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

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*.

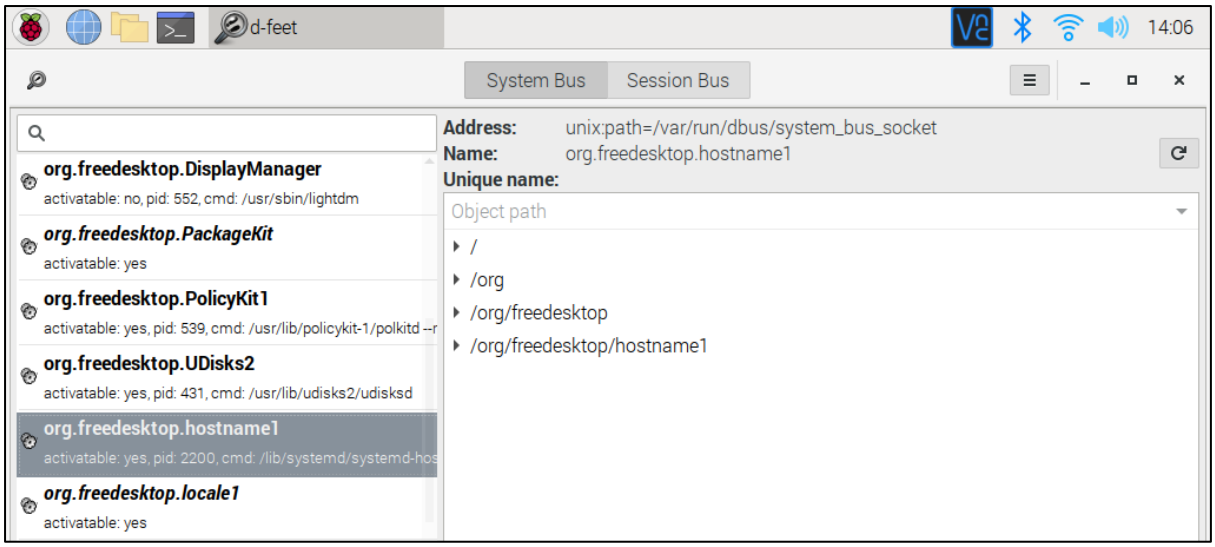


Figure 1 - org.freedesktop.hostname1

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

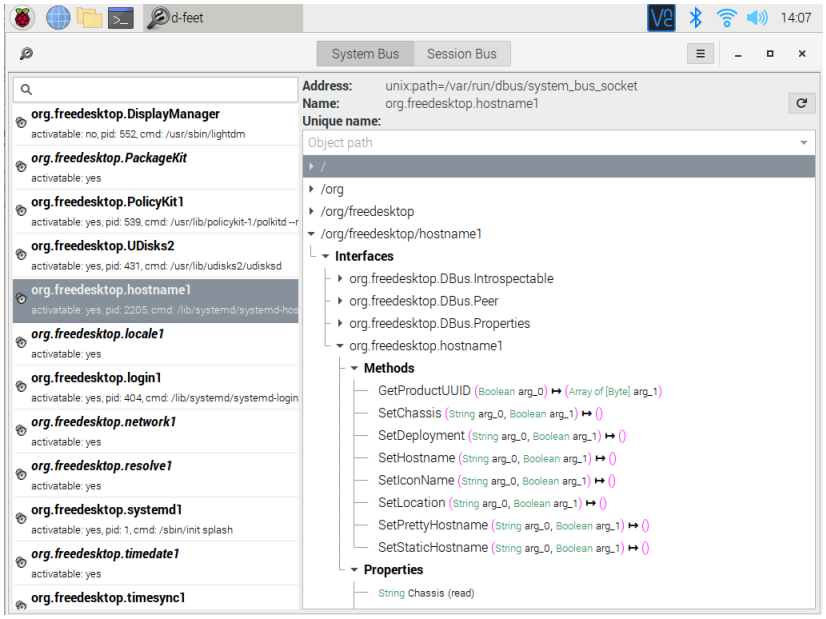


Figure 2 - org/freedesktop/hostname1 - Methods

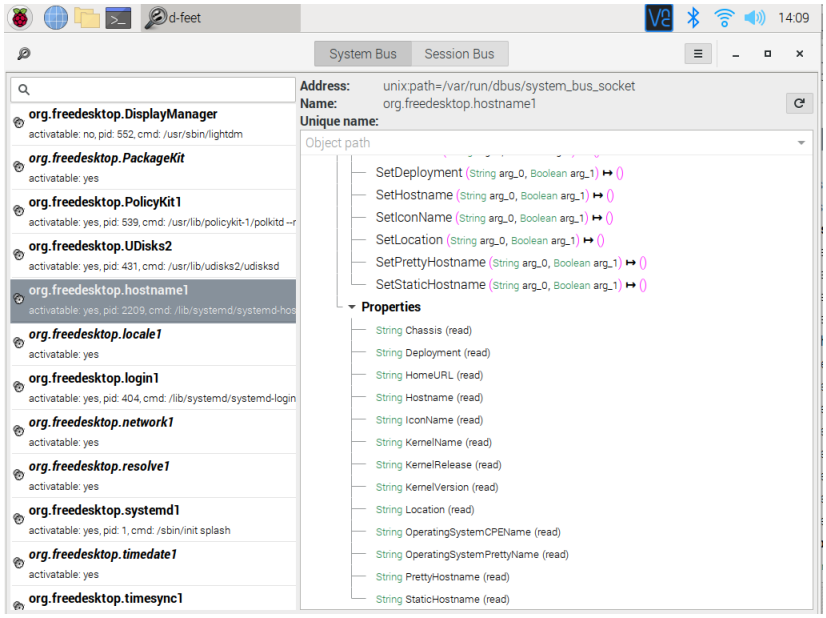


