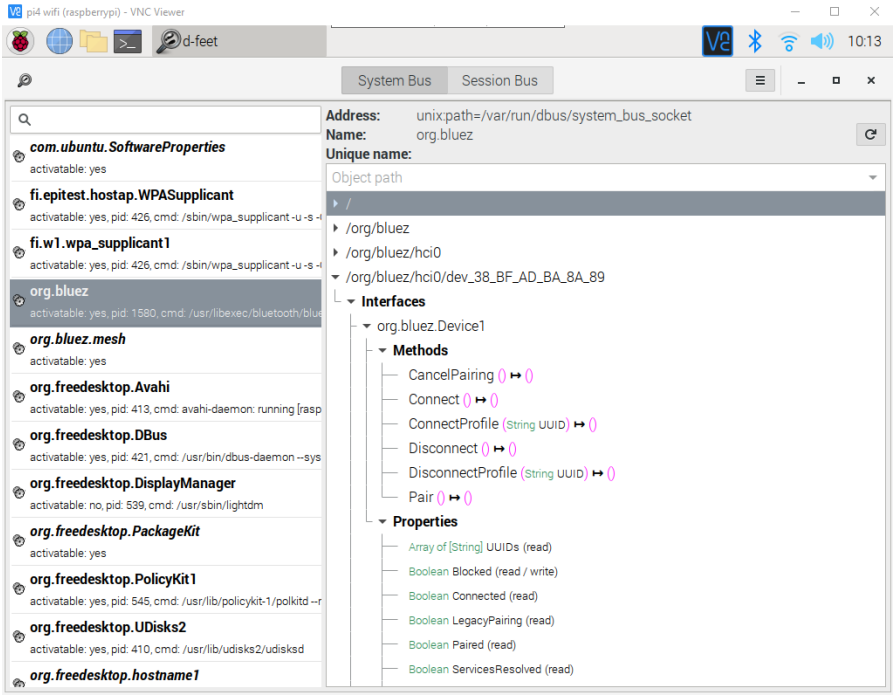


Figure 3 - d-feet used via VNC on a Raspberry Pi

4.2.2 dbus-monitor

dbus-monitor is a command line tool that allows messages exchanged with the dbus daemon to be viewed in real time. When running dbus-monitor, either the system bus or session bus must be selected. Some system messages will only be visible if running as root and if *eavesdropping* has been enabled.

To monitor the system bus:

```
sudo dbus-monitor --system
```

To monitor the session bus:

```
sudo dbus-monitor --session
```

NB: requires an X11 display so run from the desktop.

Eavesdropping is enabled by temporarily adding a security policy for the root user in a file which you will need to create, /etc/dbus-1/system-local.conf like this:

```
pi@raspberrypi:~ $ sudo cat /etc/dbus-1/system-local.conf
<!DOCTYPE busconfig PUBLIC
"-//freedesktop//DTD D-Bus Bus Configuration 1.0//EN"
"http://www.freedesktop.org/standards/dbus/1.0/busconfig.dtd">
<busconfig>
    <policy user="root">
        <allow eavesdrop="true"/>
        <allow eavesdrop="true" send_destination="*"/>
    </policy>
</busconfig>
```

See https://wiki.ubuntu.com/DebuggingDBus for the origin of this configuration.

To activate the change, a system reboot is required.

4.2.3 dbus-send

dbus-send is a command line tool which allows the manual injection of messages onto a DBus bus. This is very useful for testing where for example, we could simulate a signal being emitted by a service and check that it is delivered to the expected connected applications and handled properly.

Example:

```
dbus-send --system --type=signal / com.studyguide.greeting_signal string:"hello universe!"
```

4.3 Examples

4.3.1 A D-Bus Method Call Message

```
method call time=1634811621.566895 sender=:1.52 -> destination=:1.16
serial=10 path=/org/bluez/hci0/dev_EB_EE_7B_08_EC_3D;
interface=org.bluez.Device1; member=Connect
```

This D-Bus message was sent by an application with D-Bus connection ID :1.52 to an application with connection :1.16. The destination application has a Bluetooth device object with path identifier equal to /org/bluez/hci0/dev_EB_EE_7B_08_EC_3D. The object implements the org.bluez.Device1 interface which includes the method to be executed, Connect. We can see the device represented as a D-Bus object in d-feet in Figure 4.

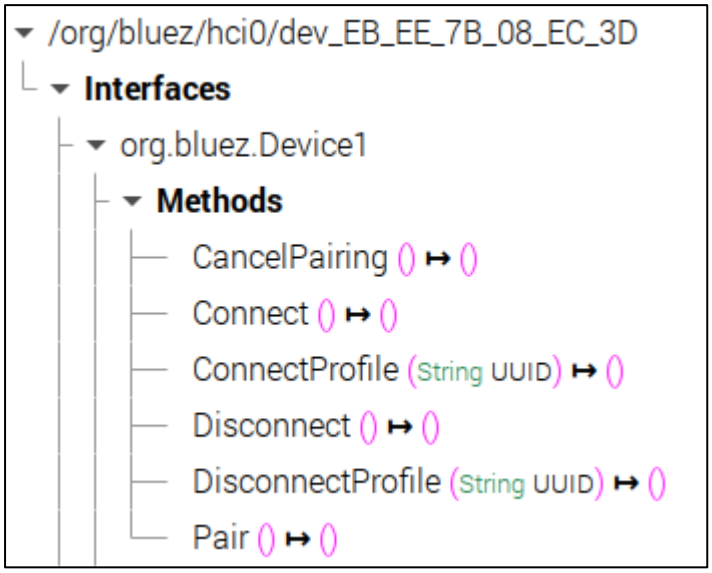


Figure 4 - a device object in d-feet

4.3.2 A D-Bus Signal Message

```
signal time=1634895336.839277 sender=:1.33 -> destination=(null destination)
serial=349 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 [
               12
            ]
      )
   ]
   array [
   ]
```

This D-Bus message is a signal which was sent from the BlueZ daemon and delivered to all registered applications (indicated by the null destination value in the header). The signal originates from a characteristic object with path identifier /org/bluez/hci0/dev_EE_CB_92_97_DB_18/service0025/char0026 and contains a notification value in the form of an array of bytes. Only one object can emit a signal whilst zero or more objects can consume it.

In this particular case, this is a notification from the Temperature characteristic which is owned by the Temperature service of a BBC micro:bit (see https://lancaster-university.github.io/microbit-docs/resources/bluetooth/bluetooth_profile.html) . The value 12 is a hexadecimal value representing the temperature 18 degrees Celsius.

5. Programming Languages and APIs

You can develop software for Linux using any number of programming language and will generally decide which to use based on your skills and preferences. Occasionally the problem domain you're working in will suit the use of a specialist programming language for that domain rather than some other, general purpose language.

Developing software which uses Bluetooth on Linux does not require any special programming language features. It does require a library that provides an API and there may be other requirements not relating directly to Bluetooth that lead you to select one language rather than another (e.g. user interface requirements).

There are numerous *language bindings* for D-Bus and many are listed here:

https://www.freedesktop.org/wiki/Software/DBusBindings/

Not all language bindings are equal. Some are regarded as offering *low-level APIs* and others *high-level APIs* (these are not well defined terms). Some are well documented. Some are not. You will need to do your own research to determine which meets your needs the best.

The relationship between language and D-Bus API is not a one to one relationship. Some languages have more than one API available to them. For example, C programmers can use GIO or the Embedded Linux Library (ELL). ELL seems well thought of but has no API documentation, requiring the developer to learn how to use it by looking at a few examples. GIO is documented but the documentation is of mixed quality and there's definitely a learning curve to getting started with it. Python developers have at least a couple of choices. dbus-python is used by the BlueZ project itself for testing tools which are included in the BlueZ distribution. Its documentation describes itself as

the reference implementation of the D-Bus protocol. However, the official list of D-Bus bindings (see link above) lists dbus-python as an *obsolete library* and instead recommends the use of pydbus. That said, dbus-python is definitely an active project, with a release having made quite recently in 2021 and it is the binding that BlueZ itself uses which means there's a source of useful examples in the BlueZ distribution itself.

As you are hopefully beginning to see, knowing where to start with programming languages and D-Bus bindings is a bit of a minefield.

The priority of this study guide is to help the reader get to grips with the generally applicable principles of developing software that uses BlueZ on Linux rather than on the foibles of one specific language and D-Bus binding. That said, in creating examples and exercises, a preference for well documented and supported bindings has been important in choosing their basis.

Python offers a relatively high-level API and this makes it easier to learn from. Consequently much of what follows in this study guide is based on Python. Despite the fact that there is a view that dbus-python is obsolete (perhaps deprecated would be a better description), it is used in this study guide since it is used by the BlueZ developers themselves. When this changes, it is expected that the code in this study guide will be migrated to be based on whatever BlueZ itself adopts.

It is hoped that code examples will make sense to all developers regardless of their background. For those readers with no prior experience of Python, completing a Python tutorial first may be advisable before attempting the exercises in other modules.

Bluetooth for Linux Developers Study Guide

Mastering D-Bus Basics using Python

Release : 1.0.1

Document Version: 1.0.0

Last updated : 16th November 2021

Contents

1. REVISION HISTORY 3

2. INTRODUCTION 4

3. HELLO UNIVERSE 4

3. CALLING METHODS 5

3.1 Exploring with DBus tools 5

3.2 Implementing hostname retrieval in Python 9

4. RECEIVING SIGNALS 12

5. RECEIVING METHOD CALLS 14

6. EMITTING SIGNALS 20

7. SUMMARY 22

1. Revision History

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

2. Introduction

In this module you'll learn how to do basic DBus programming using Python. Use cases will not involve Bluetooth, to keep things as simple as possible to begin with.

3. Hello Universe

In a feeble attempt to avoid the ubiquitous Hello World example, we'll be more inclusive and broaden the scope of our code generated greeting to apply to the whole universe. We're nice and friendly like that.

To validate your Python environment before we move on to all things DBus, add the following code to a file:

```
#!/usr/bin/python3
print("Hello Universe!")
```

Make sure the file is executable and then run it. Check your results match those which are shown.

```
pi@raspberrypi:~/projects/ldsg/solutions/python $ python3 hello_universe.py
Hello Universe!
```

Note that the first line of the code is the Linux *shebang* which tells the parent interpreter which interpreter to use to execute the remainder of the script. In our case, it's the Python 3 interpreter at /usr/bin/python3. This means you can either explicitly run the interpreter (python3) with the .py file as an argument, per the example (`python3 hello_universe.py`) or you can just execute the .py file directly (`./hello_universe.py`). If you're editing source code on a Windows PC and then transferring the code to your Linux computer for testing, watch out for end-of-line issues. Windows uses different characters to Linux to mark the end of a line in a text file. If you get this error:

```
pi@raspberrypi:~/projects/ldsg/solutions/python $ ./hello_universe.py
-bash: ./hello_universe.py: /usr/bin/python3^M: bad interpreter: No such file or directory
pi@raspberrypi:~/projects/ldsg/solutions/python $
```

It means you have Windows carriage return/line feed characters in your source code. Change your Windows text editor settings to use Unix end of line characters and fix the file in question with the dos2unix tool, which you may need to install with `sudo apt-get install dos2unix`.

```
pi@raspberrypi:~/projects/ldsg/solutions/python $ ./hello_universe.py
-bash: ./hello_universe.py: /usr/bin/python3^M: bad interpreter: No such file or directory
pi@raspberrypi:~/projects/ldsg/solutions/python $ dos2unix hello_universe.py
dos2unix: converting file hello_universe.py to Unix format...
pi@raspberrypi:~/projects/ldsg/solutions/python $ ./hello_universe.py
Hello Universe!
```

When you have proven that you have a working Python development environment, proceed.

3. Calling Methods

3.1 Exploring with DBus tools

Some of the key DBus concepts which were first introduced in Module 03 will now be illustrated. You will start by using the GUI tool D-Feet and then implement a simple example in Python, one step at a time.

Run D-Feet and with the System Bus window selected, find the service with well known name *org.freedesktop.hostname1*.

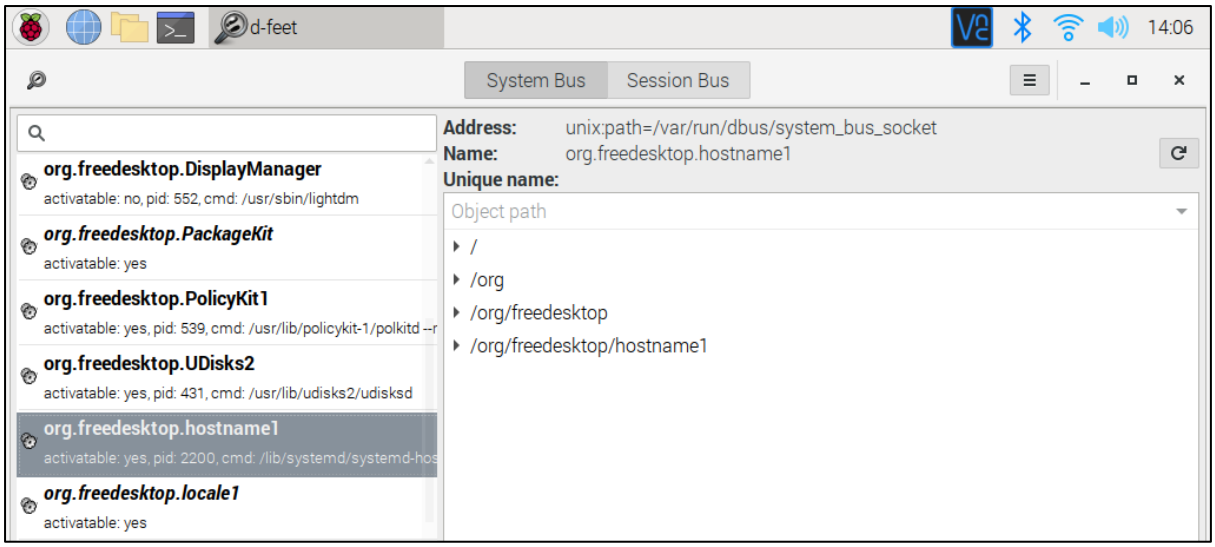


Figure 1 - org.freedesktop.hostname1

Explore the objects owned by this service in the right hand pane. Pay particular attention to the object with identifying path /org/freedesktop/hostname1.

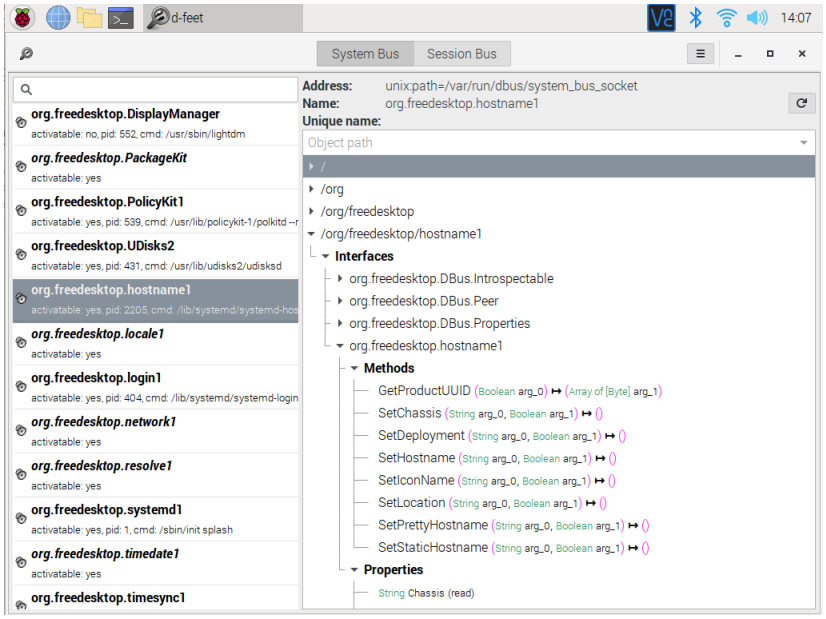


Figure 2 - org/freedesktop/hostname1 - Methods

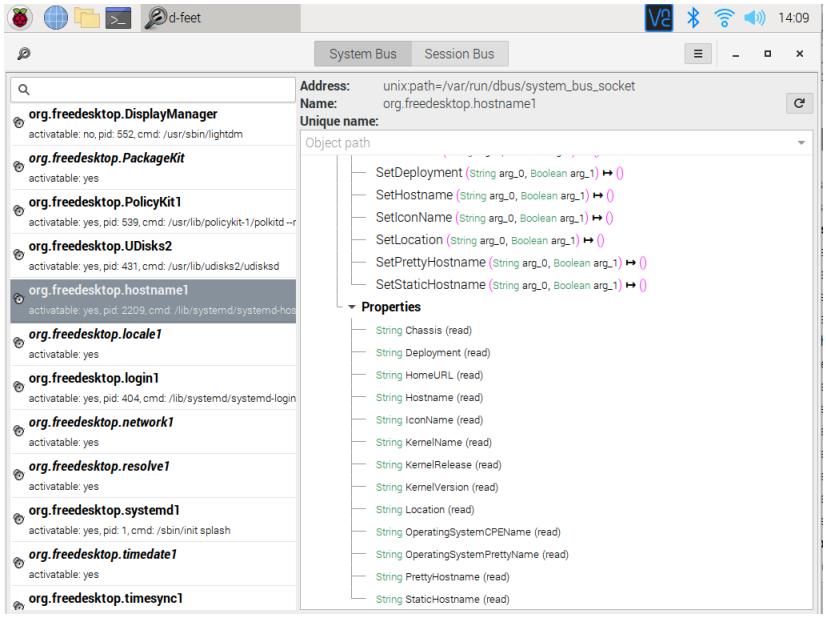


Figure 3 - org/freedesktop/hostname1 - Properties

This is a system service which provides access to the hostname and other, related machine metadata via Dbus. It is exposed by the system daemon systemd-hostname and you can read all about it here: https://www.freedesktop.org/software/systemd/man/org.freedesktop.hostname1.html

Double click on the Hostname property. This should cause D-Feet to request the value of the property which is

- named “Hostname”
- is a member of the object called org/freedesktop/hostname1
- and the interface which this object implements, org.freedesktop.hostname1

On a Raspberry Pi the result will be something like Figure 4:

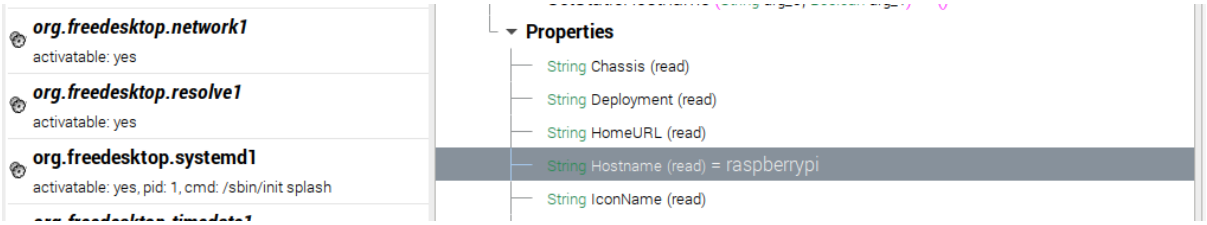


Figure 4 - D-Feet and the Hostname property

D_Feet is not acquiring the property name directly from the org.freedesktop.hostname1 interface. In fact it is performing the remote execution of a method called Get which belongs to the standard and commonly implemented org.freedesktop.DBus.Properties interface of the selected object.

