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

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.

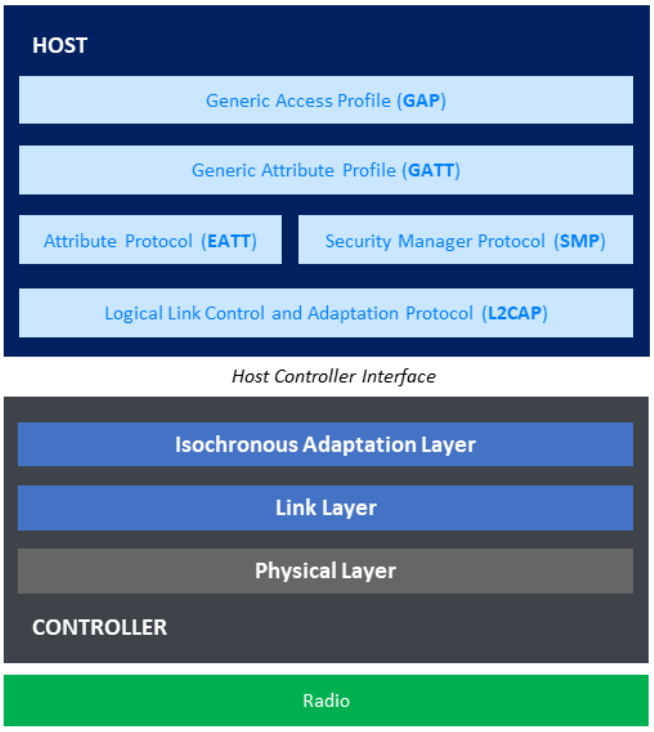

HOST
Generic Access Profile (GAP)
Generic Attribute Profile (GATT)
Attribute Protocol (EATT)
Security Manager Protocol (SMP)
Logical Link Control and Adaptation Protocol (L2CAP)
Host Controller Interface
Isochronous Adaptation Layer
Link Layer
Physical Layer
CONTROLLER
Radio

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*.

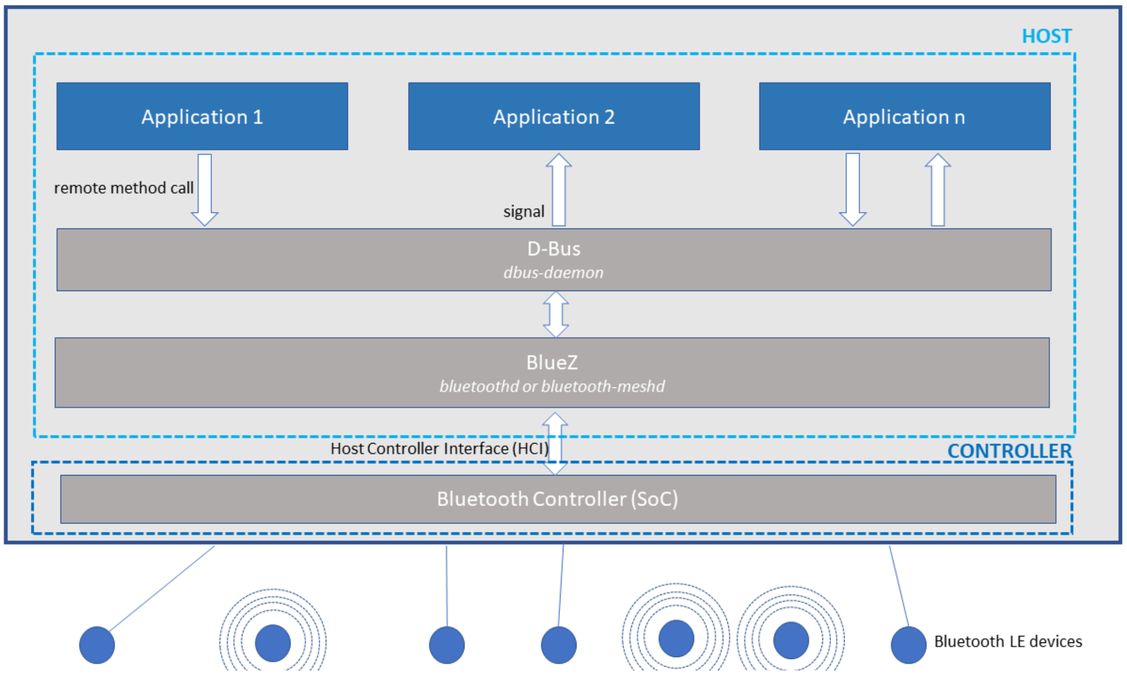


Figure 2 - Architecture

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.