Figure 3 - org/freedesktop/hostname1 - Properties

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:

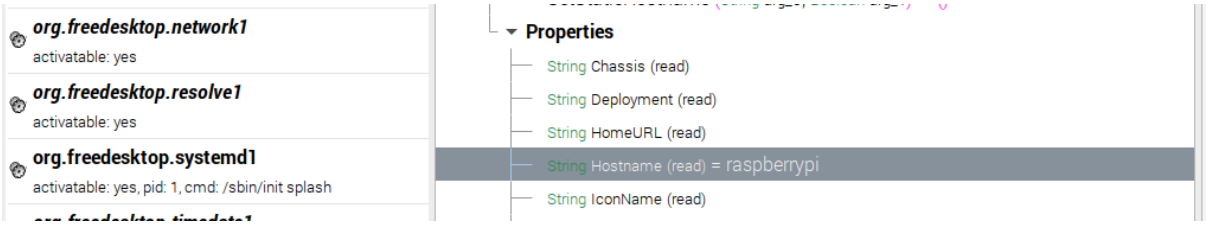


Figure 4 - D-Feet and the Hostname property

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.

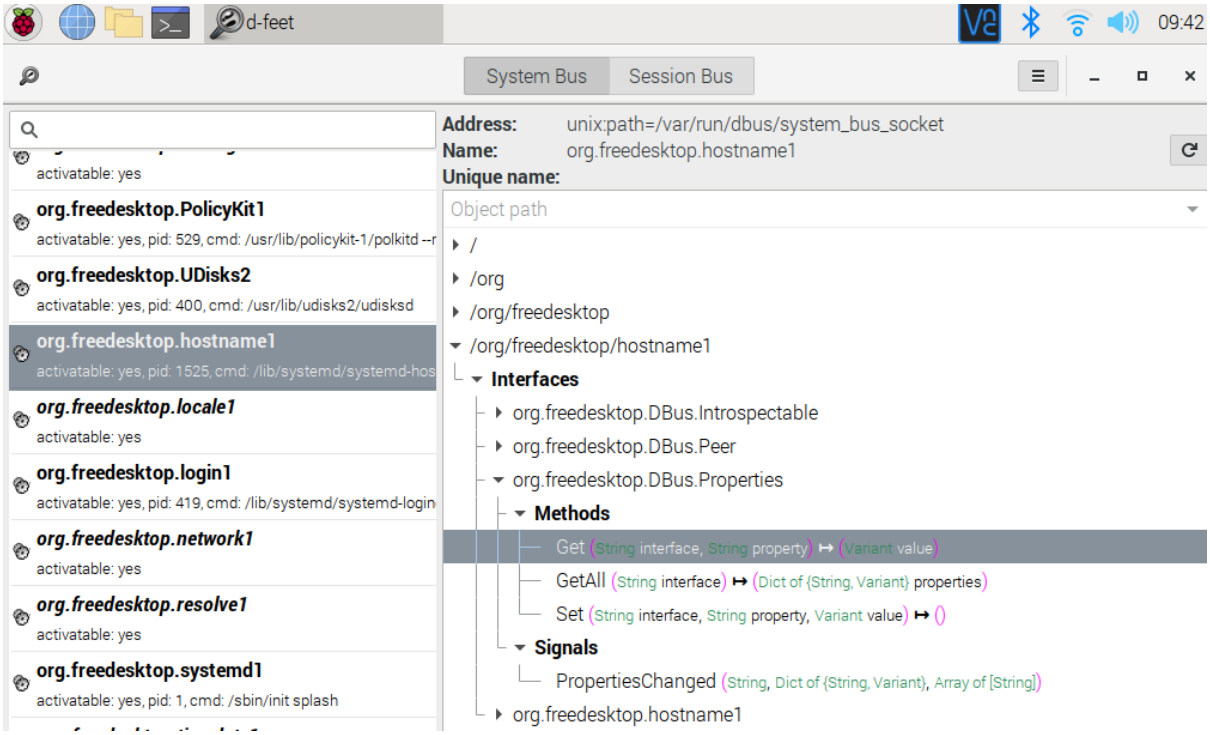


Figure 5 - The org.freedesktop.DBus.Properties interface

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.