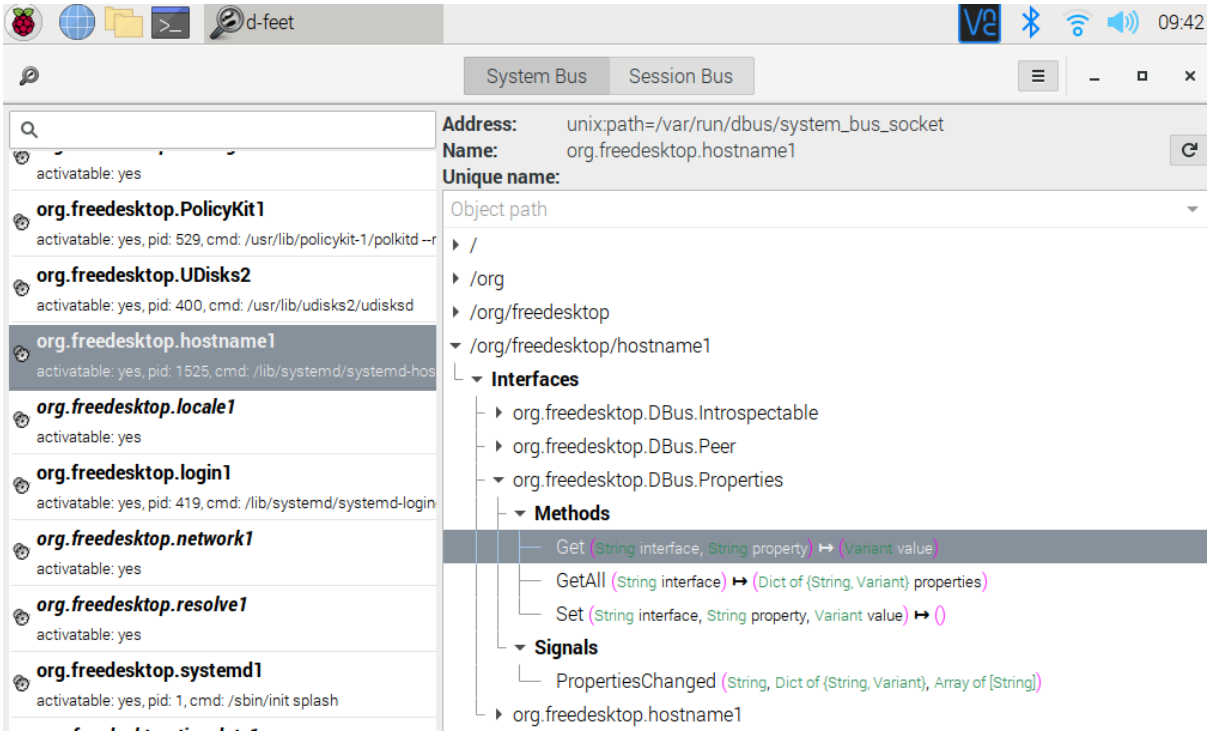


Figure 5 - The org.freedesktop.DBus.Properties interface

We can verify this and watch it happen in real-time using the command *sudo dbus-monitor --system*. In another terminal window, run this command and then double click on the Hostname property in D-Feet again. You'll see quite a few lines output in the dbus-monitor window but if you look carefully you will find the method Get being called and its response. Don't forget, what you're actually seeing is a message sent by the D-Feet application to another process with the DBus system service sitting in the middle and acting as a broker. The text output by dbus-monitor is a faithful rendering of those messages.

```
method call time=1635167872.257591 sender=:1.47 -> destination=:1.94 serial=458
path=/org/freedesktop/hostname1; interface=org.freedesktop.DBus.Properties; member=Get
   string "org.freedesktop.hostname1"
   string "Hostname"

method return time=1635167872.257850 sender=:1.94 -> destination=:1.47 serial=10
reply_serial=458
   variant       string "raspberrypi"
```

Figure 6 - Properties Get method call message and return value

You can see that the DBus message contains various attributes.

- The path to the target object (/org/freedesktop/hostname1) is specified in the *path* attribute.
- The method to execute and the interface of the target object that it belongs to is specified in the *interface* and *member* attributes.
- The Get method requires two arguments and the type and value of the two arguments follow the member attribute. Both are of type *string*. The first names the interface that owns the property whose value we wish to be read and the second is the string name of that property.

We then see the response message which contains a single variant which wraps a string value of "raspberrypi".

What else can we learn about DBus from dbus-monitor? Apart from some obvious timestamps, the other interesting attributes shown here are the sender and destination which have values :1.47 and :1.94 respectively. In module 03 we learned that processes connected to a DBus bus are identified with a unique value of the form :n.nn and that's what the values :1.47 and :1.94 are. Specifically though, :1.94 identifies the service which has the well known name org.freedesktop.hostname1. This is not surprising, since we're trying to read the value of a property which belongs to one of the objects that D-Feet tells us is owned by this service. But how does the system know that the connection identified by :1.94 belongs to the org.freedesktop.hostname1 server?

Look a little further back in your dbus-monitor output and you'll find a message pair like this:

```
method call time=1635167872.253307 sender=:1.47 -> destination=org.freedesktop.DBus
serial=456 path=/org/freedesktop/DBus; interface=org.freedesktop.DBus; member=GetNameOwner
   string "org.freedesktop.hostname1"
method return time=1635167872.253402 sender=org.freedesktop.DBus -> destination=:1.47
serial=441 reply_serial=456
   string ":1.94"
```

This is another method call but this time, we're seeing a kind of name resolution operation taking place. There's a service called org.freedesktop.DBus and it has a method, *GetNameOwner* which belongs to the org.freedesktop.DBus interface. The method is being passed an argument with value org.freedesktop.hostname1 and you can see that the response is the numeric connection identifier of :1.94.

Hopefully you're beginning to develop some insight now into how the DBus system works.

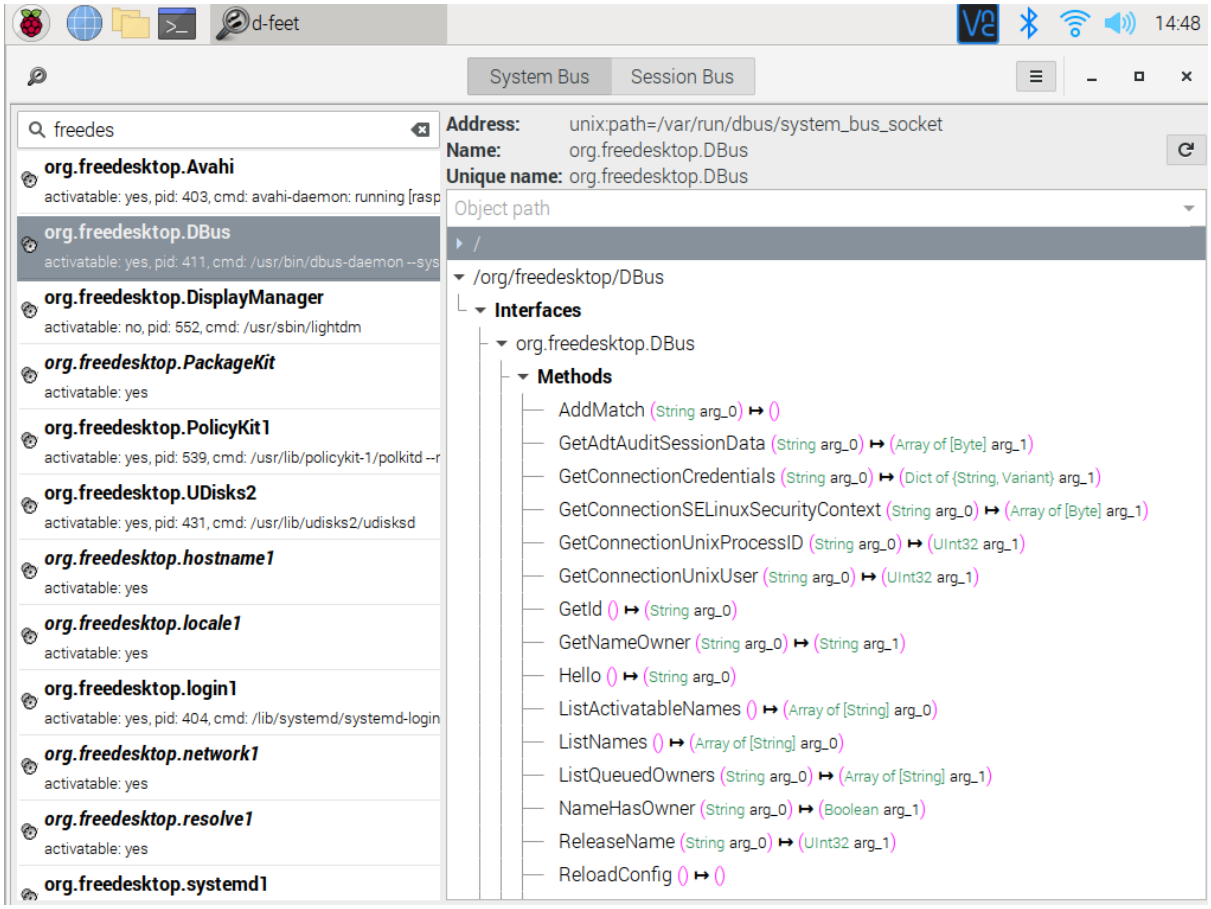


Figure 7 - org.freedesktop.DBus supports the DBus system itself

3.2 Implementing hostname retrieval in Python

Your next task is to mimic what we did in 3.1 with D-Feet and implement a simple Python script which retrieves the Hostname property from the org.freedesktop.hostname1 DBus service and displays it.

Create a source file with the following in it to start things off:

```
#!/usr/bin/python3
import dbus
```

The *import dbus* statement provides access to the dbus-python module and its API.

By implementing the following four steps, your script will retrieve and display Hostname.

1. Connect to the DBus system bus
2. Create a proxy to the /org/freedesktop/hostname1 object owned by the org.freedesktop.hostname1 service so that we can easily invoke its methods with local calls instead of by formulating and sending the low level DBus messages that are required.
3. Obtain a reference to the org.freedesktop.DBus.Properties interface because it contains the method we want to call.

4. Call the interface's Get method with suitable arguments and synchronously receive the result.

Update your code to cover these steps so that it looks like this:

```
#!/usr/bin/python3
#
https://www.freedesktop.org/software/systemd/man/org.freedesktop.hostname1.html

import dbus

bus = dbus.SystemBus()
proxy = bus.get_object('org.freedesktop.hostname1','/org/freedesktop/hostname1')
interface = dbus.Interface(proxy, 'org.freedesktop.DBus.Properties')

print("----------------")
hostname = interface.Get('org.freedesktop.hostname1','Hostname')
print("The host name is ",hostname)
```

Review and check that you are happy with how the code corresponds to the four steps that were listed. Run the code and check that the output looks something like this:

```
pi@raspberrypi:~/projects/ldsg/solutions/python $ python3 ./hostname.py
The host name is raspberrypi
```

You may have noticed that the org.freedesktop.DBus.Properties interface also has a GetAll method. Let's add a call to this method to the script just out of curiosity.

Update your code so that it looks like the following and then run it.

```
#!/usr/bin/python3
#
https://www.freedesktop.org/software/systemd/man/org.freedesktop.hostname1.html

import dbus

bus = dbus.SystemBus()
proxy =
bus.get_object('org.freedesktop.hostname1','/org/freedesktop/hostname1')
interface = dbus.Interface(proxy, 'org.freedesktop.DBus.Properties')

all_props = interface.GetAll('org.freedesktop.hostname1')
print(all_props)

print("----------------")
hostname = interface.Get('org.freedesktop.hostname1','Hostname')
print("The host name is ",hostname)
```

```
pi@raspberrypi:~/projects/ldsg/solutions/python $ python3 ./hostname.py
dbus.Dictionary({dbus.String('Hostname'): dbus.String('raspberrypi', variant_level=1),
dbus.String('StaticHostname'): dbus.String('raspberrypi', variant_level=1),
dbus.String('PrettyHostname'): dbus.String('', variant_level=1), dbus.String('IconName'):
dbus.String('computer', variant_level=1), dbus.String('Chassis'): dbus.String('',
variant_level=1), dbus.String('Deployment'): dbus.String('', variant_level=1),
dbus.String('Location'): dbus.String('', variant_level=1), dbus.String('KernelName'):
dbus.String('Linux', variant_level=1), dbus.String('KernelRelease'): dbus.String('5.4.79-
v7l+', variant_level=1), dbus.String('KernelVersion'): dbus.String('#1373 SMP Mon Nov 23
13:27:40 GMT 2020', variant_level=1), dbus.String('OperatingSystemPrettyName'):
dbus.String('Raspbian GNU/Linux 10 (buster)', variant_level=1),
dbus.String('OperatingSystemCPEName'): dbus.String('', variant_level=1),
dbus.String('HomeURL'): dbus.String('http://www.raspbian.org/', variant_level=1)},
signature=dbus.Signature('sv'))
----------------
The host name is  raspberrypi
```

The GetAll method takes one argument only, the name of the interface that owns the properties to be retrieved. As you can see from the result, it returns a dictionary of all supported properties as a series of dictionary entries each of which has a string type key and a variant type value part. You can see these details in D-Feet.

4. Receiving Signals

Signals are messages which an application might receive asynchronously. To receive particular signals of interest, an application must register its interest in the signal with the bus it is connected to and provide a callback function to process signals and their arguments (if any).

To asynchronously receive anything in a DBus application requires the use of an event loop. When an event loop is started, it will cause the application to block during which time it can receive callbacks to its signal handler functions.

Event loop implementations are available from various libraries and generally all function in the same way. Glib provides such an event loop and it is suitable for use within DBus applications and shall be used in our exercises and examples. See https://docs.gtk.org/glib/main-loop.html for more information.

Your next task is to write a Python script which registers for a signal which delivers a string value as an argument and prints that argument whenever the expected signal is received. To generate the signal, you'll use a command line tool called *dbus-send*. This relieves us of the need to write another DBus application which generates signals, purely for testing purposes.

Create a file called signal_test.py and add the following content.

```
import dbus
import dbus.mainloop.glib
from gi.repository import GLib

mainloop = None


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

All we've done here is to make a start by importing the dbus library as usual, importing an implementation of GLib whose event loop we intend to use and specified that it should be used for all DBus connections. See https://dbus.freedesktop.org/doc/dbus-python/dbus.mainloop.html#module-dbus.mainloop.glib

It's possible you will need to install GLib before this code will work. Run it and if you get this result....

```
pi@raspberrypi:~/projects/ldsg/solutions/python $ python3 signal_test.py
Traceback (most recent call last):
  File "wip.py", line 3, in <module>
    from gi.repository import GLib
ModuleNotFoundError: No module named 'gi'
```

... then you need to install GLib which you can do like this:

```
pi@raspberrypi:~/projects/ldsg/solutions/python $ sudo apt-get install python3-gi
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  gir1.2-glib-2.0
The following NEW packages will be installed:
```

```
  gir1.2-glib-2.0 python3-gi
0 upgraded, 2 newly installed, 0 to remove and 228 not upgraded.
Need to get 0 B/304 kB of archives.
After this operation, 1,546 kB of additional disk space will be used.
Do you want to continue? [Y/n] y
Selecting previously unselected package gir1.2-glib-2.0:armhf.
(Reading database ... 96669 files and directories currently installed.)
Preparing to unpack .../gir1.2-glib-2.0_1.58.3-2_armhf.deb ...
Unpacking gir1.2-glib-2.0:armhf (1.58.3-2) ...
Selecting previously unselected package python3-gi.
Preparing to unpack .../python3-gi_3.30.4-1_armhf.deb ...
Unpacking python3-gi (3.30.4-1) ...
Setting up gir1.2-glib-2.0:armhf (1.58.3-2) ...
Setting up python3-gi (3.30.4-1) ...
```

When you have verified your GLib library is installed and working from your code, proceed. We need a function which can receive callbacks when a signal comes in. Update your code as shown:

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

mainloop = None

def greeting_signal_received(greeting):
    print(greeting)

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

Next, we'll add code which will

1. connect to the system bus
2. register to receive the signal
3. acquire and start a mainloop

To be able to do this, we need details of the signal to be received so we can specify these details when registering with the system bus. In our case, the signal is called GreetingSignal and it is emitted by the com.example.greeting interface of a remote object. We don't care which object this is or which service owns that object.

Update your code:

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

mainloop = None

def greeting_signal_received(greeting):
    print(greeting)
```

```
dbus.mainloop.glib.DBusGMainLoop(set_as_default=True)
bus = dbus.SystemBus()
bus.add_signal_receiver(greeting_signal_received,
        dbus_interface = "com.example.greeting",
        signal_name = "GreetingSignal")

mainloop = GLib.MainLoop()
mainloop.run()
```

The add_signal_receiver function registers this application's requirement that signals named GreetingSignal emitted by an interface called com.example.greeting are delivered by callback to the function greeting_signal_received. We know that this signal includes a string argument since that's part of its specification and so our callback function accommodated this.

Run your code in one window and in another, run the following command repeatedly with varying parameter values to test:

```
pi@raspberrypi:~ $ dbus-send --system --type=signal / com.example.greeting.GreetingSignal
string:"hello"
pi@raspberrypi:~ $ dbus-send --system --type=signal / com.example.greeting.GreetingSignal
string:"wotcha"
pi@raspberrypi:~ $ dbus-send --system --type=signal / com.example.greeting.GreetingSignal
string:"howdy"
```

The parameters to dbus-send include options which indicate the system bus is to be used and that the message is of type signal. The forward slash is the path (object identifier) of the object that owns the interface that emitted the signal. In our case we're using the *root path*. We could specify the object path when registering our signal handler so that we only process signals from that specific object. In our case we've kept things simple and only match on the interface and signal name.

In the window running your signal handler code you should see:

```
pi@raspberrypi:~/projects/ldsg/solutions/python $ python3 signal_test.py
hello
wotcha
howdy
```

For more information on dbus-send see https://dbus.freedesktop.org/doc/dbus-send.1.html

5. Receiving Method Calls

For an application to be able to receive method calls from other D-Bus services or emit its own signals it must register objects and any owned interfaces and their callable methods with a bus. As always, objects are identified with a path and interfaces with the dot notation you have already seen. Methods are registered with a name and the signature of any input arguments and return values. The bus knows little about such objects other than their identifier and which connected application owns the implementation of the object, as indicated by its well known name or system allocated connection name. This is how the bus knows how to route method call messages sent by other connected applications.

The act of registering an object, its interfaces and callable methods in this way is known as *exporting*.

Different languages and D-Bus libraries take various approaches to exporting objects at a code level. In Python, it's quite easy however requiring only a basic knowledge of Python's approach to object-orientation and some special code *decorations* which we'll come to.

Create a new source file called *calculator.py* and add the following code:

```
#!/usr/bin/python3
import dbus
import dbus.service
import dbus.mainloop.glib
from gi.repository import GLib

mainloop = None

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

mainloop = GLib.MainLoop()
print("waiting for some calculations to do....")
mainloop.run()
```

Most of this code should look familiar. We're using the Glib library once again because we need an event loop so that we can receive method calls in exactly the same way that we needed one to receive signals. This is a general rule in D-Bus programming.

Next, add the highlighted code to create a class which will implement our callable methods.

```
#!/usr/bin/python3
import dbus
import dbus.service
import dbus.mainloop.glib
from gi.repository import GLib

mainloop = None

class Calculator(dbus.service.Object):
    # constructor
    def __init__(self, bus):
        self.path = '/com/example/calculator'
        dbus.service.Object.__init__(self, bus, self.path)

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

calc = Calculator(bus)

mainloop = GLib.MainLoop()
```

```
print("waiting for some calculations to do....")
mainloop.run()
```

Points to note:

- class Calculator(dbus.service.Object) declares a class called Calculator which inherits from (i.e. is a subclass of) a Python dbus class called dbus.service.Object. Subclassing this class is all that is required for your object to automatically support D-Bus introspection.
- def __init__(self, bus) is the constructor method for the Calculator subclass. It requires the standard Python *self* argument which is set to an instance of this class at runtime when the class is instantiated and a D-Bus bus value. This means that when creating a Calculator object we must indicate which bus we want it to be exported to.
- We then create an instance variable called self.path with the value "/com/example/calculator". This will be the identifying path value known to the bus once we've exported the object.
- Next, we call the constructor method (*__init__*) of the superclass dbus.service.Object with the bus and path arguments. This exports the object and it should now be known to the specified bus.
- In the main body of the script, we attach to the GLib mainloop and then we create an instance of the Calculator class, passing an instance of the D-Bus system bus into the constructor. At this stage we should now have successfully exported the object. It will support basic introspection but not be able calculate anything yet. We'll address that slight limitation shortly.

Note that when exporting objects or receiving signals we must always attach to a main loop. We do this in this code with `dbus.mainloop.glib.DBusGMainLoop(set_as_default=True)`. Failure to attach to a mainloop will result in this runtime error:

```
RuntimeError: To make asynchronous calls, receive signals or export objects, D-Bus
connections must be attached to a main loop by passing mainloop=... to the constructor or
calling dbus.set_default_main_loop(...)
```

Run your code and use D-Feet to find it. Examine its structure which should look like this:

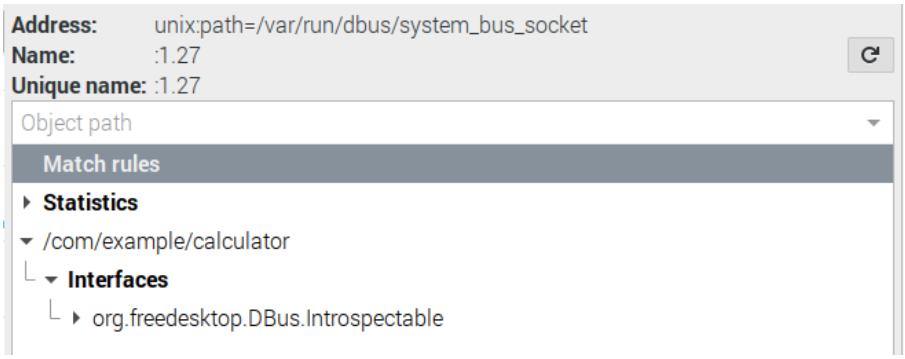


Figure 8 - Calculator object exported to the D-Bus system bus

Now let's give our D-Bus calculator service a useful capability. Without wishing to get over-ambitious with our mathematics, let's give it the ability to.... add together two integers and return the result.