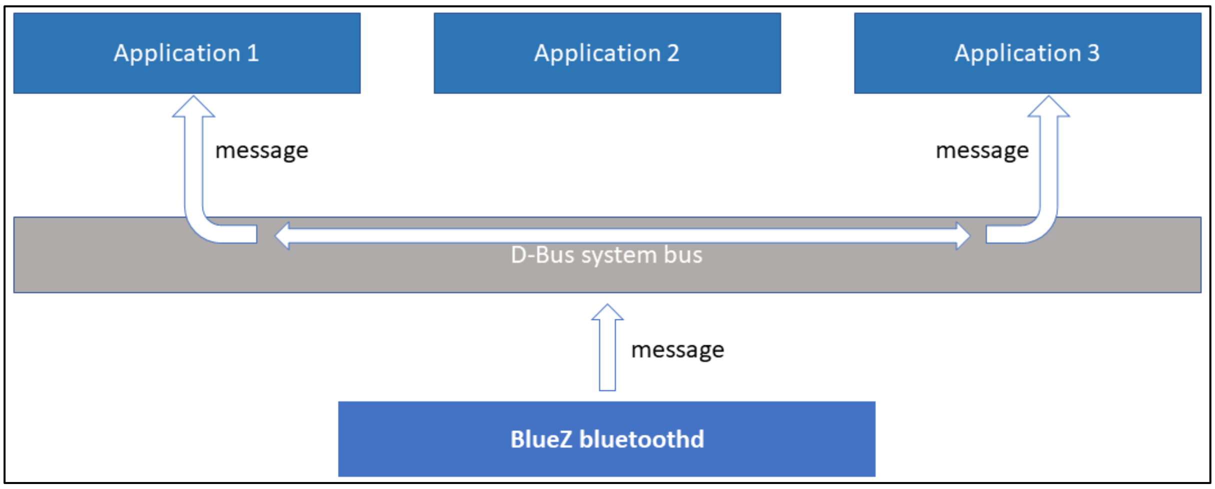


Figure 1 - messages delivered using the D-Bus system bus

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*.

