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

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.

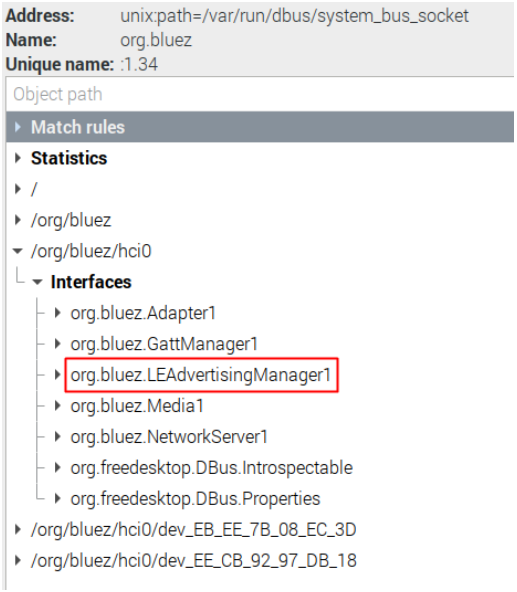

Figure 1 - This adapter implements LEAdvertisingManager1

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.

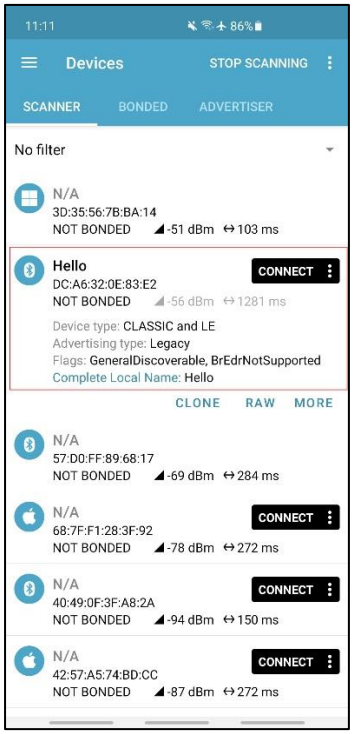

Figure 2 - Device advertising with local name "Hello" and Flags indicating general discoverable mode

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:

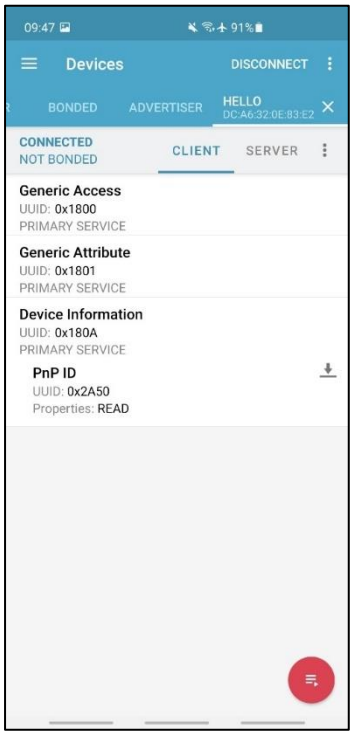

Figure 3 - Default GATT services

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.

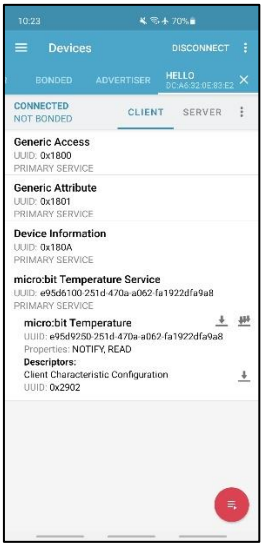

Figure 4 - The temperature service and characteristic

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.

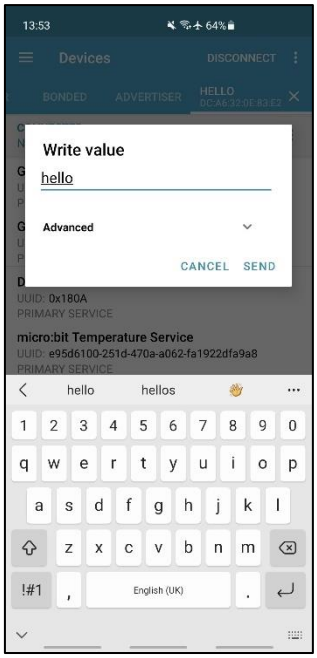

Figure 5 - using nRF Connect to write to the LED Text characteristic

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.