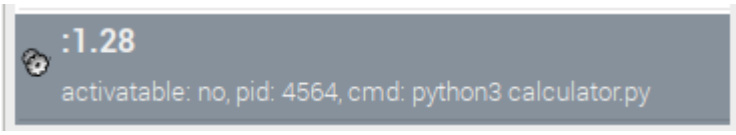
Update your code as shown:

```
#!/usr/bin/python3
import dbus
import dbus.service
import dbus.mainloop.glib
from gi.repository import GLib

mainloop = None

class Calculator(dbus.service.Object):
    # constructor
    def __init__(self, bus):
        self.path = '/com/example/calculator'
        dbus.service.Object.__init__(self, bus, self.path)

    @dbus.service.method("com.example.calculator_interface",
                         in_signature='ii',
                         out_signature='i')
    def Add(self, a1, a2):
        sum = a1 +a2
        print(a1," + ",a2," = ",sum)
        return sum

dbus.mainloop.glib.DBusGMainLoop(set_as_default=True)
bus = dbus.SystemBus()
calc = Calculator(bus)
mainloop = GLib.MainLoop()
print("waiting for some calculations to do....")
mainloop.run()
```

@dbus.service.method is a *decorator*. There are tutorials on the internet that will explain everything you might want to know and more about decorators. For our purposes, all we need to know is that if we want to export a method within an exported object, this is the syntax to use. Arguments to the decorator start with an interface name. It is this interface that the exported method will belong to. Then we have a type specifier that indicates the types of any input arguments and a final parameter that indicates the type(s) of results returned by the method. The D-Bus documentation lists the type specifier values that are defined and the syntax for combining them into tuples, arrays, dictionaries and so on. In our example the method accepts two 32-bit integer arguments as input and returns a single 32-bit result.

After the decorator, we have the actual method implementation which has the name *Add* and expected arguments per the decorator export declaration. And the implementation itself? Well, hopefully that needs no explanation.

Run your code. Use D-Feet to find the service with an executable named calculator.py:



Select it and explore the objects and interfaces it exports in the right hand pane.

The Interfaces section should contain the calculator_interface interface and it should contain the Add method. In short, it should look like this:

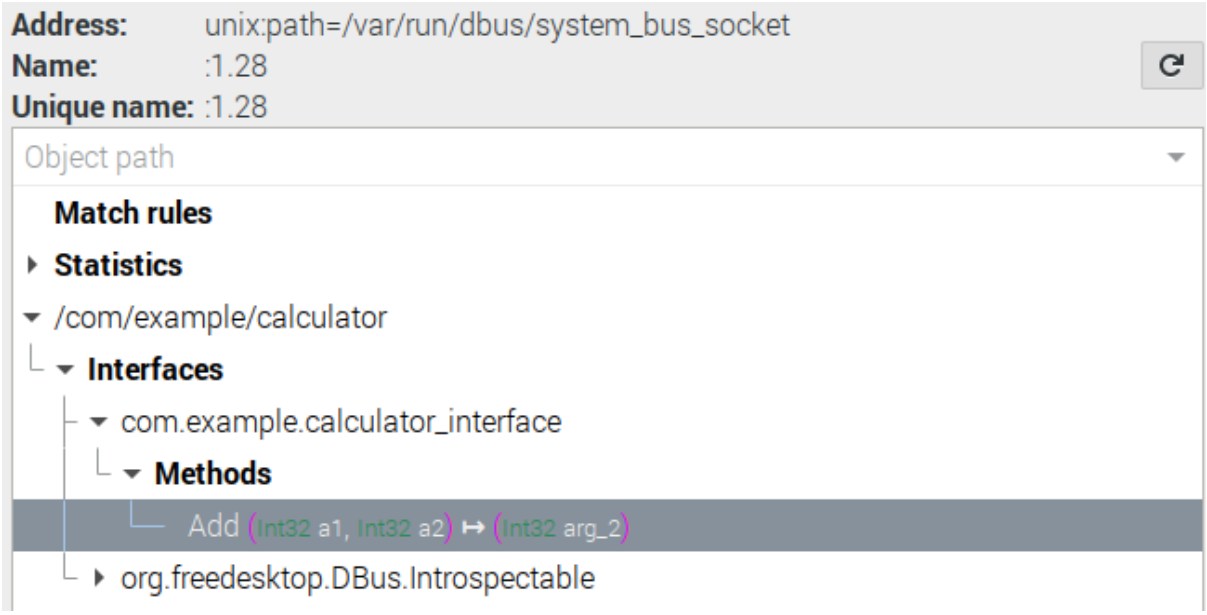


Figure 9 - Calculator object, interface and Add method

You can see that we also have the org.freedesktop.Dbus.Introspectable interface, as expected but with almost no code required from us.

Double click on the Add method in D-Feet and test your method. It's very likely you will get an **AccessDenied** error like this:

```
('g-dbus-error-quark: GDBus.Error:org.freedesktop.DBus.Error.AccessDenied: '
 'Rejected send message, 1 matched rules; type="method_call", sender=":1.11" '
 '(uid=1000 pid=4854 comm="/usr/bin/python3 /usr/bin/d-feet ") '
 'interface="com.example.calculator_interface" member="Add" error '
 'name="(unset)" requested_reply="0" destination=":1.13" (uid=1000 pid=4873 '
 'comm="python3 wip.py ") (9)')
```

This is because the D-Bus system includes a security policies feature which allows control to be exercised over which Linux users can send messages and whether or not they can monitor those messages using the eavesdropping capability that you should already have encountered if you followed the instructions in module A1 to set your environment up. A detailed review of security policies is outside the scope of this study guide but to fix the error and allow testing to proceed, perform the following actions:

Create the file /etc/dbus-1/system.d/com.example.calculator.conf with this content:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE busconfig PUBLIC
 "-//freedesktop//DTD D-BUS Bus Configuration 1.0//EN"
 "http://www.freedesktop.org/standards/dbus/1.0/busconfig.dtd">
<busconfig>
```

```
    <policy user="pi">
      <allow eavesdrop="true"/>
      <allow eavesdrop="true" send_destination="*"/>
      <allow own="com.example.calculator_interface"/>
      <allow eavesdrop="true" send_interface="com.example.calculator_interface"
send_member="Add"/>
    </policy>
</busconfig>
```

Note: this policy assumes you are performing your testing as a Linux user “pi”. Change this if required.

Restart the D-Bus daemon with:

```
sudo systemctl restart dbus
```

Your D-Feet instance will exit at this point. Restart it and your Python script and test again. It should work now.

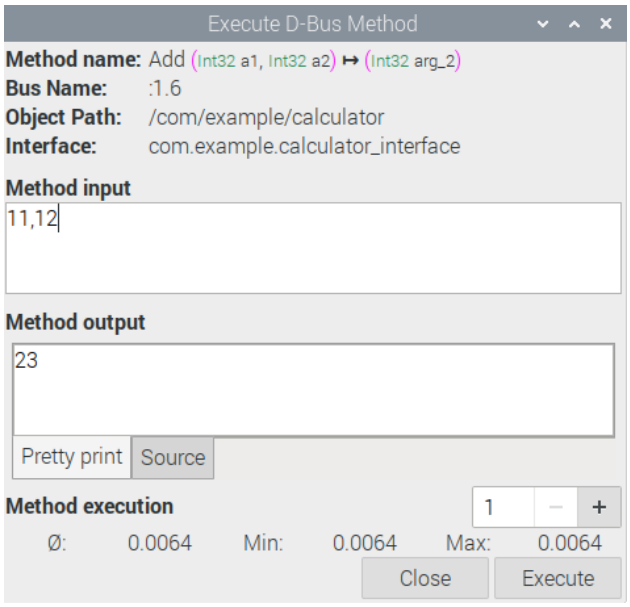


Figure 10 - Testing the Calculator

Output from your Python script during testing will look something like this.

```
pi@raspberrypi:~/projects/ldsg/solutions/python $ python3 calculator.py
waiting for some calculations to do....
12  +  11  =  23
563  +  99  =  662
1  +  1  =  2
1  +  -1  =  0
```

Have a go at adding further methods for operations such as multiplication, division and subtraction. Remember that you’ll need to use the right type specifiers if you intend to support anything other than integers.

6. Emitting Signals

You already know how to register for and receive signals emitted by other services but your own objects can also emit signals for other D-Bus services to receive. To do so, an object which owns the signal must be exported in the same way as for exposing a method which can be remotely called.

Let's create an application which increments an integer counter once a second and emits its value as a signal.

Create a file called counter_signal.py with this code in it:

```
#!/usr/bin/python3
import dbus
import dbus.service
import dbus.mainloop.glib
from gi.repository import GLib
import time

mainloop = None

class Counter(dbus.service.Object):

    def __init__(self, bus):
        self.path = '/com/example/counter'
        self.c = 0
        dbus.service.Object.__init__(self, bus, self.path)

    def increment(self):
        self.c = self.c + 1
        print(self.c)

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

while True:
    counter.increment()
    time.sleep(1)
```

And run it. You'll see output like this:

```
pi@raspberrypi:~/projects/ldsg/solutions/python $ python3 counter_signal.py
1
2
3
4
5
6
7
8
9
10
11
```

Exciting stuff. This code is doing everything we need it to except for one thing. It doesn't emit signals containing the counter, it just prints its value to the local console. So let's fix that now. Update your code like this:

```
#!/usr/bin/python3
import dbus
import dbus.service
import dbus.mainloop.glib
from gi.repository import GLib
import time

mainloop = None

class Counter(dbus.service.Object):

    def __init__(self, bus):
        self.path = '/com/example/counter'
        self.c = 0
        dbus.service.Object.__init__(self, bus, self.path)

    @dbus.service.signal('com.example.Counter')
    def CounterSignal(self, counter):
        # nothing else to do so...
        pass

    def emitCounterSignal(self):
        self.CounterSignal(self.c)

    def increment(self):
        self.c = self.c + 1
        print(self.c)

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

while True:
    counter.increment()
    counter.emitCounterSignal()
    time.sleep(1)
```

Look at the code we just added. Key points to note are:

- We declare and export a signal using the @dbus.service.signal decorator in a way which is reminiscent of the way we dealt with methods we wanted to be callable by other D-Bus services.

- The implementation of the signal is empty apart from a *pass* statement which in Python does nothing. This is included solely to make the function def syntactically valid.
- To actually emit a signal, we call the signal function, in our case indirectly via another function *emitCounterSignal*.
- We emit signals at each counter increment from within the while loop.

Run this code in one window and dbus-monitor --system in another. You should see signals being received on the system bus like this:

```
signal time=1635329225.879510 sender=:1.52 -> destination=(null destination) serial=2
path=/com/example/counter; interface=com.example.Counter; member=CounterSignal
   int32 1
signal time=1635329226.881303 sender=:1.52 -> destination=(null destination) serial=3
path=/com/example/counter; interface=com.example.Counter; member=CounterSignal
   int32 2
signal time=1635329227.883016 sender=:1.52 -> destination=(null destination) serial=4
path=/com/example/counter; interface=com.example.Counter; member=CounterSignal
   int32 3
signal time=1635329228.884821 sender=:1.52 -> destination=(null destination) serial=5
path=/com/example/counter; interface=com.example.Counter; member=CounterSignal
   int32 4
signal time=1635329229.886635 sender=:1.52 -> destination=(null destination) serial=6
path=/com/example/counter; interface=com.example.Counter; member=CounterSignal
   int32 5
signal time=1635329230.888305 sender=:1.52 -> destination=(null destination) serial=7
path=/com/example/counter; interface=com.example.Counter; member=CounterSignal
   int32 6
```

7. Summary

You now officially know the D-Bus basics that you need to be able to use BlueZ from your own code. In summary, you should now know how to:

- Establish a D-Bus connection with the system bus or session bus
- Create proxy objects
- Call remote methods implemented by objects and interfaces owned by other D-Bus applications
- Register for and receive signals
- Export objects, interfaces and methods which may then be called by other processes
- Set up a basic D-Bus security policy
- Emit signals
- Use the tools D-Feet, dbus-monitor and dbus-send.

Move on to the next module when ready.

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 3

2. INTRODUCTION 4

3. DEVICE DISCOVERY 4

3.1 Device Discovery and BlueZ 4

3.2 Implementing Device Discovery 10

3.2.1 StartDiscovery and InterfacesAdded 11

3.2.3 Adding a time limit 14

3.2.3 InterfacesRemoved 16

3.2.4 PropertiesChanged 19

3.2.5 GetManagedObjects 23

4. CONNECTING AND DISCONNECTING 28

4.1 Connecting 28

4.2 Disconnecting 31

4.3 Extra Credit 32

5. SERVICE DISCOVERY 33

5.1 BlueZ and Service Discovery 33

5.2 Getting Started 37

5.3 Tracking Service Discovery 39

5.4 Validating Services Found 41

6. READING CHARACTERISTICS 47

6.1 Obtaining service and characteristic paths 49

6.2 Reading the temperature characteristic value 53

7. WRITING CHARACTERISTICS 57

7.1 Writing to the LED Text characteristic from Python 57

7.2 Write Requests vs Write Commands 61

8. USING NOTIFICATIONS 62

8.1 Creating the client_monitor_temperature.py script 62

9. SUMMARY 67

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.

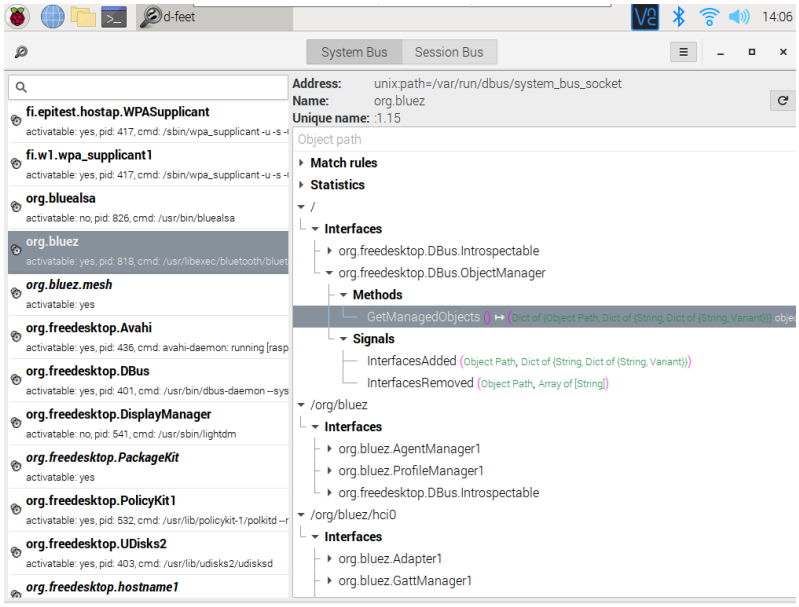


Figure 2 - The ObjectManager interface of the BlueZ root object

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:

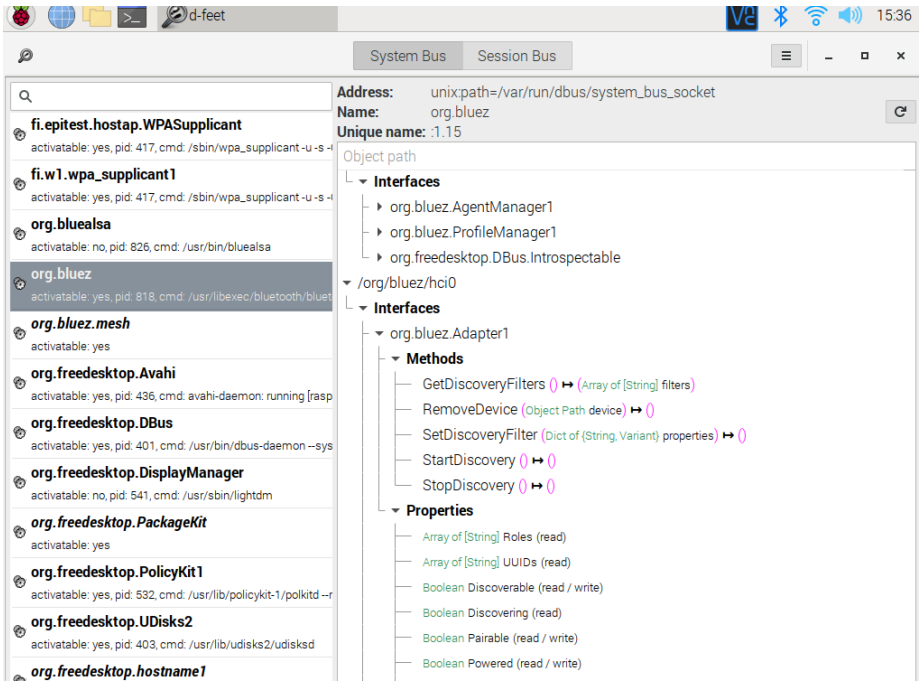


Figure 5 - The Adapter1 interface

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.

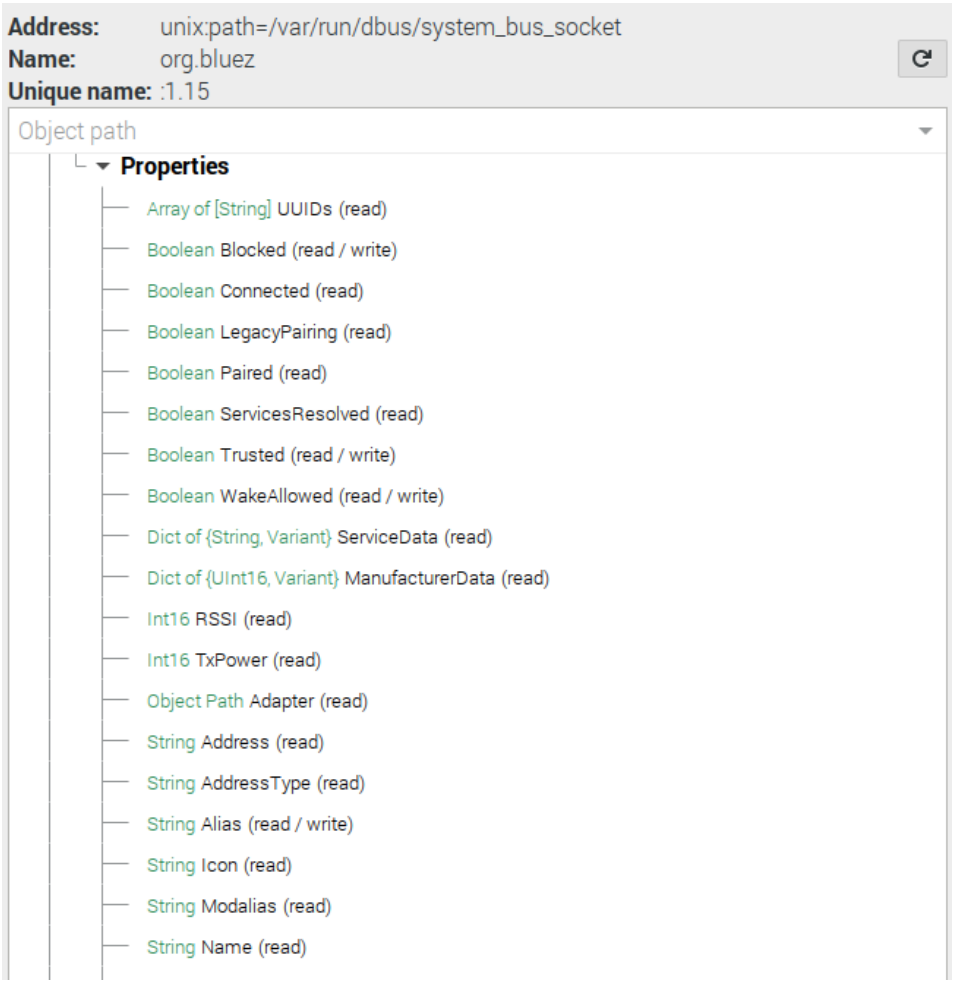


Figure 6 - some of the properties of a Device object

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.

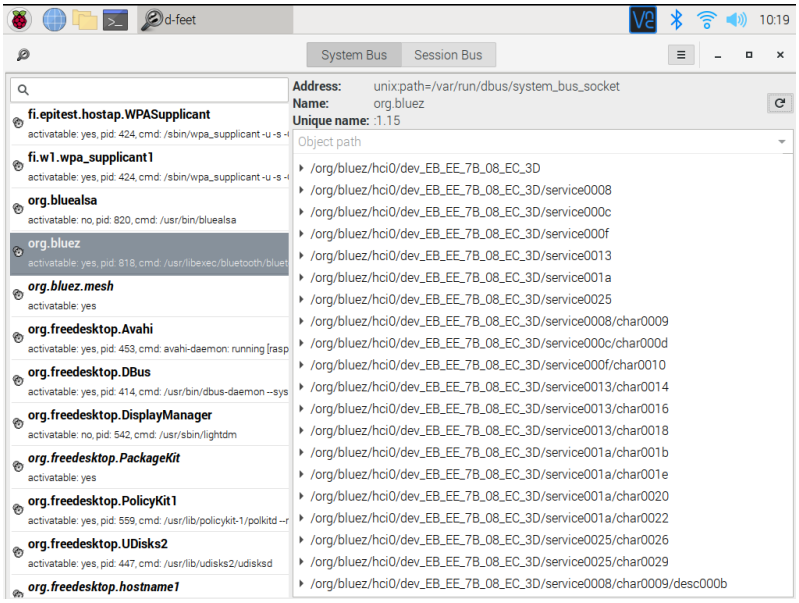


Figure 7 - a device, its services, characteristics and one descriptor as D-Bus objects

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/

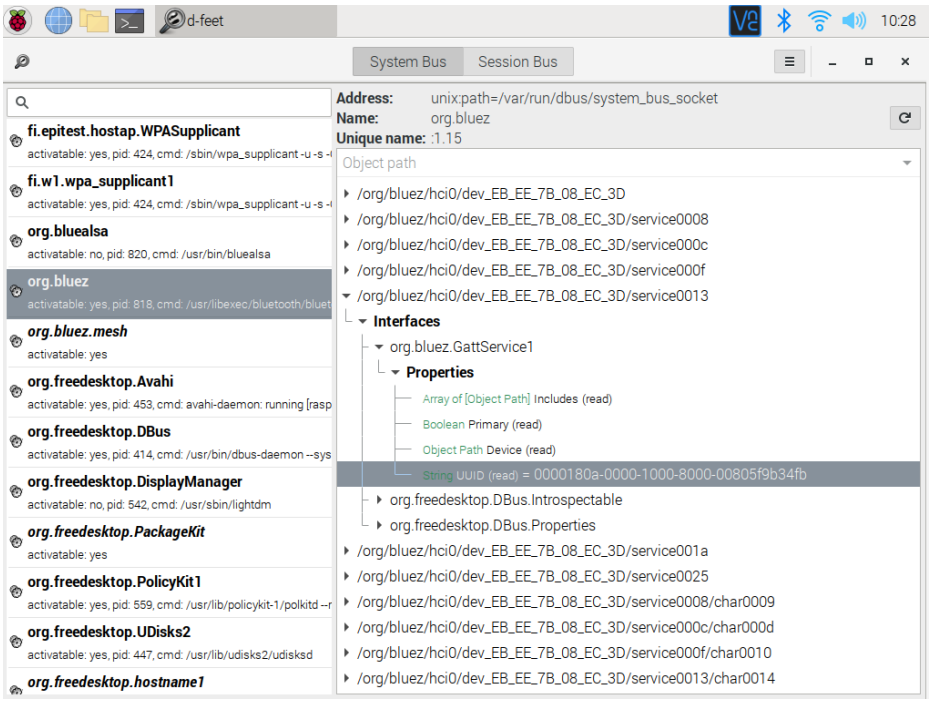


Figure 8 - A service object whose UUID property identifies it as representing the Device Information Service

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.

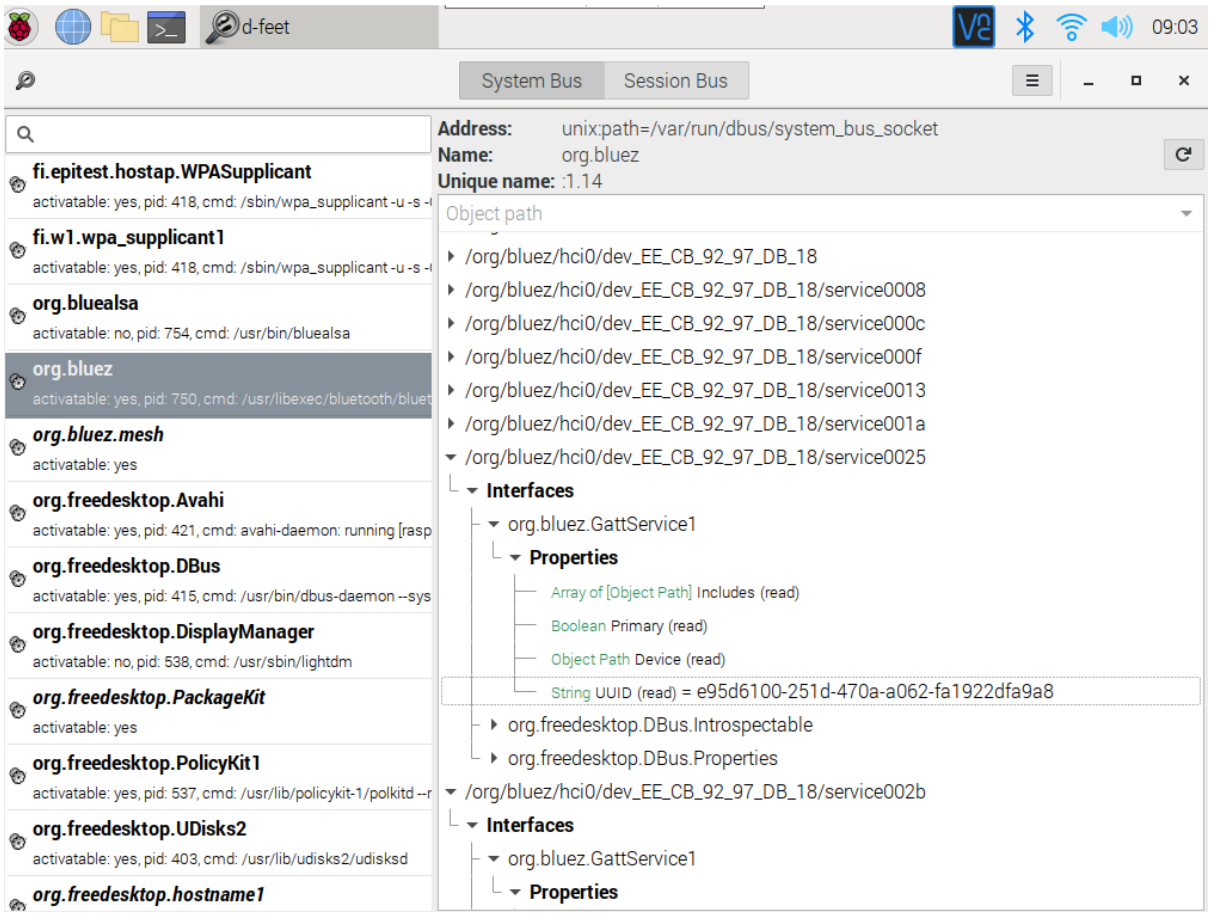


Figure 9 - service0025 has been found to represent the temperature service on this device

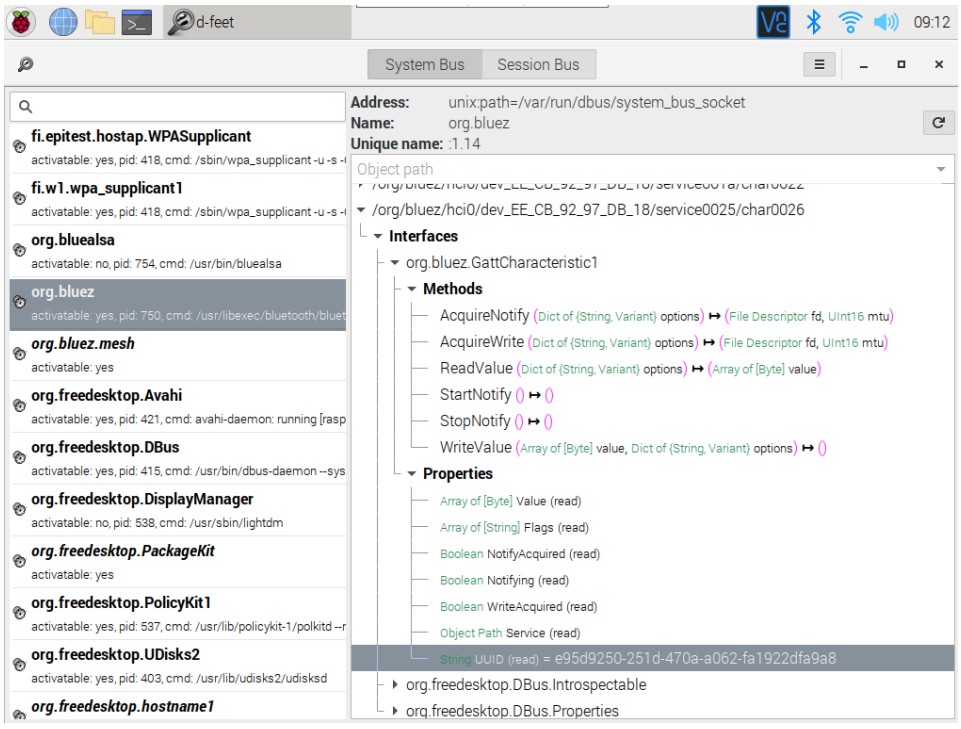


Figure 10 - char0026 within service0025 is the temperature characteristic on this device

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

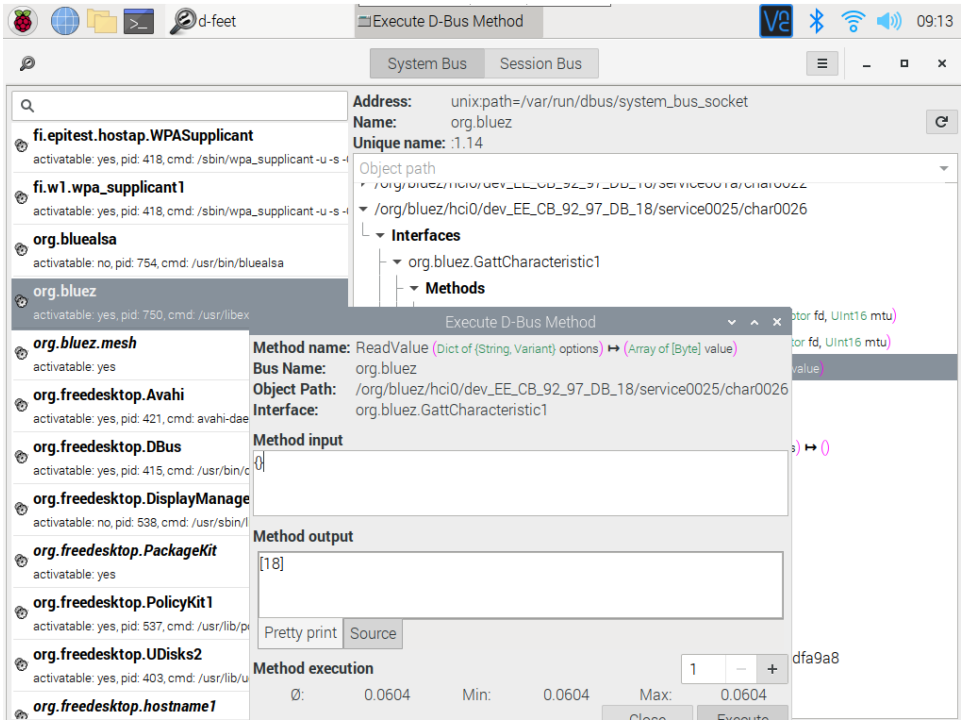


Figure 11 - Executing ReadValue within D-Feet

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.

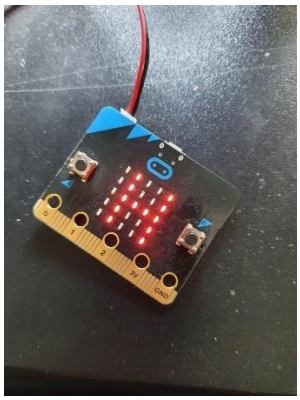


Figure 12 - Text scrolling across the micro:bit display

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.

Bluetooth for Linux Developers Study Guide

Developing LE Peripheral Devices using Python

Release : 1.0.1

Document Version: 1.0.0

Last updated : 16th November 2021

Contents

1. REVISION HISTORY 3

2. INTRODUCTION 4

3. ADVERTISING 5

3.1 Getting Started 5

3.2 Advertisement Properties 7

3.3 Advertising 10

4. HANDLING CONNECTION REQUESTS AND DISCONNECTIONS 15

4.1 Connection event handling with BlueZ and DBus 15

4.2 Handling connection and disconnect events 17

5. IMPLEMENTING GATT SERVICES AND CHARACTERISTICS 21

5.1 GATT with BlueZ and DBus 21

5.2 The Superclasses 21

5.3 The Temperature Service 23

6. HANDLING CHARACTERISTIC READ REQUESTS 28

7. NOTIFYING 29

8. HANDLING CHARACTERISTIC WRITES 32

9. PROPERTIES 34

10. PERMISSIONS 36

10.1 Access Permissions 36

10.2 Encryption Permissions 37

10.2.1 Setting the encryption flag in code 37

10.2.2 Pairing 37

10.2.3 Just Works pairing 38

10.2.4 Passkey Pairing 39

10.3 Authentication Permissions 41

10.4 Authorization Permissions 41

11. SUMMARY 43

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 examine and learn about the most important use cases for Peripheral GATT server devices including advertising, handling connect requests and disconnections, characteristic read and write requests and generating characteristic notifications. We'll also examine how to use the various types of permission flag that can be associated with a characteristic.

You'll be given the opportunity to complete exercises where you will write code that exhibits specified functionality.

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 Central device like a smartphone to test with.

3. Advertising

Advertising is a procedure which involves the periodic broadcasting of small packets of data. The payload of such packets consists of a series of one or more fields in the Tag Length Value (TLV) format. The types which may be advertised are defined in the Bluetooth Core Specification Supplement (CSS).

BlueZ defines an API for advertising which includes the LEAdvertisement1 and the LEAdvertisingManager1 interfaces. To advertise requires code to do the following:

1. Implement a class that defines your Advertisement object and is a sub-class of dbus.service.Object. Allocate to it a path to act as its identifier when it is registered with DBus.
2. Assign values to properties which will become TLV fields that appear in the payload of advertising packets according to your requirements. Remember that advertising packets can contain no more than 31 bytes in the payload field.
3. Implement the GetAll method of the org.freedesktop.DBus.Properties interface.
4. Implement the Release method of the org.bluez. LEAdvertisingManager1 interface.
5. Obtain an instance of the LEAdvertisingManager1 interface from the adapter.
6. Create an instance of your Advertisement class and register it with DBus using the LEAdvertisingManager1 interface. This will cause advertising to start.

3.1 Getting Started

Create a new script file called server_advertising.py containing the following code.

```
#!/usr/bin/python3
# Broadcasts connectable advertising packets

import bluetooth_constants
import bluetooth_exceptions
import dbus
import dbus.exceptions
import dbus.service
import dbus.mainloop.glib
import sys
from gi.repository import GLib
sys.path.insert(0, '.')

bus = None
adapter_path = None
adv_mgr_interface = None

# much of this code was copied or inspired by test\example-advertisement in
the BlueZ source
class Advertisement(dbus.service.Object):
    PATH_BASE = '/org/bluez/ldsg/advertisement'

    def __init__(self, bus, index, advertising_type):
```

```
        self.path = self.PATH_BASE + str(index)
        self.bus = bus
        self.ad_type = advertising_type
        self.service_uuids = None
        self.manufacturer_data = None
        self.solicit_uuids = None
        self.service_data = None
        self.local_name = 'Hello'
        self.include_tx_power = False
        self.data = None
        self.discoverable = True
        dbus.service.Object.__init__(self, bus, self.path)



dbus.mainloop.glib.DBusGMainLoop(set_as_default=True)
bus = dbus.SystemBus()
# we're assuming the adapter supports advertising
adapter_path = bluetooth_constants.BLUEZ_NAMESPACE +
bluetooth_constants.ADAPTER_NAME
print(adapter_path)
```

Note the first comment. The BlueZ source contains a number of examples which may be informative and this is one occasion where that is the case and the example.advertisement.py script has been plundered to help us get started.

There's another comment which indicates an assumption is being made regarding the adapter. Not all Bluetooth controllers support advertising. Some only support the procedures required by one particular GAP role such as the GAP Central role. Some support only those required by the GAP Peripheral role (which includes advertising). Some support more than one GAP role and associated procedures. In our case we want to advertise and this means the DBus adapter object must implement the org.bluez.LEAdvertisingManager1 interface as shown in figure 1.

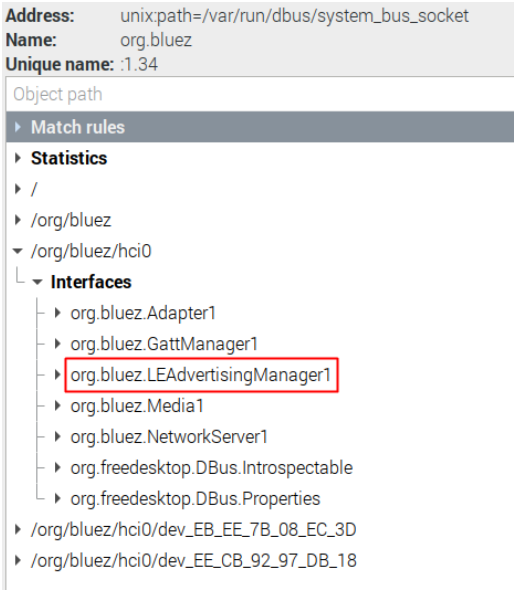


Figure 1 - This adapter implements LEAdvertisingManager1

This code was written specifically for a Raspberry Pi 4 whose adapter is known to support advertising and so we're taking advantage of this known fact about the target platform. The original BlueZ test/example-advertisement.py script shows how to obtain an adapter which supports advertising, assuming the system has one.

Note the way a path is constructed from a base value of '/org/bluez/ldsg/advertisement' plus an index value which is to be provided to the constructor when an Advertisement object is instantiated. You can use any value for the path name as indicated by the phrase *Object path freely definable* in the BlueZ advertising-api.txt documentation. The value used here was chosen to be consistent with those used by BlueZ itself.

What we have at this stage is a basic framework to extend. It creates a class called Advertisement which extends the dbus.service.Object class as required (this allows us to register it with DBus) and the constructor creates properties corresponding to advertising TLV fields and assigns one of them, *local_name* a value of 'hello'. You'll see this value in advertising packets later on.

Run this code now just to check there are no typos.