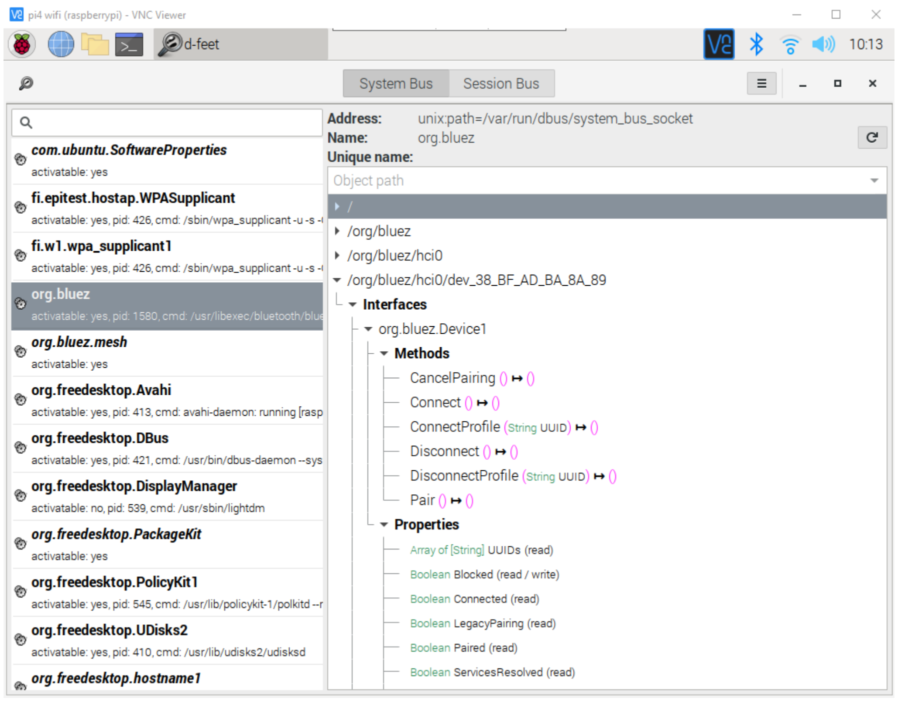
4.2.1 D-feet

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
```



com.ubuntu.SoftwareProperties
activatable: yes
fi.epitest.hostap.WPASupplicant
activatable: yes, pid: 426, cmd: /sbin/wpa_supplicant -u -s -
fi.w1.wpa_supplicant1
activatable: yes, pid: 426, cmd: /sbin/wpa_supplicant -u -s -
org.bluez
activatable: yes, pid: 1580, cmd: /usr/libexec/bluetooth/blue
org.bluez.mesh
activatable: yes
org.freedesktop.Avahi
activatable: yes, pid: 413, cmd: avahi-daemon: running [rasp
org.freedesktop.DBus
activatable: yes, pid: 421, cmd: /usr/bin/dbus-daemon --sys
org.freedesktop.DisplayManager
activatable: no, pid: 539, cmd: /usr/sbin/lightdm
org.freedesktop.PackageKit
activatable: yes
org.freedesktop.PolicyKit1
activatable: yes, pid: 545, cmd: /usr/lib/policykit-1/polkitd --r
org.freedesktop.UDisks2
activatable: yes, pid: 410, cmd: /usr/lib/udisks2/udisksd
org.freedesktop.hostname1

Address: unix:path=/var/run/dbus/system_bus_socket
Name: org.bluez
Unique name:

Object path

```
/
/org/bluez
/org/bluez/hci0
/org/bluez/hci0/dev_38_BF_AD_BA_8A_89
  Interfaces
    org.bluez.Device1
      Methods
        CancelPairing () ↦ ()
        Connect () ↦ ()
        ConnectProfile (String UUID) ↦ ()
        Disconnect () ↦ ()
        DisconnectProfile (String UUID) ↦ ()
        Pair () ↦ ()
      Properties
        Array of [String] UUIDs (read)
        Boolean Blocked (read / write)
        Boolean Connected (read)
        Boolean LegacyPairing (read)
        Boolean Paired (read)
        Boolean ServicesResolved (read)
```

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.

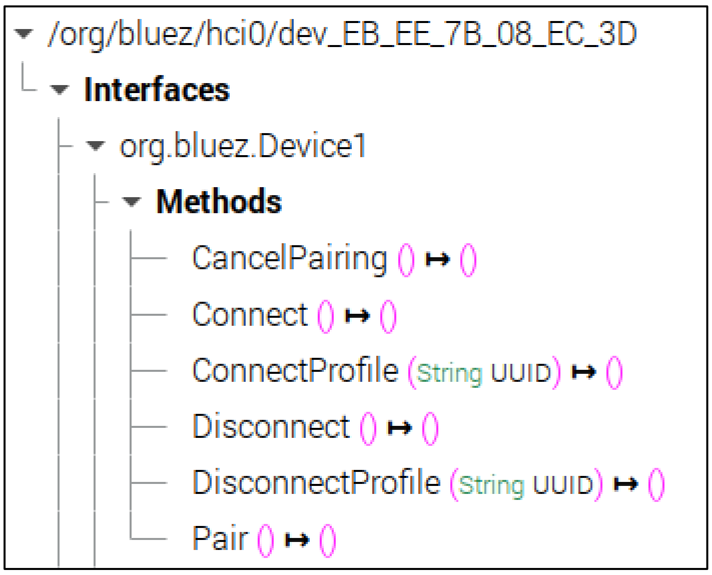

```
/org/bluez/hci0/dev_EB_EE_7B_08_EC_3D
  Interfaces
    org.bluez.Device1
      Methods
        CancelPairing () ↦ ()
        Connect () ↦ ()
        ConnectProfile (String UUID) ↦ ()
        Disconnect () ↦ ()
        DisconnectProfile (String UUID) ↦ ()
        Pair () ↦ ()
```

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.