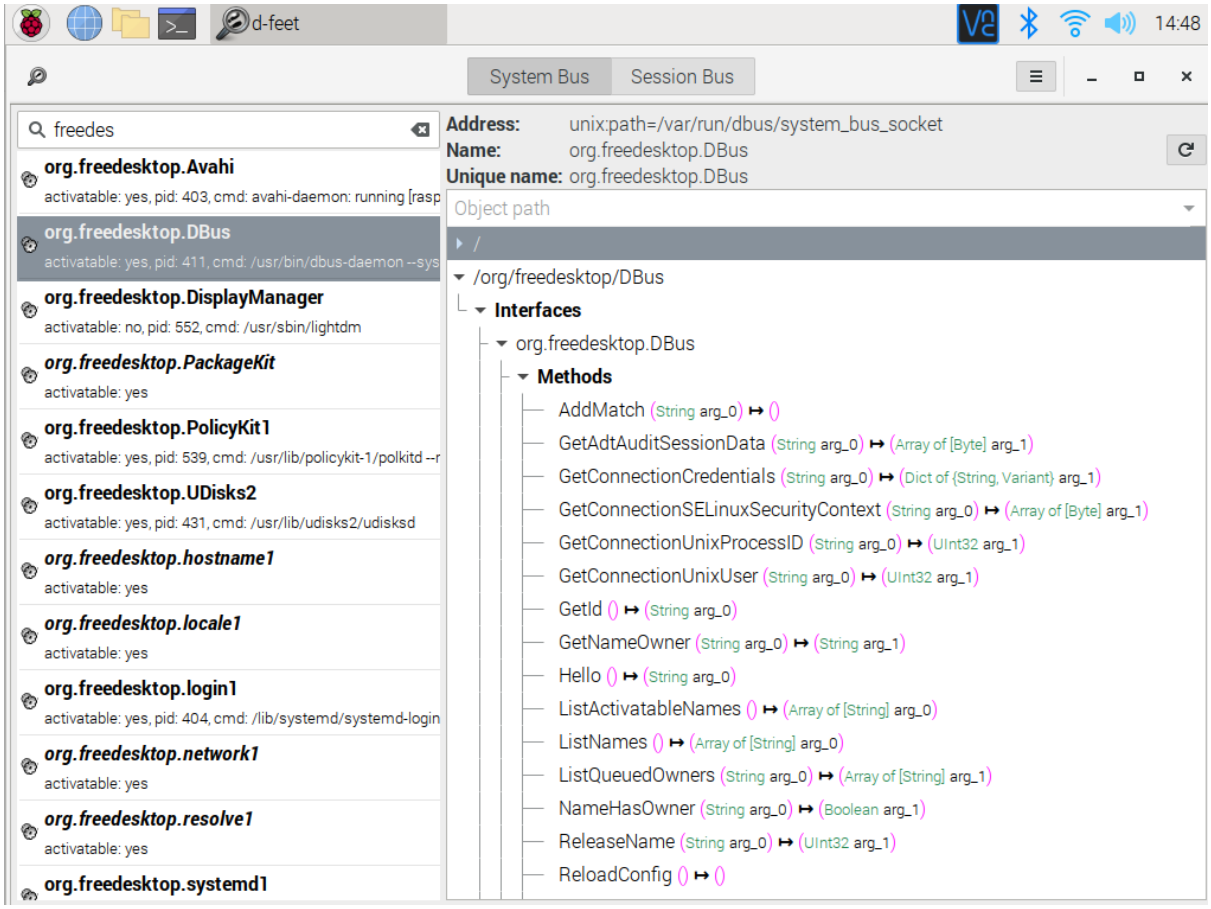


Figure 7 - org.freedesktop.DBus supports the DBus system itself

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:

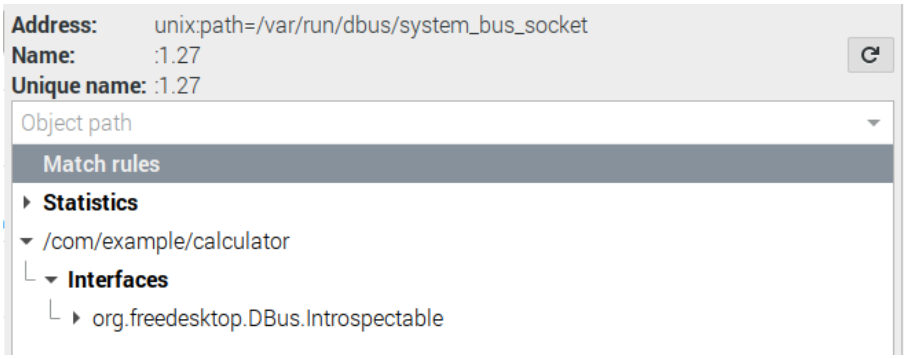


Figure 8 - Calculator object exported to the D-Bus system bus

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.