```
pi@raspberrypi:~/projects/ldsg/solutions/python/bluetooth $ python3 wip.py
/org/bluez/hci0
```

The code runs, displays the path name of the adapter and exits. That's all we expect at this stage.

3.2 Advertisement Properties

Next, we'll ensure the properties of our Advertisement class can be retrieved by BlueZ by implementing and exporting the DBus properties GetAll method. At the same time, we'll add a method that returns the path of an Advertisement object, just so we can log it to the console and we'll implement the Release method of the advertising manager interface. Update your code so that it looks like this:

```
#!/usr/bin/python3
# Broadcasts connectable advertising packets

import bluetooth_constants
import bluetooth_exceptions
import dbus
import dbus.exceptions
import dbus.service
import dbus.mainloop.glib
import sys
from gi.repository import GLib
sys.path.insert(0, '.')

bus = None
adapter_path = None
adv_mgr_interface = None

# much of this code was copied or inspired by test\example-advertisement in
the BlueZ source
class Advertisement(dbus.service.Object):
    PATH_BASE = '/org/bluez/ldsg/advertisement'

    def __init__(self, bus, index, advertising_type):
        self.path = self.PATH_BASE + str(index)
        self.bus = bus
        self.ad_type = advertising_type
        self.service_uuids = None
        self.manufacturer_data = None
        self.solicit_uuids = None
        self.service_data = None
        self.local_name = 'Hello'
        self.include_tx_power = False
        self.data = None
        self.discoverable = True
        dbus.service.Object.__init__(self, bus, self.path)

    def get_properties(self):
        properties = dict()
        properties['Type'] = self.ad_type
        if self.service_uuids is not None:
            properties['ServiceUUIDs'] = dbus.Array(self.service_uuids,
                                                    signature='s')
        if self.solicit_uuids is not None:
            properties['SolicitUUIDs'] = dbus.Array(self.solicit_uuids,
                                                    signature='s')
        if self.manufacturer_data is not None:
            properties['ManufacturerData'] = dbus.Dictionary(
                self.manufacturer_data, signature='qv')
```

```
        if self.service_data is not None:
            properties['ServiceData'] = dbus.Dictionary(self.service_data,
                                                        signature='sv')
        if self.local_name is not None:
            properties['LocalName'] = dbus.String(self.local_name)
        if self.discoverable is not None and self.discoverable == True:
            properties['Discoverable'] = dbus.Boolean(self.discoverable)
        if self.include_tx_power:
            properties['Includes'] = dbus.Array(["tx-power"], signature='s')

        if self.data is not None:
            properties['Data'] = dbus.Dictionary(
                self.data, signature='yv')
        print(properties)
        return {bluetooth_constants.ADVERTISING_MANAGER_INTERFACE:
properties}

    def get_path(self):
        return dbus.ObjectPath(self.path)

    @dbus.service.method(bluetooth_constants.DBUS_PROPERTIES,
                         in_signature='s',
                         out_signature='a{sv}')
    def GetAll(self, interface):
        if interface != bluetooth_constants.ADVERTISEMENT_INTERFACE:
            raise bluetooth_exceptions.InvalidArgsException()
        return
self.get_properties()[bluetooth_constants.ADVERTISING_MANAGER_INTERFACE]

    @dbus.service.method(bluetooth_constants.ADVERTISING_MANAGER_INTERFACE,
                         in_signature='',
                         out_signature='')
    def Release(self):
        print('%s: Released' % self.path)

dbus.mainloop.glib.DBusGMainLoop(set_as_default=True)
bus = dbus.SystemBus()
# we're assuming the adapter supports advertising
adapter_path = bluetooth_constants.BLUEZ_NAMESPACE +
bluetooth_constants.ADAPTER_NAME
print(adapter_path)
```

Note the following points relating to the latest changes:

- There are two instance of the @dbus.service.method decorator. We use this to export the GetAll properties method and the advertising manager Release method.
- GetAll returns a dictionary of named property values which will be turned into advertising packet TLV fields once we start advertising.
- The property Discoverable is set to True. This causes the Flags field to be included in advertising packets with bits set to indicate *General Discoverable Mode* (see core specification section 9.2.4 General Discoverable mode in Volume 3 Part C).
- The Release method is called when an Advertisement object is removed by BlueZ from the bluetooth daemon. According to the BlueZ advertising-api.txt file, it provides an opportunity to perform cleanup tasks. We're just logging the fact that a call to the method has been received out of interest.

3.3 Advertising

We now have everything in place to allow us to start advertising. This is accomplished by creating an instance of the Advertisement class and registering its path with DBus using the LEAdvertisingManager1 method RegisterAdvertisement. We need to run an event loop so that advertising will continue without our code exiting.

Make the highlighted changes to your code:

```
#!/usr/bin/python3
# Broadcasts connectable advertising packets

import bluetooth_constants
import bluetooth_exceptions
import dbus
import dbus.exceptions
import dbus.service
import dbus.mainloop.glib
import sys
from gi.repository import GLib
sys.path.insert(0, '.')

bus = None
adapter_path = None
adv_mgr_interface = None

# much of this code was copied or inspired by test\example-advertisement in
the BlueZ source
class Advertisement(dbus.service.Object):
    PATH_BASE = '/org/bluez/ldsg/advertisement'

    def __init__(self, bus, index, advertising_type):
        self.path = self.PATH_BASE + str(index)
        self.bus = bus
        self.ad_type = advertising_type
        self.service_uuids = None
        self.manufacturer_data = None
```

```
        self.solicit_uuids = None
        self.service_data = None
        self.local_name = 'Hello'
        self.include_tx_power = False
        self.data = None
        self.discoverable = True
        dbus.service.Object.__init__(self, bus, self.path)

    def get_properties(self):
        properties = dict()
        properties['Type'] = self.ad_type
        if self.service_uuids is not None:
            properties['ServiceUUIDs'] = dbus.Array(self.service_uuids,
                                                    signature='s')
        if self.solicit_uuids is not None:
            properties['SolicitUUIDs'] = dbus.Array(self.solicit_uuids,
                                                    signature='s')
        if self.manufacturer_data is not None:
            properties['ManufacturerData'] = dbus.Dictionary(
                self.manufacturer_data, signature='qv')
        if self.service_data is not None:
            properties['ServiceData'] = dbus.Dictionary(self.service_data,
                                                        signature='sv')
        if self.local_name is not None:
            properties['LocalName'] = dbus.String(self.local_name)
        if self.discoverable is not None and self.discoverable == True:
            properties['Discoverable'] = dbus.Boolean(self.discoverable)
        if self.include_tx_power:
            properties['Includes'] = dbus.Array(["tx-power"], signature='s')

        if self.data is not None:
            properties['Data'] = dbus.Dictionary(
                self.data, signature='yv')
        print(properties)
        return {bluetooth_constants.ADVERTISING_MANAGER_INTERFACE:
properties}

    def get_path(self):
        return dbus.ObjectPath(self.path)

    @dbus.service.method(bluetooth_constants.DBUS_PROPERTIES,
                         in_signature='s',
                         out_signature='a{sv}')
    def GetAll(self, interface):
        if interface != bluetooth_constants.ADVERTISEMENT_INTERFACE:
            raise bluetooth_exceptions.InvalidArgsException()
        return
self.get_properties()[bluetooth_constants.ADVERTISING_MANAGER_INTERFACE]
```

```
    @dbus.service.method(bluetooth_constants.ADVERTISING_MANAGER_INTERFACE,
                         in_signature='',
                         out_signature='')
    def Release(self):
        print('%s: Released' % self.path)

def register_ad_cb():
    print('Advertisement registered OK')

def register_ad_error_cb(error):
    print('Error: Failed to register advertisement: ' + str(error))
    mainloop.quit()

def start_advertising():
    global adv
    global adv_mgr_interface
    # we're only registering one advertisement object so index (arg2) is
hard coded as 0
    print("Registering advertisement",adv.get_path())
    adv_mgr_interface.RegisterAdvertisement(adv.get_path(), {},
                                            reply_handler=register_ad_cb,
                                            error_handler=register_ad_error_cb)

dbus.mainloop.glib.DBusGMainLoop(set_as_default=True)
bus = dbus.SystemBus()
# we're assuming the adapter supports advertising
adapter_path = bluetooth_constants.BLUEZ_NAMESPACE +
bluetooth_constants.ADAPTER_NAME
print(adapter_path)

adv_mgr_interface =
dbus.Interface(bus.get_object(bluetooth_constants.BLUEZ_SERVICE_NAME,adapter
_path), bluetooth_constants.ADVERTISING_MANAGER_INTERFACE)
# we're only registering one advertisement object so index (arg2) is hard
coded as 0
adv = Advertisement(bus, 0, 'peripheral')
start_advertising()

print("Advertising as "+adv.local_name)

mainloop = GLib.MainLoop()
mainloop.run()
```

Points to note:

- The advertising manager interface is obtained from the adapter object

- We create an Advertisement object and indicate a type value of ‘peripheral’. The alternative value is ‘broadcast’. These values correspond to the GAP roles of these names.
- In the new start_advertising function we call RegisterAdvertisement. This causes BlueZ to instruct the controller to start advertising with properties obtained from the Advertisement object via the GetAll method which is called by BlueZ.

Run your code now. It should produce this output:

```
pi@raspberrypi:~/projects/ldsg/solutions/python/bluetooth $ python3 server_advertising.py
/org/bluez/hci0
Registering advertisement /org/bluez/ldsg/advertisement0
Advertising as Hello
{'Type': 'peripheral', 'LocalName': dbus.String('Hello')}
Advertisement registered OK
```

Using a suitable application such as nRF Connect for Android or iOS, perform device discovery by scanning. You will see your Linux device advertising with the local name of “Hello” as shown in Figure 2.

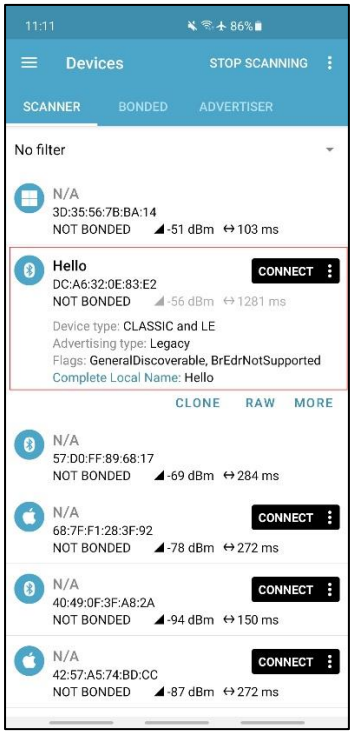


Figure 2 - Device advertising with local name "Hello" and Flags indicating general discoverable mode

Rather than use a smartphone application, if you have another Linux computer with Bluetooth support you can use the BlueZ tool *bluetoothctl* to scan and report discovered devices like this:

```
pi@ssmsprimary:~ $ bluetoothctl
Agent registered
[bluetooth]# scan on
Discovery started
[CHG] Controller E4:5F:01:01:42:EE Discovering: yes
[NEW] Device 68:7F:F1:28:3F:92 68-7F-F1-28-3F-92
[NEW] Device DC:A6:32:0E:83:E2 Hello
[NEW] Device 42:57:A5:74:BD:CC 42-57-A5-74-BD-CC
[bluetooth]# scan off
Discovery stopped
```

Using your smartphone app or other tool, connect to the device and after service discovery has completed, you should see the following services:

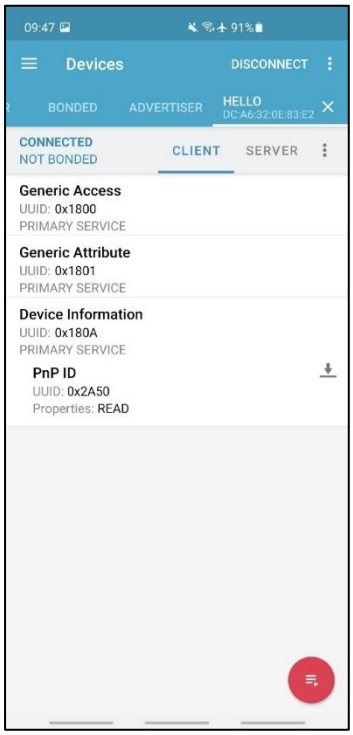


Figure 3 - Default GATT services

These are a combination of mandatory services per the Bluetooth core specification plus the device information service, which BlueZ has created by default.

If you can see your device advertising with a local name of “Hello” then your code is working!

4. Handling Connection Requests and Disconnections

4.1 Connection event handling with BlueZ and DBus

When a remote device connects to an advertising Peripheral, it's common to want to capture this event and take some kind of action. Similarly, if a peer disconnects it can be useful to detect that this has happened.

BlueZ indicates both connections being established and lost using standard DBus signals.

When a connection for a previously unknown device is established, an InterfacesAdded signal is emitted by a device object (i.e. an object which implements the org.bluez.Device1 interface). By "unknown device" we mean a device whose path identifier is not registered to DBus.

```
signal time=1636540910.588206 sender=:1.13 -> destination=(null destination) serial=53
path=/; interface=org.freedesktop.DBus.ObjectManager; mem
ber=InterfacesAdded
   object path "/org/bluez/hci0/dev_7F_3C_14_39_EB_90"
   array [
      dict entry(
         string "org.freedesktop.DBus.Introspectable"
         array [
         ]
      )
      dict entry(
         string "org.bluez.Device1"
         array [
            dict entry(
               string "Address"
               variant                    string "7F:3C:14:39:EB:90"
            )
            dict entry(
               string "AddressType"
               variant                    string "random"
            )
            dict entry(
               string "Alias"
               variant                    string "7F-3C-14-39-EB-90"
            )
            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 "Connected"
               variant                    boolean true
            )
            dict entry(
               string "UUIDs"
               variant                    array [
                  ]
            )
            dict entry(
               string "Adapter"
               variant                    object path "/org/bluez/hci0"
            )
            dict entry(
               string "ServicesResolved"
               variant                    boolean false
```

```
            )
         ]
      )
      dict entry(
         string "org.freedesktop.DBus.Properties"
         array [
         ]
      )
   ]
```

Note the Connected property in the InterfacesAdded signal.

When a connection for a device which is already known is established, a PropertiesChanged signal is instead emitted with the Connected property set to *true*.

```
signal time=1636541088.149906 sender=:1.13 -> destination=(null destination) serial=85
path=/org/bluez/hci0/dev_7F_3C_14_39_EB_90; interface=org.freedesktop.DBus.Properties;
member=PropertiesChanged
   string "org.bluez.Device1"
   array [
      dict entry(
         string "Connected"
         variant             boolean true
      )
   ]
   array [
   ]
```

Similarly, when a device disconnects, a PropertiesChanged signal is emitted but with the Connected property having a value of *false*.

```
signal time=1636546346.734315 sender=:1.13 -> destination=(null destination) serial=287
path=/org/bluez/hci0/dev_57_B0_FD_AF_2B_C3; interface=org.freedesktop.DBus.Properties;
member=PropertiesChanged
   string "org.bluez.Device1"
   array [
      dict entry(
         string "ServicesResolved"
         variant             boolean false
      )
      dict entry(
         string "Connected"
         variant             boolean false
      )
      dict entry(
         string "UUIDs"
         variant             array [
            ]
      )
   ]
   array [
   ]
```

When a Peripheral is connected to, it's common to want to stop advertising. In some devices this may be mandatory depending on how many concurrent link layer instances are supported. To stop advertising, the UnregisterAdvertisement method of the LEAdvertisingManager1 interface is called with the path with which the advertisement object was originally registered as an argument.

Here's an example:

```
adv_mgr_interface.UnregisterAdvertisement(adv.get_path())
```

4.2 Handling connection and disconnect events

Copy your server_advertising.py script to create a new script called server_connect_disconnect.py.

Modify the new code so that PropertiesChanged and InterfacesAdded signals are received and handled by suitable functions. You already know how to do this.

Informed by the description of DBus signals and connect/disconnect events in section 4.1, in your signal handler functions use the value of the Connected property reported by the received signal to set the status of a variable to 1 (connected) or 0 (not connected) and print a suitable message to the console. When a connection is accepted, stop advertising by unregistering your Advertisement object. When the connection is lost, start advertising again.

Test your code.

One possible solution looks like this:

```
#!/usr/bin/python3
# Broadcasts connectable advertising packets

import bluetooth_constants
import bluetooth_exceptions
import dbus
import dbus.exceptions
import dbus.service
import dbus.mainloop.glib
import sys
from gi.repository import GLib
sys.path.insert(0, '.')

bus = None
adapter_path = None
adv_mgr_interface = None
connected = 0

# much of this code was copied or inspired by test\example-advertisement in
the BlueZ source
class Advertisement(dbus.service.Object):
    PATH_BASE = '/org/bluez/ldsg/advertisement'

    def __init__(self, bus, index, advertising_type):
        self.path = self.PATH_BASE + str(index)
        self.bus = bus
        self.ad_type = advertising_type
        self.service_uuids = None
        self.manufacturer_data = None
        self.solicit_uuids = None
        self.service_data = None
        self.local_name = 'Hello'
        self.include_tx_power = False
        self.data = None
```

```
        self.discoverable = True
        dbus.service.Object.__init__(self, bus, self.path)

    def get_properties(self):
        properties = dict()
        properties['Type'] = self.ad_type
        if self.service_uuids is not None:
            properties['ServiceUUIDs'] = dbus.Array(self.service_uuids,
                                                    signature='s')
        if self.solicit_uuids is not None:
            properties['SolicitUUIDs'] = dbus.Array(self.solicit_uuids,
                                                    signature='s')
        if self.manufacturer_data is not None:
            properties['ManufacturerData'] = dbus.Dictionary(
                self.manufacturer_data, signature='qv')
        if self.service_data is not None:
            properties['ServiceData'] = dbus.Dictionary(self.service_data,
                                                        signature='sv')
        if self.local_name is not None:
            properties['LocalName'] = dbus.String(self.local_name)
        if self.discoverable is not None and self.discoverable == True:
            properties['Discoverable'] = dbus.Boolean(self.discoverable)
        if self.include_tx_power:
            properties['Includes'] = dbus.Array(["tx-power"], signature='s')

        if self.data is not None:
            properties['Data'] = dbus.Dictionary(
                self.data, signature='yv')
        print(properties)
        return {bluetooth_constants.ADVERTISING_MANAGER_INTERFACE:
properties}

    def get_path(self):
        return dbus.ObjectPath(self.path)

    @dbus.service.method(bluetooth_constants.DBUS_PROPERTIES,
                         in_signature='s',
                         out_signature='a{sv}')
    def GetAll(self, interface):
        if interface != bluetooth_constants.ADVERTISEMENT_INTERFACE:
            raise bluetooth_exceptions.InvalidArgsException()
        return
self.get_properties()[bluetooth_constants.ADVERTISING_MANAGER_INTERFACE]

    @dbus.service.method(bluetooth_constants.ADVERTISING_MANAGER_INTERFACE,
                         in_signature='',
                         out_signature='')
    def Release(self):
```

```
        print('%s: Released' % self.path)

def register_ad_cb():
    print('Advertisement registered OK')

def register_ad_error_cb(error):
    print('Error: Failed to register advertisement: ' + str(error))
    mainloop.quit()

def set_connected_status(status):
    global connected
    if (status == 1):
        print("connected")
        connected = 1
        stop_advertising()
    else:
        print("disconnected")
        connected = 0
        start_advertising()

def properties_changed(interface, changed, invalidated, path):
    if (interface == bluetooth_constants.DEVICE_INTERFACE):
        if ("Connected" in changed):
            set_connected_status(changed["Connected"])

def interfaces_added(path, interfaces):
    if bluetooth_constants.DEVICE_INTERFACE in interfaces:
        properties = interfaces[bluetooth_constants.DEVICE_INTERFACE]
        if ("Connected" in properties):
            set_connected_status(properties["Connected"])

def stop_advertising():
    global adv
    global adv_mgr_interface
    print("Unregistering advertisement",adv.get_path())
    adv_mgr_interface.UnregisterAdvertisement(adv.get_path())

def start_advertising():
    global adv
    global adv_mgr_interface
    # we're only registering one advertisement object so index (arg2) is
hard coded as 0
    print("Registering advertisement",adv.get_path())
    adv_mgr_interface.RegisterAdvertisement(adv.get_path(), {},
                                            reply_handler=register_ad_cb,
                                            error_handler=register_ad_error_cb)

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

```
bus = dbus.SystemBus()
# we're assuming the adapter supports advertising
adapter_path = bluetooth_constants.BLUEZ_NAMESPACE +
bluetooth_constants.ADAPTER_NAME
print(adapter_path)

bus.add_signal_receiver(properties_changed,
        dbus_interface = bluetooth_constants.DBUS_PROPERTIES,
        signal_name = "PropertiesChanged",
        path_keyword = "path")

bus.add_signal_receiver(interfaces_added,
        dbus_interface = bluetooth_constants.DBUS_OM_IFACE,
        signal_name = "InterfacesAdded")

adv_mgr_interface =
dbus.Interface(bus.get_object(bluetooth_constants.BLUEZ_SERVICE_NAME,adapter
_path), bluetooth_constants.ADVERTISING_MANAGER_INTERFACE)
# we're only registering one advertisement object so index (arg2) is hard
coded as 0
adv = Advertisement(bus, 0, 'peripheral')
start_advertising()

print("Advertising as "+adv.local_name)

mainloop = GLib.MainLoop()
mainloop.run()
```

Test your code by running it and then connecting with another device such as your smartphone and the nRF Connect application. Then disconnect. And reconnect again. And disconnect again. You should see something like this:

```
pi@raspberrypi:~/projects/ldsg/solutions/python/bluetooth $ python3
server_connect_disconnect.py
/org/bluez/hci0
Registering advertisement /org/bluez/ldsg/advertisement0
Advertising as Hello
{'Type': 'peripheral', 'LocalName': dbus.String('Hello'), 'Discoverable':
dbus.Boolean(True)}
Advertisement registered OK
connected
Unregistering advertisement /org/bluez/ldsg/advertisement0
disconnected
Registering advertisement /org/bluez/ldsg/advertisement0
{'Type': 'peripheral', 'LocalName': dbus.String('Hello'), 'Discoverable':
dbus.Boolean(True)}
Advertisement registered OK
connected
Unregistering advertisement /org/bluez/ldsg/advertisement0
disconnected
Registering advertisement /org/bluez/ldsg/advertisement0
{'Type': 'peripheral', 'LocalName': dbus.String('Hello'), 'Discoverable':
dbus.Boolean(True)}
Advertisement registered OK
```

5. Implementing GATT Services and Characteristics

5.1 GATT with BlueZ and DBus

GAP Peripherals are typicically also GATT servers. This means that a number of GATT services with their constituent characteristics and descriptors are implemented, can be discovered by a connected GATT client and operations such as read, write and notify can be used with the characteristics depending on which operations each characteristic supports.

BlueZ and DBus use an object hierarchy which mirrors the hierarchical relationship of services, characteristics and descriptors which collectively are implemented by a device or *application*. As such, when writing code which acts as a GAP Peripheral and GATT server, in addition to everything that was covered in sections 3 and 4, it is necessary to create a containment hierarchy of application, service, characteristic and descriptor classes (or similar, depending on the programming language).