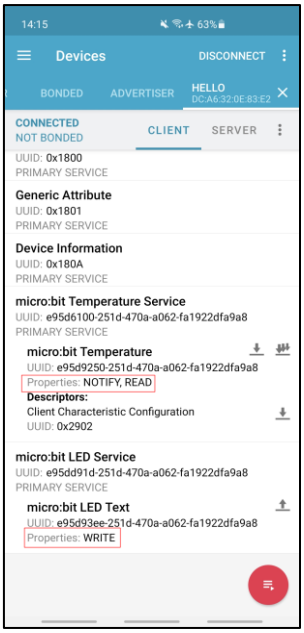

Figure 6 - characteristic properties

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.

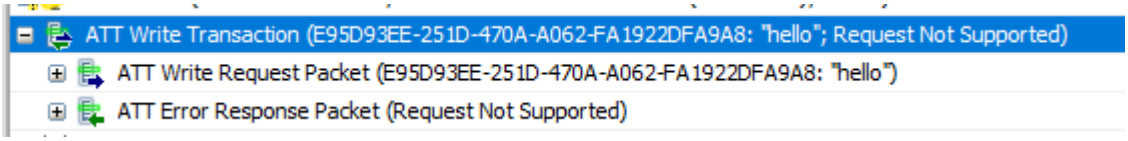

Figure 7 - request not supported

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*.

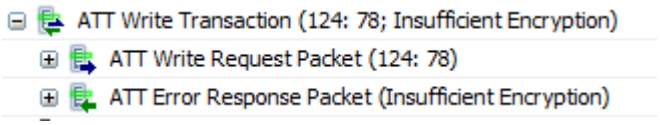

Figure 8 - An Insufficient Encryption error being returned

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.

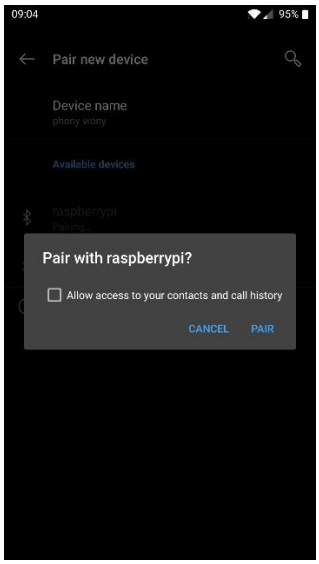

Figure 9 - Just Works pairing proceeding on an Android smartphone

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.

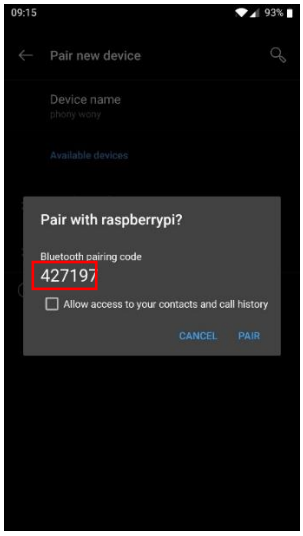

Figure 10 - Passkey pairing on an Android smartphone

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.

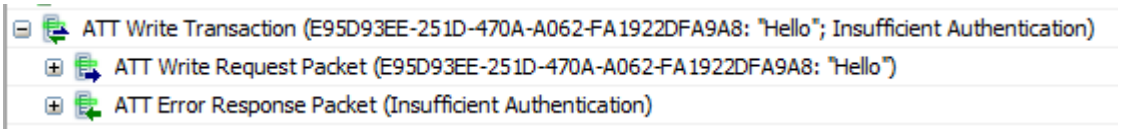

Figure 11 - Failed write attempt due to a non-authenticating pairing method having been used

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.

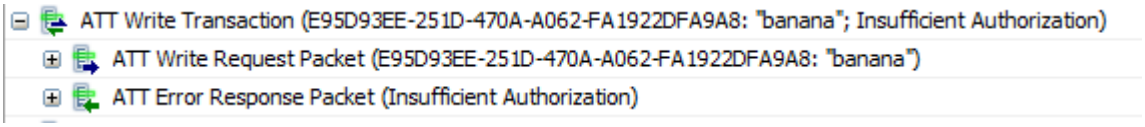

Figure 12 - insufficient authorization

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.