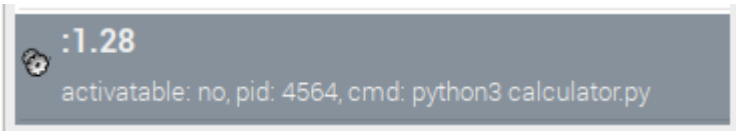
Update your code as shown:

```
#!/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:

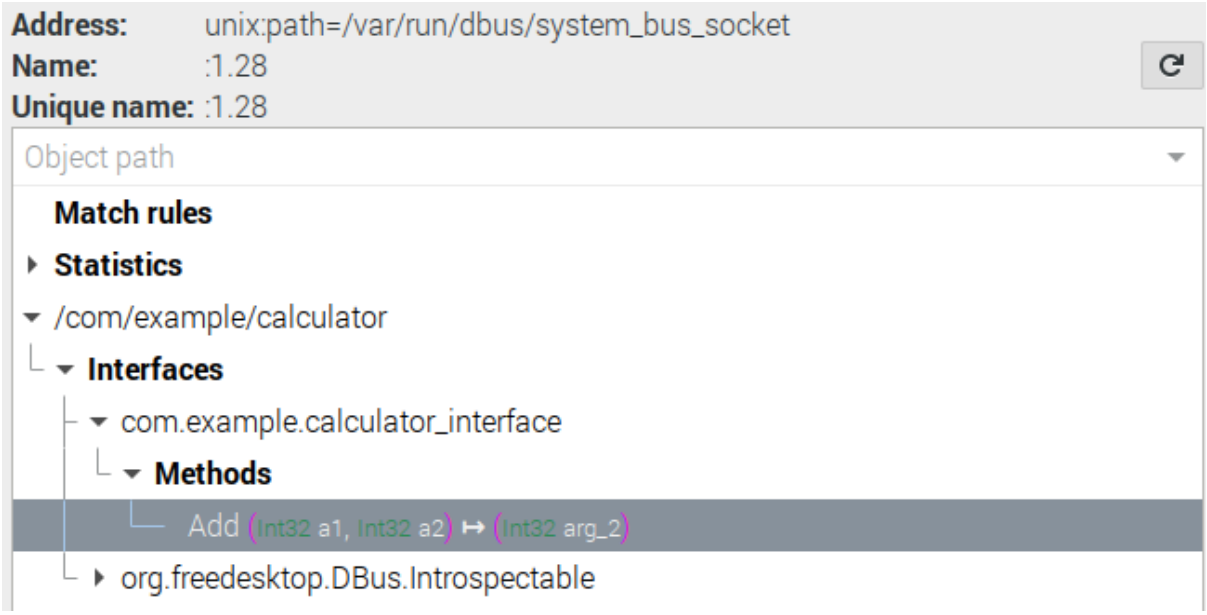


Figure 9 - Calculator object, interface and Add method

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.

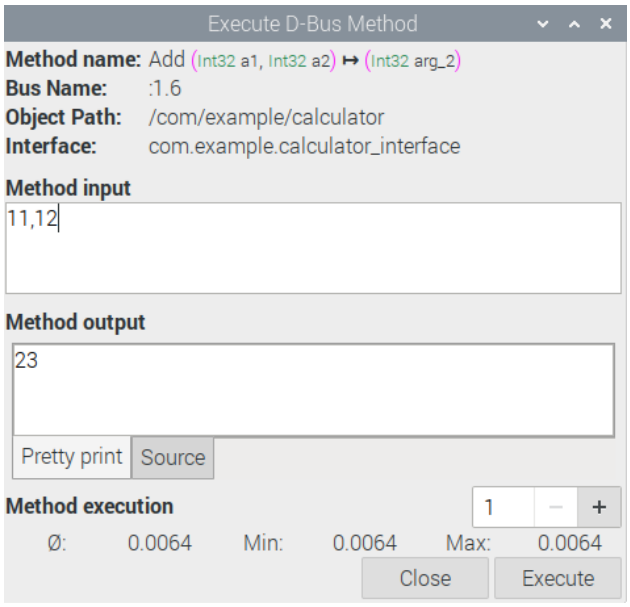


Figure 10 - Testing the Calculator

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.