From an object-orientation point of view, we are never interested solely in implementing classes which represent services, characteristics and descriptors in general. Our interest is always in implementing concrete subclasses of these abstract ideas. So services like the Immediate Alert service are generally implemented as a subclass of an abstract class called *service*. Details will of course vary according to the programming language you are using to implement and BlueZ itself doesn't care what your code looks like, only that you register the right objects and export the required methods and interfaces needed for the system to work.

In fact, as will become clear as we start to write code, we only need to explicitly register the top level object in our hierarchy i.e. the path for our *application* object. This object must implement the GetManagedObjects method of the org.freedesktop.DBus.ObjectManager interface and it is this method which is called by BlueZ and returns information about the services, characteristics and descriptors implemented by the application.

5.2 The Superclasses

Whilst working through this section, have the BlueZ GATT API documentation open.

Classes which implement the generally applicable capabilities and properties of services, characteristics and descriptors have been written already and can be found implemented in the bluetooth_gatt.py script. Open this file now and review.

Some key points to note, starting with the Service class are as follows:

```
class Service(dbus.service.Object):
    """
    org.bluez.GattService1 interface implementation
    """

    def __init__(self, bus, path_base, index, uuid, primary):
        self.path = path_base + "/service" + str(index)
        self.bus = bus
        self.uuid = uuid
        self.primary = primary
        self.characteristics = []
        dbus.service.Object.__init__(self, bus, self.path)
```

- Service extends dbus.service.Object and therefore can be exported to DBus.
- The constructor has a means of deriving a path value from a base, an index (intended to be different for each concrete subclass) and a literal value of "/service".
- The constructor also requires a DBus bus object, a UUID with which GATT clients can identify the type of GATT service represented and an indicator of whether it is a primary or secondary service.
- There's also an empty array that will hold the characteristics belonging to the service when a concrete subclass is created.

You'll also notice that a method GetAll is exported:

```
    @dbus.service.method(bluetooth_constants.DBUS_PROPERTIES,
                         in_signature='s',
                         out_signature='a{sv}')
    def GetAll(self, interface):
        if interface != bluetooth_constants.GATT_SERVICE_INTERFACE:
            raise
bluetooth_exceptions.bluetooth_exceptions.InvalidArgsException()

        return
self.get_properties()[bluetooth_constants.GATT_SERVICE_INTERFACE]
```

This is one of the methods of the org.freedesktop.DBus.Properties interface and allows discovery of the properties of the Service object. The properties themselves are returned in the get_properties() method and consist of the service UUID, the primary/secondary service indicator and the array of characteristics which belong to the service. Remember we're looking at a superclass here and we won't see concrete details like specific UUIDs until we instantiate our own services which extend this Service class in our own code.

```
    def get_properties(self):
        return {
                bluetooth_constants.GATT_SERVICE_INTERFACE: {
                        'UUID': self.uuid,
                        'Primary': self.primary,
                        'Characteristics': dbus.Array(
                                self.get_characteristic_paths(),
                                signature='o')
                }
        }
```

The Characteristic class follows a similar pattern. In addition to the properties set using its constructor and the exporting of the GetAll method, you can see skeleton implementations of BlueZ GATT API functions ReadValue, WriteValue, StartNotify and StopNotify. In all cases, these methods should never be called directly but instead should be overridden by a class which extends Characteristic and represents a specific characteristic type. Much the same can be said of the Descriptor class.

5.3 The Temperature Service

If you completed module 05, you'll have encountered the temperature service used by the BBC micro:bit. We'll mimic that service by implementing a service with the same UUID and which contains the same temperature characteristic. Rather than go to the trouble of integrating a real temperature sensor though, we'll simulate temperature readings and fluctuations.

Copy your most recent work, which should be called server_connect_disconnect.py or similar to a new source file called server_gatt_temp_svc.py. At this stage, this new code advertises and handles connect/disconnect events but does not implement any GATT services.

This script will become quite large as we incrementally develop it so *only changes that need to be made will be presented from now on, not the whole of the script*. A complete solution is included in the study guide for you to consult if you run into problems.

Our first task will be to modify the script so that a GATT client like the nRF Connect application can discover the temperature service and its temperature characteristic. We will defer implementing support for the characteristic value to be read or to notify until the next section.

Add the following code which includes a class that implements the temperature characteristic and looks like this:

```
import bluetooth_gatt
import random


class TemperatureCharacteristic(bluetooth_gatt.Characteristic):
    temperature = 0
    delta = 0
    notifying = False

    def __init__(self, bus, index, service):
        bluetooth_gatt.Characteristic.__init__(
                self, bus, index,
                bluetooth_constants.TEMPERATURE_CHR_UUID,
                ['read','notify'],
                service)
        self.notifying = False
        self.temperature = random.randint(0, 50)
        print("Initial temperature set to "+str(self.temperature))
        self.delta = 0
```

Note that:

- This class extends bluetooth_gatt.Characteristic which is the superclass for all characteristics which was described in section 5.2.
- Member variables temperature and delta will be used in the simulation of temperature variations which we will implement in the next section.

- The constructor arguments include a bus object, an index value to be used in deriving a value for the object path and a reference to the service (object) that the characteristic is owned by.
- The constructor calls the superclass constructor, passing the bus, index value, the UUID for the temperature characteristic (as used by a BBC micro:bit), the owning service and a list of flag values that indicate the operations supported by this characteristic, namely read and notify.
- An initial characteristic value is selected at random.

Next, we'll create a class for the temperature service. Add the following code:

```
class TemperatureService(bluetooth_gatt.Service):
#   Fake micro:bit temperature service that simulates temperature sensor
measurements
#   ref: https://lancaster-university.github.io/microbit-
docs/resources/bluetooth/bluetooth_profile.html
#   temperature_period characteristic not implemented to keep things simple

     def __init__(self, bus, path_base, index):
        print("Initialising TemperatureService object")
        bluetooth_gatt.Service.__init__(self, bus, path_base, index,
bluetooth_constants.TEMPERATURE_SVC_UUID, True)
        print("Adding TemperatureCharacteristic to the service")
        self.add_characteristic(TemperatureCharacteristic(bus, 0, self))
```

Notes:

- This is the TemperatureService class in its entirety.
- It extends the Service superclass and its constructor calls the superclass constructor with arguments that allow a path to be derived, the service UUID and True indicating that this is a primary service.
- The Service superclass has an array which is used as a container for characteristics owned by the service. An instance of TemperatureCharacteristic is created and added to this array by calling the superclass method add_characteristic.

The new service will not yet be visible to GATT clients. This is because we still need to create an application object which acts as the root of our services/characteristics/descriptors hierarchy, exports methods of its object manager interface and then register it with BlueZ using the GattManager1 interface (see [https://git.kernel.org/pub/scm/bluetooth/bluez.git/tree/doc/gatt-api.txt for details of GattManager1](https://git.kernel.org/pub/scm/bluetooth/bluez.git/tree/doc/gatt-api.txt)). Let's do that now.

Add the following code:

```
class Application(dbus.service.Object):
    """
    org.bluez.GattApplication1 interface implementation
    """
    def __init__(self, bus):
```

```
        self.path = '/'
        self.services = []
        dbus.service.Object.__init__(self, bus, self.path)
        print("Adding TemperatureService to the Application")
        self.add_service(TemperatureService(bus, '/org/bluez/ldsg', 0))

    def get_path(self):
        return dbus.ObjectPath(self.path)

    def add_service(self, service):
        self.services.append(service)

    @dbus.service.method(bluetooth_constants.DBUS_OM_IFACE,
out_signature='a{oa{sa{sv}}}')
    def GetManagedObjects(self):
        response = {}
        print('GetManagedObjects')

        for service in self.services:
            print("GetManagedObjects: service="+service.get_path())
            response[service.get_path()] = service.get_properties()
            chrcs = service.get_characteristics()
            for chrc in chrcs:
                response[chrc.get_path()] = chrc.get_properties()
                descs = chrc.get_descriptors()
                for desc in descs:
                    response[desc.get_path()] = desc.get_properties()
        return response

# under the start_advertising method above the main entry point code
def register_app_cb():
    print('GATT application registered')

def register_app_error_cb(error):
    print('Failed to register application: ' + str(error))
    mainloop.quit()

# towards the end of your code before the mainloop.run() statement

app = Application(bus)

print('Registering GATT application...')

service_manager = dbus.Interface(
        bus.get_object(bluetooth_constants.BLUEZ_SERVICE_NAME,
adapter_path),
        bluetooth_constants.GATT_MANAGER_INTERFACE)

service_manager.RegisterApplication(app.get_path(), {},
```

```
                                    reply_handler=register_app_cb,
                                    error_handler=register_app_error_cb)
```

Notes:

- Application extends dbus.service.Object and so can be registered on a bus
- It includes a list of services and in its constructor it instantiates TemperatureService and adds it to its own list.
- Instantiating the Application object causes TemperatureService to be instantiated and this causes its constructor to instantiate the TemperatureCharacteristic. In each case the constructor of the appropriate superclass is also called. So the Application causes the entire service, characteristic and descriptor object hierarchy to be created.
- The Object Manager interface method GetManagedObjects is exported. This makes the method available to be called by other DBus applications, in our case the BlueZ bluetooth daemon and this is how it determines the list of services, characteristics and descriptors implemented by our application and results in DBus objects for each being registered on the system bus.
- After creating an instance of Application, we acquire the org.bluez.GattManager1 interface from the adapter and then call its RegisterApplication method with the path of our application object and a couple of success/fail callback functions. At this point, the new GATT service and characteristic should be discoverable.

Run your code now and using a suitable application or tool, discover and connect to it. You should see the temperature service and its characteristic as shown in Figure 4.

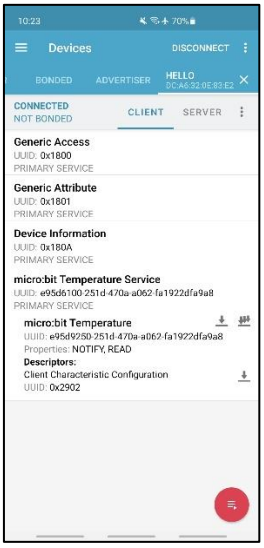


Figure 4 - The temperature service and characteristic

Now try reading the temperature characteristic value.

```
pi@raspberrypi:~/projects/ldsg/solutions/python/bluetooth $ python3 server_gatt_temp_svc.py
/org/bluez/hci0
Registering advertisement /org/bluez/ldsg/advertisement0
```

```
Advertising as Hello
Adding TemperatureService to the Application
Initialising TemperatureService object
Adding TemperatureCharacteristic to the service
creating Characteristic with path=/org/bluez/ldsg/service0/char0
Initial temperature set to 17
Registering GATT application...
GetManagedObjects
GetManagedObjects: service=/org/bluez/ldsg/service0
{'Type': 'peripheral', 'LocalName': dbus.String('Hello'), 'Discoverable':
dbus.Boolean(True)}
GATT application registered
Advertisement registered OK
connected
Unregistering advertisement /org/bluez/ldsg/advertisement0
Default ReadValue called, returning error
```

As expected, we get at error because ReadValue has not been implemented in our TemperatureCharacteristic class and therefore it is the superclass' method that gets called. These methods would be designated *abstract* in some other programming languages in that they must always be overridden in a subclass and never called directly so that's what's happening here.

If you can see the service and characteristic from your GATT client, everything is as it should be at this stage.

6. Handling Characteristic Read Requests

The next task is to implement support for reading the temperature characteristic value.

Add the following method to the TemperatureCharacteristic class:

```
    def ReadValue(self, options):
        print('ReadValue in TemperatureCharacteristic called')
        print('Returning '+str(self.temperature))
        value = []
        value.append(dbus.Byte(self.temperature))
        return value
```

Notes:

- The ReadValue method is specified in the BlueZ gatt-api.txt file and implementing it here overrides the method of the same name in the Characteristic superclass.
- At this stage the temperature value is assigned a random value when the characteristic is created and does not change after that.
- It is a requirement of the BlueZ API that ReadValue returns an array of bytes containing the value. The micro:bit profile defines temperature as a signed 8 bit integer and so only a single byte is required. A byte array containing the temperature value as a single byte is created and returned to the caller (BlueZ over DBus).

Test your code. You should see the same temperature value returned to your GATT client every time your issue a *read* against the temperature characteristic.

```
pi@raspberrypi:~/projects/ldsg/solutions/python/bluetooth $ python3 server_gatt_temp_svc.py
/org/bluez/hci0
Registering advertisement /org/bluez/ldsg/advertisement0
Advertising as Hello
Adding TemperatureService to the Application
Initialising TemperatureService object
Adding TemperatureCharacteristic to the service
creating Characteristic with path=/org/bluez/ldsg/service0/char0
Initial temperature set to 42
Registering GATT application...
GetManagedObjects
GetManagedObjects: service=/org/bluez/ldsg/service0
{'Type': 'peripheral', 'LocalName': dbus.String('Hello'), 'Discoverable':
dbus.Boolean(True)}
GATT application registered
Advertisement registered OK
connected
Unregistering advertisement /org/bluez/ldsg/advertisement0
ReadValue in TemperatureCharacteristic called
Returning 42
ReadValue in TemperatureCharacteristic called
Returning 42
ReadValue in TemperatureCharacteristic called
Returning 42
ReadValue in TemperatureCharacteristic called
Returning 42
```

7. Notifying

The temperature characteristic also supports value notifications. Whether or not a characteristic transmits notifications depends on the value of an associated descriptor called the Client Characteristic Configuration descriptor. BlueZ automatically creates this descriptor whenever a characteristic supports the notify or indicate operations. The BlueZ StartNotify and StopNotify methods are used to update the descriptor value and as the names suggest, start or stop the transmission of notifications from the owning characteristic.

To make this more interesting, we'll start by adding code which will simulate temperature variations so that we will see a series of different values transmitted when notifying. Add the simulate_temperature method to the TemperatureCharacteristic class and call it once a second from a timer which is established in the constructor.

```
class TemperatureCharacteristic(bluetooth_gatt.Characteristic):
    temperature = 0
    delta = 0
    notifying = False

    def __init__(self, bus, index, service):
        bluetooth_gatt.Characteristic.__init__(
                self, bus, index,
                bluetooth_constants.TEMPERATURE_CHR_UUID,
                ['read','notify'],
                service)
        self.notifying = False
        self.temperature = random.randint(0, 50)
        print("Initial temperature set to "+str(self.temperature))
        self.delta = 0
       GLib.timeout_add(1000, self.simulate_temperature)

    def simulate_temperature(self):
        self.delta = random.randint(-1, 1)
        self.temperature = self.temperature + self.delta
        if (self.temperature > 50):
            self.temperature = 50
        elif (self.temperature < 0):
            self.temperature = 0
        print("simulated temperature: "+str(self.temperature)+"C")
        GLib.timeout_add(1000, self.simulate_temperature)
```

Run your code. Try to enable notifications on the temperature characteristic from your GATT client.

You should see a series of temperature values logged to the console but an attempt to start notifications will fail because we haven't yet overridden the StartNotify method in the Characteristic superclass.

```
pi@raspberrypi:~/projects/ldsg/solutions/python/bluetooth $ python3 server_gatt_temp_svc.py
/org/bluez/hci0
```

```
Registering advertisement /org/bluez/ldsg/advertisement0
Advertising as Hello
Adding TemperatureService to the Application
Initialising TemperatureService object
Adding TemperatureCharacteristic to the service
creating Characteristic with path=/org/bluez/ldsg/service0/char0
Initial temperature set to 29
Registering GATT application...
GetManagedObjects
GetManagedObjects: service=/org/bluez/ldsg/service0
{'Type': 'peripheral', 'LocalName': dbus.String('Hello'), 'Discoverable':
dbus.Boolean(True)}
GATT application registered
Advertisement registered OK
simulated temperature: 29C
simulated temperature: 29C
simulated temperature: 30C
simulated temperature: 30C
simulated temperature: 29C
simulated temperature: 29C
connected
Unregistering advertisement /org/bluez/ldsg/advertisement0
simulated temperature: 28C
simulated temperature: 27C
simulated temperature: 28C
simulated temperature: 29C
simulated temperature: 30C
Default StartNotify called, returning error
simulated temperature: 29C
simulated temperature: 29C
simulated temperature: 30C
simulated temperature: 30C
simulated temperature: 29C
simulated temperature: 28C
```

Let's finish the temperature characteristic implementation now by allowing notifications to be started and stopped by a connected client. Update your TemperatureCharacteristic code as follows:

```
    def simulate_temperature(self):
        self.delta = random.randint(-1, 1)
        self.temperature = self.temperature + self.delta
        if (self.temperature > 50):
            self.temperature = 50
        elif (self.temperature < 0):
            self.temperature = 0
        print("simulated temperature: "+str(self.temperature)+"C")
        if self.notifying:
            self.notify_temperature()
        GLib.timeout_add(1000, self.simulate_temperature)

    def notify_temperature(self):
        value = []
        value.append(dbus.Byte(self.temperature))
        print("notifying temperature="+str(self.temperature))
        self.PropertiesChanged(bluetooth_constants.GATT_CHARACTERISTIC_INTER
FACE, { 'Value': value }, [])
        return self.notifying

    def StartNotify(self):
        print("starting notifications")
        self.notifying = True
```

```
    def StopNotify(self):
        print("stopping notifications")
        self.notifying = False
```

Notes:

- simulate_temperature now calls a new function, notify_temperature is the *notifying* member variable is True.
- In notify_temperature we create a byte array containing the temperature value in the same way as was done in ReadValue. We then cause the notification to be transmitted by generating a PropertiesChanged signal from the Characteristic1 interface. Note that the signal is defined in the Characteristic superclass. The signal is received by the BlueZ bluetooth daemon and this causes it to instruct the controller to transmit the notification.
- We now implement the BlueZ GATT API functions StartNotify and StopNotify. These simple functions merely change the value of the *notifying* variable which is tested in the *simulate_temperature* function.

Test your code. You should be able to enable and disable notifications from your GATT client and when enabled, see temperature characteristic value notifications arriving.

8. Handling Characteristic Writes

There are two ways in which write operations can be performed. Each allows a connected GATT client to change the value of a characteristic in a remote GATT server, provided the characteristic supports one or both of the two types of write operation.

The first of the two variants is the Write Request. This involves an *attribute protocol* (ATT) request PDU being sent by the client to the server and a response PDU being sent back from the server to the client. Write Requests are considered reliable, with both acknowledgements taking place at the link layer of the Bluetooth stack and the ATT response providing the client with confirmation that the write was carried out successfully or an error code if it was not. Whatever the outcome, the client at least knows the ATT request was received by the remote application and an attempt made to execute the request.

The second is called Write Without Response (WWR). This involves the client sending a *command* PDU for which no corresponding response is defined in the attribute protocol. WWR commands are charactertised by being fast (there's no need to wait for a reply before sending the next command) but potentially, less reliable than write requests. ATT commands still benefit from acknowledgements at the link layer so that the client will know whether or not the command reached the link layer in the remote device, there is no way for the client to know whether the command successfully made its way up the stack to the application and there's always a risk of issues like buffer overflow hampering this.

The micro:bit LED Service includes a characteristic called LED Text. This characteristic supports the Write Request operation and when implemented on a micro:bit, writing a short ASCII text value to the characteristic results in the text scrolling across the LED matrix display of the device.

Your next task will be carried out solo. You know everything you need to know to complete the exercise without help so this should be a good test of what you've learned so far. Your task is to implement the LED service and its LED Text characteristic. The service and characteristic must be discovered during service discovery and be listed at the client alongside the temperature service and characteristic implemented in sections 6 and 7. The LED Text characteristic must support the write request operation and when written to, the text sent by the client must be logged to the console.

Information you need to complete this task is as follows:

LED Service UUID	Already implemented in bluetooth_constants.py `LED_SVC_UUID = "e95dd91d-251d-470a-a062-fa1922dfa9a8"`
LED Text characteristic UUID	Already implemented in bluetooth_constants.py `LED_TEXT_CHR_UUID = "e95d93ee-251d-470a-a062-fa1922dfa9a8"`
Converting the DBus byte array *value* to ascii text	`ascii_bytes = bluetooth_utils.dbus_to_python(value)` `text = ''.join(chr(i) for i in ascii_bytes)`

Copy your server_gatt_temp_svc.py file to create server_gatt_led_svc.py. Edit this file and add whatever code is required for the goals of this exercise to be accomplished. Test your code when ready and ensure that the new service and characteristic are discovered and that writing to the characteristic works as expected.

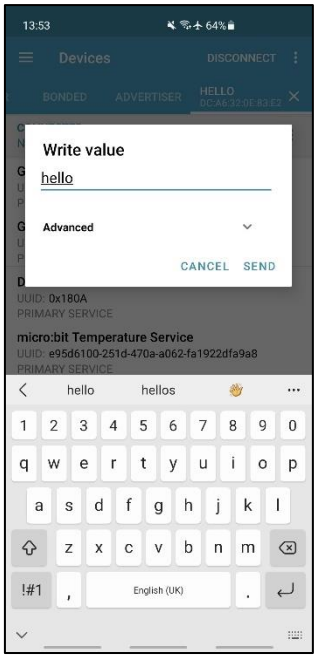


Figure 5 - using nRF Connect to write to the LED Text characteristic

Your test results should look similar to these:

```
pi@raspberrypi:~/projects/ldsg/solutions/python/bluetooth $ python3 server_gatt_led_svc.py
Registering advertisement /org/bluez/ldsg/advertisement0
Adding TemperatureService to the Application
Initialising TemperatureService object
Adding TemperatureCharacteristic to the service
creating Characteristic with path=/org/bluez/ldsg/service0/char0
Initial temperature set to 13
Initialising LedService object
Adding LedTextharacteristic to the service
creating Characteristic with path=/org/bluez/ldsg/service1/char0
Registering GATT application...
GetManagedObjects
GetManagedObjects: service=/org/bluez/ldsg/service0
GetManagedObjects: service=/org/bluez/ldsg/service1
GATT application registered
{'Type': 'peripheral', 'LocalName': dbus.String('Hello'), 'Discoverable':
dbus.Boolean(True)}
Advertisement registered OK
connected
Unregistering advertisement /org/bluez/ldsg/advertisement0
[104, 101, 108, 108, 111] = hello
disconnected
Registering advertisement /org/bluez/ldsg/advertisement0
{'Type': 'peripheral', 'LocalName': dbus.String('Hello'), 'Discoverable':
dbus.Boolean(True)}
Advertisement registered OK
```

9. Properties

All characteristics have a field called the *characteristic properties* field in their definition. This is an 8-bit field of flags which indicate the list of operation types which may be performed against the characteristic value. It is here that the characteristic's definition indicates that it supports, read, write without response and notify operations for example. Section 3.3.1 of the Bluetooth Core Specification, Volume 3 Part G defines the full list of property flags and values.

You have already seen how BlueZ allows properties to be defined for a characteristic using an array of string literal constants in the Flags property of a characteristic. For example, the temperature characteristic sets the Flags field to indicate that read and notify operations are supported by the characteristic.

```
        bluetooth_gatt.Characteristic.__init__(
                self, bus, index,
                bluetooth_constants.TEMPERATURE_CHR_UUID,
                ['read','notify'],
                service)
```

The full list of possible Flags constants is defined in the BlueZ gatt-api.txt file and currently consists of the following values:

```
                                "broadcast"
                                "read"
                                "write-without-response"
                                "write"
                                "notify"
                                "indicate"
                                "authenticated-signed-writes"
                                "extended-properties"
                                "reliable-write"
                                "writable-auxiliaries"
                                "encrypt-read"
                                "encrypt-write"
                                "encrypt-authenticated-read"
                                "encrypt-authenticated-write"
                                "secure-read" (Server only)
                                "secure-write" (Server only)
                                "authorize"
```

To appreciate the effect of these properties, run your latest code. Connect with a GATT client application and note that the properties indicated are those defined in your code. This is a consequence of service discovery delivering details of discovered services, characteristics and descriptors that include the properties field to the client.

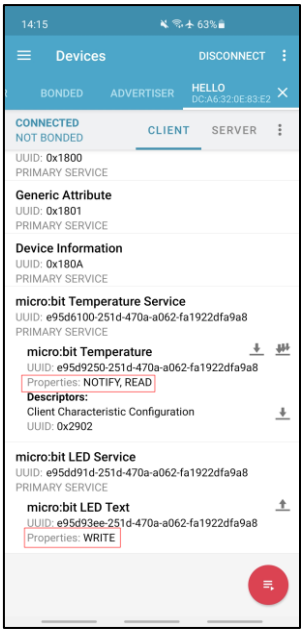


Figure 6 - characteristic properties

As an experiment, replace the 'read' property from your TemperatureCharacteristic class with 'write'. Run the code again and reconnect your client. You should see that the revised property value is reflected in the discovered characteristic details. If you do not, it may be that your device has cached the results of the previous service discovery and you should clear the cache using your client or a suitable tool.

A client application which ignores the property flags on a characteristic and attempts an operation which is not supported should result in a Request Not Supported error (error code 0x06) being returned. You can see an example of this happening in the protocol analyser output in Figure 7.

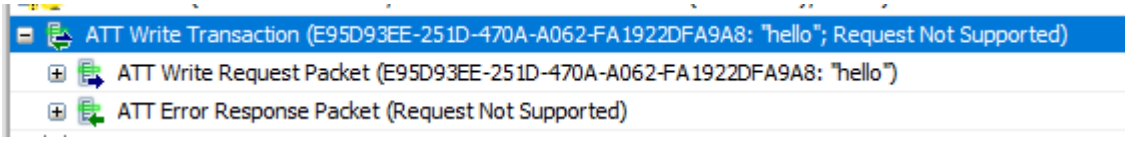


Figure 7 - request not supported

10. Permissions

As well as the properties flags described in section 9, characteristics have a set of *permissions flags*. The permissions flags are read-only and require no special permissions or conditions for them to be read by a client.

Between them, the permissions flags indicate 4 specific types of permission requirement which may or may not need to be satisfied before a client can be permitted to perform an operation on the characteristic. The four types of permission are *access*, *encryption*, *authentication* and *authorization*.

Access Permissions

This permission type defines whether or not an attribute may be read, written to, or both read and written to by a connected client.

Encryption Permissions

This permission type indicates whether or not access to the associated application must only be granted when an encrypted link is in use or, alternatively, that no encryption is required.

Authentication Permissions

This permission type indicates whether or not access to the associated attribute must only be granted when the client device was authenticated when it was paired, using an appropriate pairing association model.

Authorization Permissions

Authorization permissions indicate whether or not a client must obtain authorization before being allowed to access the attribute. But what does this mean?

This permission type allows the implementation of all manner of miscellaneous rules. For example, an authorization rule might be specified in a profile that the client may only write to the characteristic if it is estimated (using the received signal strength indicator) to be close nearby. To meet this requirement, the characteristic would have the authorization permission flag set and some code written as part of the characteristic's implementation to check the signal strength before permitting or denying the requested operation.

In this section we'll examine how BlueZ and its APIs accommodate characteristic permissions.

10.1 Access Permissions

The access permission flag provides a broad mechanism for indicating whether or not a client may acquire (read) the value of a characteristic or change it (write).

Giving a characteristic definition the 'read' property in the Flags array automatically results in the characteristic having the read access permission allocated to it by BlueZ. Assigning Flags properties such as 'write' and 'write-without-response' results in the write access permission being allocated.

So BlueZ determines the access permission flag value automatically for us, based on the operations specified as supported in the Flags property and there's nothing more for you to do in your code in this respect.

10.2 Encryption Permissions

10.2.1 Setting the encryption flag in code

Copy your server_gatt_led_svc.py file to create new file server_gatt_encryption_perm.py. Modify the LEDTextCharacteristic so that the Flags property indicates that encryption is required for permission to write to the characteristic, like this:

```
class LedTextCharacteristic(bluetooth_gatt.Characteristic):

    text = ""

    def __init__(self, bus, index, service):
        bluetooth_gatt.Characteristic.__init__(
                self, bus, index,
                bluetooth_constants.LED_TEXT_CHR_UUID,
                ['encrypt-write'],
                service)
```

Run your new script and attempt to write to the LED Text characteristic from a smartphone application. Your smartphone should indicate that it wants to pair with your Linux device. This is expected and due to the fact that the write request was rejected with an error code which means *Insufficient Encryption*.

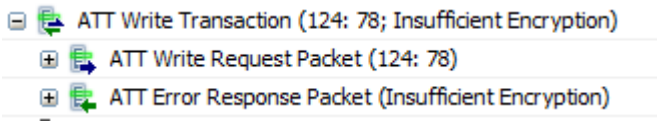


Figure 8 - An Insufficient Encryption error being returned

Don't pair yet. Abandon the procedure on your phone and we'll next look at pairing on Linux before trying the write operation again.

10.2.2 Pairing

The *encryption permission* and *authentication permission* flags both require devices to have been paired. It is only by pairing that it becomes possible for the link to be encrypted and the authentication permission is directly concerned with exactly *how* pairing was performed.

Pairing can be initiated in a number of ways. You can anticipate the need to pair and do so in advance of ever trying to act upon characteristics implemented in the server or you can attempt to act upon a characteristic which is protected by an encryption or authentication permission flag and this will trigger pairing if it has not already been done.

In most cases, devices will need something which can output or input data as part of the pairing process. Devices which have neither capability can still be paired but for our purposes, we'll assume we're working with a Linux computer like a Raspberry Pi and a smartphone. Both of these device types have both a display and support keyboard input.

A process or tool which handles interactions with the user during pairing is called an *agent*. To carry out pairing on Linux requires something to act as an agent. Fortunately most Linux desktops include

a Bluetooth configuration GUI component that acts as a default agent. Furthermore, the BlueZ source includes a Python script called simple-agent.py in the *test/* folder and this can act as an agent.

To prepare the Linux computer for pairing from the command line, the BlueZ tool *bluetoothctl* may be used. Using bluetoothctl we simply make our Linux device discoverable and ready to be paired and use the agent to provide information to the user and acquire any required input such as a passkey. The best way to understand this is to experience it so your next task will be to pair a smartphone with your Linux computer.

The simple-agent.py script takes an optional string argument (-c) which indicates the input/output capabilities we want our device to be deemed to have for pairing purposes. By default it is assumed that the device has a keyboard and a display. In fact the full set of supported capabilities argument values is documented in the BlueZ API document file agent-api.txt. The supported values are:

```
DisplayOnly
DisplayYesNo
KeyboardOnly
NoInputNoOutput
KeyboardDisplay (default)
```

Depending on the capabilities specified to the agent, the way in which we see pairing proceed may differ[1].

10.2.3 Just Works pairing

Just Works pairing is not recommended. It is vulnerable to MITM attacks, having no authentication mechanism and the entropy involved in key generation is very small so that it is also vulnerable to brute force attacks which yield the long term key used in encryption establishment. But, for the purposes of demonstration, we'll use it here just to show how it behaves.

Start simple-agent.py in a terminal with the NoInputNoOutput -c argument.

```
pi@raspberrypi:~/bluez-5.58/test $  ./simple-agent -c NoInputNoOutput
Agent registered
```

Run the command *bluetoothctl* in another terminal and run the series of commands highlighted below (

```
pi@raspberrypi:~/projects/ldsg/solutions/python/bluetooth $ bluetoothctl
Agent registered
[bluetooth]# pairable on
Changing pairable on succeeded
[CHG] Controller DC:A6:32:0E:83:E2 Pairable: yes
[bluetooth]# discoverable on
Changing discoverable on succeeded
```

Your Linux computer should now be discoverable and available for pairing.

[1] To understand how I/O capabilities are used during pairing download the Bluetooth LE Security Study Guide from https://www.bluetooth.com/bluetooth-resources/le-security-study-guide/

Run your server_gatt_encryption_perm.py script again and then use nRF Connect or similar on your smartphone to discover and connect to the Linux device. After service discovery has completed, attempt to write to the LED Text characteristic.

Pairing should be initiated. On an Android device you'll see something like Figure 9.

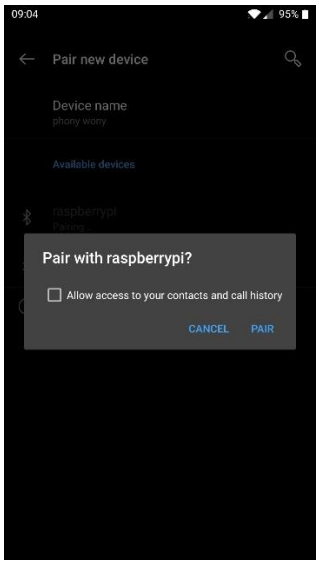


Figure 9 - Just Works pairing proceeding on an Android smartphone

When pairing has completed, the write operation should succeed and you'll see evidence in the terminal running your script.

```
pi@raspberrypi:~/projects/ldsg/solutions/python/bluetooth $ python3
server_gatt_encryption_perm.py
Registering advertisement /org/bluez/ldsg/advertisement0
Adding TemperatureService to the Application
Initialising TemperatureService object
Adding TemperatureCharacteristic to the service
creating Characteristic with path=/org/bluez/ldsg/service0/char0
Initial temperature set to 31
Initialising LedService object
Adding LedTextharacteristic to the service
creating Characteristic with path=/org/bluez/ldsg/service1/char0
Registering GATT application...
GetManagedObjects
GetManagedObjects: service=/org/bluez/ldsg/service0
GetManagedObjects: service=/org/bluez/ldsg/service1
GATT application registered
{'Type': 'peripheral', 'LocalName': dbus.String('Hello'), 'Discoverable':
dbus.Boolean(True)}
Advertisement registered OK
connected
Unregistering advertisement /org/bluez/ldsg/advertisement0
[104, 101, 108, 108, 111] = hello
disconnected
```

10.2.4 Passkey Pairing

Passkey pairing involves a random number being displayed on both devices and the user being asked to confirm that they are the same number. This authentication mechanism provides protection against MITM attacks.

Unpair your devices by using the appropriate feature of your Central device (e.g. the Forget function on an Android device in the Bluetooth section of the settings application) and the *remove* command in bluetoothctl on the Linux device.

```
[bluetooth]# paired-devices
Device D8:0B:9A:97:F1:BB Galaxy S10+
[bluetooth]# remove D8:0B:9A:97:F1:BB
[DEL] Characteristic (Handle 0x0010)
        /org/bluez/hci0/dev_6C_96_62_A2_7D_2B/service0001/char0002
        00002a05-0000-1000-8000-00805f9b34fb
        Service Changed
[DEL] Primary Service (Handle 0xed18)
        /org/bluez/hci0/dev_6C_96_62_A2_7D_2B/service0001
        00001801-0000-1000-8000-00805f9b34fb
        Generic Attribute Profile
[DEL] Device D8:0B:9A:97:F1:BB Galaxy S10+
Device has been removed
[bluetooth]# paired-devices
[bluetooth]#
```

Restart the agent script but specify the KeyboardDisplay capabilities argument.

```
pi@raspberrypi:~/bluez-5.58/test $ ./simple-agent --capability KeyboardDisplay
Agent registered
RequestConfirmation (/org/bluez/hci0/dev_94_65_2D_A8_F9_19, 427197)
Confirm passkey (yes/no): yes
```

Make your device ready for pairing using bluetoothctl.

```
pi@raspberrypi:~/projects/ldsg/solutions/python/bluetooth $ bluetoothctl
Agent registered
[bluetooth]# pairable on
Changing pairable on succeeded
[bluetooth]# discoverable on
Changing discoverable on succeeded
```

Run your Python script again and then use your smartphone app to discover your Linux device, connect to it and attempt to write to the LED Text characteristic. Pairing will be initiated again due to the encryption permission flag on the characteristic. Complete pairing and note the difference in the procedure compared with Just Works pairing. After pairing, the write operation should succeed.

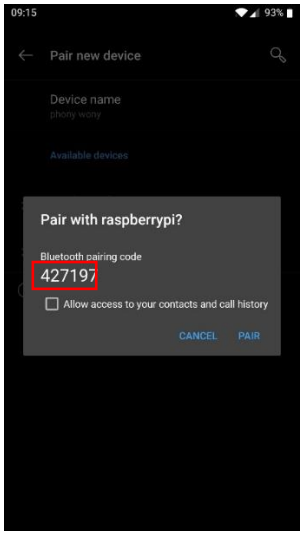


Figure 10 - Passkey pairing on an Android smartphone

10.3 Authentication Permissions

As described in section 10.2, there are various ways in which pairing can proceed depending in part on the input and output capabilities of the two devices[2]. Some methods provide protection against man in the middle (MITM) attacks using various authentication techniques and some do not. The authentication permission flag, when set for a characteristic indicates that not only must the link be encrypted before access to the characteristic is allowed but that when pairing was originally performed, it used a method which included an authentication step and therefore provided MITM protection.

Copy your server_gatt_encryption_perm.py script to create new script server_gatt_authentication_perm.py. Modify the LED Text characteristic so that instead of requiring only encryption, it requires encryption and to have been paired using an authenticating method before writing to the characteristic is permitted.

```
class LedTextCharacteristic(bluetooth_gatt.Characteristic):

    text = ""

    def __init__(self, bus, index, service):
        bluetooth_gatt.Characteristic.__init__(
                self, bus, index,
                bluetooth_constants.LED_TEXT_CHR_UUID,
                ['write', 'encrypt-authenticated-write'],
                service)
```

Clear any previous pairings from your devices and run simple-agent with argument -c NoInputNoOutput which as you saw in 10.2.3 will result in the unauthenticated Just Works pairing method being used when pairing takes place.

Run your server_gatt_authentication_perm.py script. Use a smartphone to attempt to connect to the device and write to the LED Text characteristic. Pairing should take be automatically triggered and require no other user interaction other than to allow this to happen. Disconnect your phone and scan for and then connect to the Linux device again. Attempt to write to the LED Text characteristic. Depending on the operating system, you will either find that the request is ignored by the smartphone or it will proceed but fail. If you have a protocol analyser or sniffer, you can monitor the interaction and will see that the attempt to write to the characteristic failed with an Insufficient Authentication error.

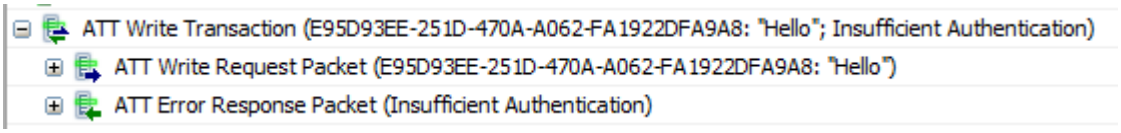


Figure 11 - Failed write attempt due to a non-authenticating pairing method having been used

10.4 Authorization Permissions

The authorization flag allows arbitrary authorization logic to be associated with a characteristic. Copy your server_gatt_authentication_perm.py script to create new script

[2] For more information on Bluetooth Low Energy security, download the Bluetooth LE Security Study Guide from https://www.bluetooth.com/bluetooth-resources/le-security-study-guide/

server_gatt_authorization_perm.py. Modify the LED Text characteristic so that writing to it requires authorization.

```
class LedTextCharacteristic(bluetooth_gatt.Characteristic):

    text = ""

    def __init__(self, bus, index, service):
        bluetooth_gatt.Characteristic.__init__(
                self, bus, index,
                bluetooth_constants.LED_TEXT_CHR_UUID,
                ['write', 'authorize'],
                service)
```

Next, we'll implement the authorization rule that only specific string values may be written to the LED Text characteristic. Add a list of approved strings to your script like this:

```
approved_strings = ['hello' , 'goodbye', 'cheese']
```

Add an authorization check function to the characteristic and modify the WriteValue method to call it.

```
  def authorized(self, text):
    global approved_strings
    if text in approved_strings:
      return True
    else:
      return False

  def WriteValue(self, value, options):
    ascii_bytes = bluetooth_utils.dbus_to_python(value)
    ascii = ''.join(chr(i) for i in ascii_bytes)
    print(str(ascii_bytes) + " = " + ascii)
    if self.authorized(ascii):
      print("authorized")
      text = ascii
    else:
      print("Not authorized")
      raise bluetooth_exceptions.NotAuthorizedException()
```

Test your new script by attempting to write a series of different strings to the LED Text characteristic.

```
pi@raspberrypi:~/projects/ldsg/solutions/python/bluetooth $ python3
server_gatt_authorization_perm.py
Registering advertisement /org/bluez/ldsg/advertisement0
Adding TemperatureService to the Application
Initialising TemperatureService object
Adding TemperatureCharacteristic to the service
creating Characteristic with path=/org/bluez/ldsg/service0/char0
Initial temperature set to 25
Initialising LedService object
Adding LedTextharacteristic to the service
creating Characteristic with path=/org/bluez/ldsg/service1/char0
Registering GATT application...
GetManagedObjects
GetManagedObjects: service=/org/bluez/ldsg/service0
GetManagedObjects: service=/org/bluez/ldsg/service1
GATT application registered
```

```
{'Type': 'peripheral', 'LocalName': dbus.String('Hello'), 'Discoverable':
dbus.Boolean(True)}
Advertisement registered OK
connected
Unregistering advertisement /org/bluez/ldsg/advertisement0
5B:44:C5:06:ED:87
disconnected
Registering advertisement /org/bluez/ldsg/advertisement0
{'Type': 'peripheral', 'LocalName': dbus.String('Hello'), 'Discoverable':
dbus.Boolean(True)}
Advertisement registered OK
connected
Unregistering advertisement /org/bluez/ldsg/advertisement0
[98, 97, 110, 97, 110, 97] = banana
Not authorized
```

If you have a sniffer or protocol analyzer you will be able to see write requests being rejected with Insufficient Authorization errors when you attempt to write a string that is not a member of your approved_strings list.

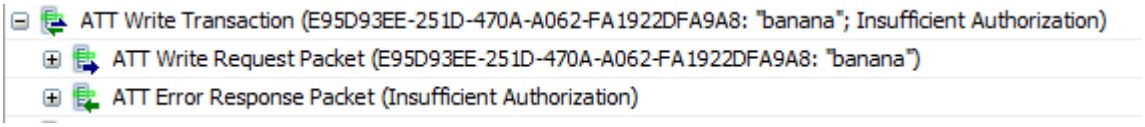


Figure 12 - insufficient authorization

11. 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. You've also explored the flags which control what operations a characteristic supports and the permissions which may be set to control the use of those operations by clients.

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 Peripheral code using BlueZ.

Bluetooth for Linux Developers Study Guide

Installation and Configuration

Release : 1.0.1

Document Version: 1.0.0

Last updated : 16th November 2021

Contents

1. REVISION HISTORY 3

2. INTRODUCTION 4

3. BLUEZ INSTALLATION 4

3.1 Kernel Configuration 4

3.2 Install BlueZ 5.58 or later and dependencies 5

3.3 Enable Bluetooth DBus Communication 6

3.4 Start the appropriate Bluetooth daemon 7

3.4.1 GAP/GATT support 7

3.4.2 Bluetooth mesh support 7

3.5 Debugging 7

3.5.1 -nd flag 7

3.5.2 btmon 9

3.5.3 btmgmt 10

4. PYTHON 10

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

Follow the instructions in this module to install and configure BlueZ for either GAP/GATT applications or for Bluetooth mesh. Bluetooth mesh uses cryptography functions such as AES-CMAC which must be supported by the Linux kernel.

A number of libraries must be installed and if your machine is to support Bluetooth mesh, the Linux Kernel must be compiled with the required cryptographic functions included.

Instructions for setting up Python3 for D-Bus development are also provided.

3. BlueZ Installation

3.1 Kernel Configuration

Note: If you are intending to develop Bluetooth mesh applications, the Linux kernel needs to be compiled with particular features enabled. A standard kernel as provided by a standard Raspbian image is suitable for using BlueZ with GAP/GATT applications. The steps in this section are therefore optional and only need to be carried out if developing for Bluetooth mesh.

1. Get your package lists up to date and then install dependencies.

```
sudo apt-get update

sudo apt-get install -y git bc libusb-dev libdbus-1-dev libglib2.0-dev libudev-dev libical-dev libreadline-dev autoconf bison flex libssl-dev libncurses-dev glib2.0 libdbus-1-dev
```

2. Download the Linux kernel source and install its dependencies

```
cd ~

wget https://github.com/raspberrypi/linux/archive/raspberrypi-kernel_1.20210303-1.tar.gz

tar -xvf raspberrypi-kernel_1.20210303-1.tar.gz

cd ./linux-raspberrypi-kernel_1.20210303-1

sudo apt install bc bison flex libssl-dev make
```

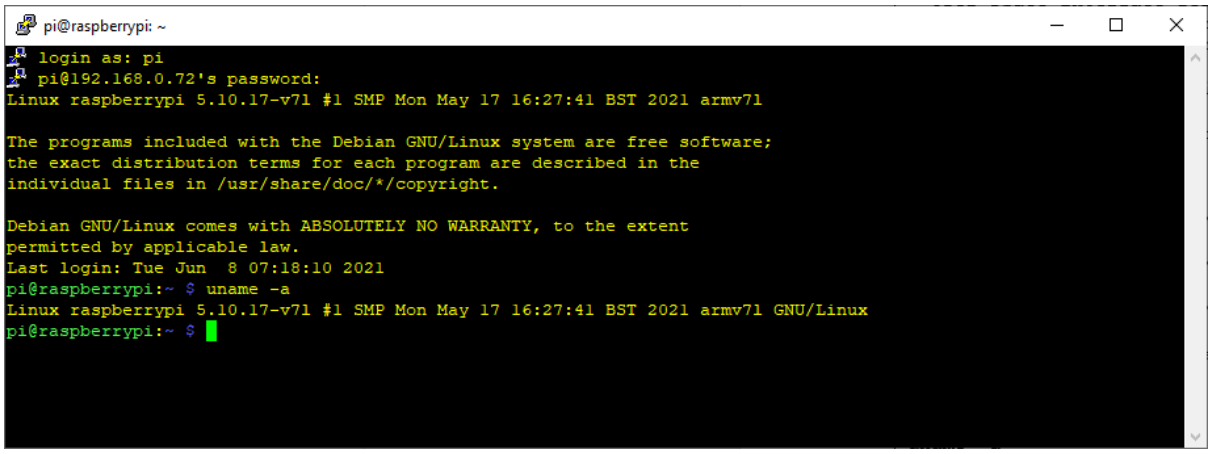
3. Configure, build and the install the recompiled kernel

```
# If you're using a Pi Zero:
KERNEL=kernel
make bcmrpi_defconfig

# If you're using a Pi 4 or Pi 400:
KERNEL=kernel71
make bcm2711_defconfig

make menuconfig

# In the menus that appear, select the following options

Cryptographic API-->
* CCM Support
* CMAC Support
* User-space interface for hash algorithms
```

```
* User-space interface for symmetric key cipher algorithms
* User-space interface for AEAD cipher algorithms

# Save to .config and exit menuconfig

make -j4 zImage modules dtbs

sudo make modules_install

sudo cp arch/arm/boot/dts/*.dtb /boot/

sudo cp arch/arm/boot/dts/overlays/*.dtb* /boot/overlays/

sudo cp arch/arm/boot/dts/overlays/README /boot/overlays/

sudo cp arch/arm/boot/zImage /boot/$KERNEL.img

sudo reboot
```

4. After rebooting, check the kernel version

```
uname -a
```



3.2 Install BlueZ 5.58 or later and dependencies

Check the version of BlueZ running on your machine by launching the *bluetooth* tool and entering the *version* command.

```
pi@raspberrypi:~ $ bluetoothctl
Agent registered
[bluetooth]# version
Version 5.50
[bluetooth]#
```

The version reported should be at least version 5.58. If an earlier version is reported, install dependencies and then download and build the required version of BlueZ. Note that this resource was created and tested using version 5.58. Later versions may be OK but this is not guaranteed.

Exit bluetoothctl by entering the *quit* command.

1. Install libraries

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

2. Install JSON library

```
sudo apt install cmake

wget https://s3.amazonaws.com/json-c_releases/releases/json-c-0.15.tar.gz
tar xvf json-c-0.15.tar.gz

cd ~/json-c-0.15
mkdir build
cd build

cmake -DCMAKE_INSTALL_PREFIX=/usr -DCMAKE_BUILD_TYPE=Release -DBUILD_STATIC_LIBS=OFF  ..

make

sudo make install
```

3. Install the Embedded Linux Library (ELL)

```
cd ~
wget https://mirrors.edge.kernel.org/pub/linux/libs/ell/ell-0.6.tar.xz
tar -xvf ell-0.6.tar.xz
cd ell-0.6/
sudo ./configure --prefix=/usr
sudo make
sudo make install
```

You can probably ignore any warnings.

4. Install BlueZ

```
cd ~

pi@raspberrypi:~ $ wget http://www.kernel.org/pub/linux/bluetooth/bluez-5.58.tar.xz
URL transformed to HTTPS due to an HSTS policy
--2021-04-08 12:28:05--  https://www.kernel.org/pub/linux/bluetooth/bluez-5.58.tar.xz
Resolving www.kernel.org (www.kernel.org)... 136.144.49.103, 2604:1380:40b0:1a00::1
Connecting to www.kernel.org (www.kernel.org)|136.144.49.103|:443... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://mirrors.edge.kernel.org/pub/linux/bluetooth/bluez-5.58.tar.xz [following]
--2021-04-08 12:28:06--  https://mirrors.edge.kernel.org/pub/linux/bluetooth/bluez-
5.58.tar.xz
Resolving mirrors.edge.kernel.org (mirrors.edge.kernel.org)... 147.75.101.1,
2604:1380:2001:3900::1
Connecting to mirrors.edge.kernel.org (mirrors.edge.kernel.org)|147.75.101.1|:443...
connected.
HTTP request sent, awaiting response... 200 OK
Length: 2060368 (2.0M) [application/x-xz]
Saving to: ‘bluez-5.58.tar.xz’

bluez-5.58.tar.xz
100%[=======================================================================================
===========>]   1.96M  6.12MB/s    in 0.3s

2021-04-08 12:28:06 (6.12 MB/s) - ‘bluez-5.58.tar.xz’ saved [2060368/2060368]

tar -xvf bluez-5.58.tar.xz
cd bluez-5.58/
./configure --enable-mesh --enable-testing --enable-tools --prefix=/usr --
mandir=/usr/share/man --sysconfdir=/etc --localstatedir=/var
sudo make
sudo make install
```

3.3 Enable Bluetooth DBus Communication

The Bluetooth API uses the Linux DBus inter-process communication system. A default security policy restricts use of DBus for Bluetooth purposes to processes owned by users that are a member

of the *bluetooth* system group. You can see this in the /etc/dbus-1/system.d/bluetooth.conf file, here:

```
  <!-- allow users of bluetooth group to communicate -->
  <policy group="bluetooth">
    <allow send_destination="org.bluez"/>
  </policy>
```

Edit /etc/group and add the *www-data* and *pi* users to the bluetooth group.

```
pi@raspberrypi:~ $ sudo vi /etc/group
pi@raspberrypi:~ $ cat /etc/group|grep bluet
bluetooth:x:112:www-data,pi
# you need to start a new bash shell for the change to be activated - do this by entering
# su - pi
# or exit this shell and start a new one

pi@raspberrypi:~ $ groups
pi adm dialout cdrom sudo audio www-data video plugdev games users input netdev bluetooth
gpio i2c spi
```

Now restart the DBus daemon:

```
pi@raspberrypi:~ $ sudo service dbus restart
```

3.4 Start the appropriate Bluetooth daemon

3.4.1 GAP/GATT support

For general GAP/GATT support, your device must run the *bluetooth* daemon. It can be started and stopped and its status checked using the *service* command.

```
sudo service bluetooth start
sudo service bluetooth status
sudo service bluetooth stop
```

The bluetooth-mesh daemon must not be running at the same time as the bluetooth daemon.

3.4.2 Bluetooth mesh support

For Bluetooth mesh support, your device must run the bluetooth-mesh daemon. It can be started and stopped and its status checked using the *service* command.

```
sudo service bluetooth-mesh start
sudo service bluetooth-mesh status
sudo service bluetooth-mesh stop
```

The bluetooth daemon must not be running at the same time as the bluetooth-mesh daemon.

3.5 Debugging

3.5.1 -nd flag

If you run into problems, it's possible to run the bluetooth and bluetooth-mesh daemons in debug mode (but not both at the same time). This is done by editing the associated Linux service file and adding the -nd flag to the end of the ExecStart line. Here's an example for the bluetooth daemon.

Note that the service file for the Bluetooth mesh daemon is in the same location and named bluetooth-mesh.service.

```
pi@raspberrypi:~/projects/ldsg/solutions/python/bluetooth $ sudo cat
/lib/systemd/system/bluetooth.service
[Unit]
Description=Bluetooth service
Documentation=man:bluetoothd(8)
ConditionPathIsDirectory=/sys/class/bluetooth

[Service]
Type=dbus
BusName=org.bluez
ExecStart=/usr/libexec/bluetooth/bluetoothd -nd
NotifyAccess=main
#WatchdogSec=10
#Restart=on-failure
CapabilityBoundingSet=CAP_NET_ADMIN CAP_NET_BIND_SERVICE
LimitNPROC=1
ProtectHome=true
ProtectSystem=full

[Install]
WantedBy=bluetooth.target
Alias=dbus-org.bluez.service
```

After adding the -nd flag you must run:

```
sudo systemctl daemon-reload
```

Debug output will be written to syslog which can be monitored with tail:

```
tail -f /var/log/syslog
```

```
Dec 14 08:05:29 raspberrypi bluetoothd[820]: bluetoothd[820]: src/agent.c:agent_ref()
0x6dc0f8: ref=1
Dec 14 08:05:29 raspberrypi bluetoothd[820]: bluetoothd[820]: src/agent.c:register_agent()
agent :1.26
Dec 14 08:05:29 raspberrypi bluetoothd[820]: src/agent.c:agent_ref() 0x6dc0f8: ref=1
Dec 14 08:05:29 raspberrypi bluetoothd[820]: src/agent.c:register_agent() agent :1.26
Dec 14 08:05:34 raspberrypi bluetoothd[820]: bluetoothd[820]:
src/adapter.c:start_discovery() sender :1.26
Dec 14 08:05:34 raspberrypi bluetoothd[820]: bluetoothd[820]:
src/adapter.c:update_discovery_filter()
Dec 14 08:05:34 raspberrypi bluetoothd[820]: src/adapter.c:start_discovery() sender :1.26
Dec 14 08:05:34 raspberrypi bluetoothd[820]: bluetoothd[820]:
src/adapter.c:discovery_filter_to_mgmt_cp()
Dec 14 08:05:34 raspberrypi bluetoothd[820]: bluetoothd[820]:
src/adapter.c:trigger_start_discovery()
Dec 14 08:05:34 raspberrypi bluetoothd[820]: bluetoothd[820]:
src/adapter.c:cancel_passive_scanning()
Dec 14 08:05:34 raspberrypi bluetoothd[820]: bluetoothd[820]:
src/adapter.c:start_discovery_timeout()
Dec 14 08:05:34 raspberrypi bluetoothd[820]: bluetoothd[820]:
src/adapter.c:start_discovery_timeout() adapter->current_discovery_filter == 0
Dec 14 08:05:34 raspberrypi bluetoothd[820]: bluetoothd[820]:
src/adapter.c:start_discovery_complete() status 0x00
Dec 14 08:05:34 raspberrypi bluetoothd[820]: bluetoothd[820]:
src/adapter.c:discovering_callback() hci0 type 7 discovering 1 method 0
Dec 14 08:05:34 raspberrypi bluetoothd[820]: src/adapter.c:update_discovery_filter()
Dec 14 08:05:34 raspberrypi bluetoothd[820]: src/adapter.c:discovery_filter_to_mgmt_cp()
Dec 14 08:05:34 raspberrypi bluetoothd[820]: src/adapter.c:trigger_start_discovery()
Dec 14 08:05:34 raspberrypi bluetoothd[820]: src/adapter.c:cancel_passive_scanning()
Dec 14 08:05:34 raspberrypi bluetoothd[820]: src/adapter.c:start_discovery_timeout()
Dec 14 08:05:34 raspberrypi bluetoothd[820]: src/adapter.c:start_discovery_timeout()
adapter->current_discovery_filter == 0
Dec 14 08:05:34 raspberrypi bluetoothd[820]: src/adapter.c:start_discovery_complete()
status 0x00
Dec 14 08:05:34 raspberrypi bluetoothd[820]: src/adapter.c:discovering_callback() hci0 type
7 discovering 1 method 0
```

3.5.2 btmon

You can also sometimes get clues using the *btmon* tool:

```
pi@raspberrypi:~ $ sudo btmon
Bluetooth monitor ver 5.58
= Note: Linux version 5.4.79-v7l+ (armv7l)
0.097785
= Note: Bluetooth subsystem version
2.22
0.097793
= New Index: DC:A6:32:0E:83:E2
(Primary,UART,hci0)
[hci0] 0.097796
= Open Index: DC:A6:32:0E:83:E2
[hci0] 0.097801
= Index Info: DC:A6:32:0E:83:E2 (Cypress Semiconductor)
[hci0] 0.097804
@ MGMT Open: bluetoothd (privileged) version 1.14
{0x0001} 0.097808
@ MGMT Command: Add Advertising (0x003e) plen 11
{0x0001} [hci0] 6.161926
        Instance: 1
        Flags: 0x00000000
        Duration: 0
        Timeout: 0
        Advertising data length: 0
        Scan response length: 0
< HCI Command: LE Set Advertising Data (0x08|0x0008) plen 32
#1 [hci0] 6.161979
        Length: 0
> HCI Event: Command Complete (0x0e) plen 4
#2 [hci0] 6.162365
      LE Set Advertising Data (0x08|0x0008) ncmd 1
        Status: Success (0x00)
< HCI Command: LE Set Scan Response Data (0x08|0x0009) plen 32
#3 [hci0] 6.162394
        Length: 0
> HCI Event: Command Complete (0x0e) plen 4
#4 [hci0] 6.162927
      LE Set Scan Response Data (0x08|0x0009) ncmd 1
        Status: Success (0x00)
< HCI Command: LE Set Random Address (0x08|0x0005) plen 6
#5 [hci0] 6.162953
        Address: 0C:2B:9E:27:57:51 (Non-Resolvable)
> HCI Event: Command Complete (0x0e) plen 4
#6 [hci0] 6.163259
      LE Set Random Address (0x08|0x0005) ncmd 1
        Status: Success (0x00)
< HCI Command: LE Set Advertising Parameters (0x08|0x0006) plen 15
#7 [hci0] 6.163287
        Min advertising interval: 62.500 msec (0x0064)
        Max advertising interval: 93.750 msec (0x0096)
        Type: Non connectable undirected - ADV_NONCONN_IND (0x03)
        Own address type: Random (0x01)
        Direct address type: Public (0x00)
        Direct address: 00:00:00:00:00:00 (OUI 00-00-00)
        Channel map: 37, 38, 39 (0x07)
        Filter policy: Allow Scan Request from Any, Allow Connect Request from Any (0x00)
> HCI Event: Command Complete (0x0e) plen 4
#8 [hci0] 6.163624
      LE Set Advertising Parameters (0x08|0x0006) ncmd 1
        Status: Invalid HCI Command Parameters (0x12)
@ MGMT Event: Command Status (0x0002) plen 3
{0x0001} [hci0] 6.163654
      Add Advertising (0x003e)
        Status: Invalid Parameters (0x0d)
= bluetoothd: src/advertising.c:add_client_complete() Failed to add advertisement: Invalid
Parameters (0x0d)
```

3.5.3 btmgmt

The btmgmt tool is another useful source of information:

```
pi@raspberrypi:~ $ sudo btmgmt
[mgmt]# info
Index list with 1 item
hci0:   Primary controller
        addr DC:A6:32:0E:83:E2 version 9 manufacturer 305 class 0x000000
        supported settings: powered connectable fast-connectable discoverable bondable
link-security ssp br/edr le advertising secure-conn debug-keys privacy static-addr
        current settings: powered bondable ssp br/edr le secure-conn
        name raspberrypi
        short name
```

4. Python

All exercises and examples in this study guide use Python3. Ensure your machine has python3 installed like this:

```
pi@raspberrypi:~/projects/dbus/bluetooth_samples/c $ python3 -V
Python 3.7.3
```

If you don't already have python3 installed, install it as follows:

```
sudo apt update
sudo apt install python3
```

You will also need an *mainloop* implementation installed so that asynchronous messaging is supported in your applications. To that end, install a GLib library as follows:

```
pi@raspberrypi:~/projects/ldsg/solutions/python $ sudo apt-get install python3-gi
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  gir1.2-glib-2.0
The following NEW packages will be installed:
  gir1.2-glib-2.0 python3-gi
0 upgraded, 2 newly installed, 0 to remove and 228 not upgraded.
Need to get 0 B/304 kB of archives.
After this operation, 1,546 kB of additional disk space will be used.
Do you want to continue? [Y/n] y
Selecting previously unselected package gir1.2-glib-2.0:armhf.
(Reading database ... 96669 files and directories currently installed.)
Preparing to unpack .../gir1.2-glib-2.0_1.58.3-2_armhf.deb ...
Unpacking gir1.2-glib-2.0:armhf (1.58.3-2) ...
Selecting previously unselected package python3-gi.
Preparing to unpack .../python3-gi_3.30.4-1_armhf.deb ...
Unpacking python3-gi (3.30.4-1) ...
Setting up gir1.2-glib-2.0:armhf (1.58.3-2) ...
Setting up python3-gi (3.30.4-1) ...
```

Create a directory in which to create your own scripts. Copy into this folder the Python files packaged in the code/solutions directory of the study guide and whose names start with *bluetooth_*. Your project directory should look like this:

```
pi@raspberrypi:~/projects/ldsg/solutions/python/bluetooth $ ls -l bluetooth_*
-rwxrwx--- 1 pi 2887 Nov  4 14:05 bluetooth_constants.py
-rwxrwx--- 1 pi  778 Nov  9 10:54 bluetooth_exceptions.py
-rwxrwx--- 1 pi 6930 Nov  4 14:26 bluetooth_gatt.py
```

```
-rwxrwx--- 1 pi 2072 Nov  4 10:32 bluetooth_utils.py
```