Interfacing with the BlueZ D-Bus API in C

If an application on a Linux system wishes to do anything with bluetooth, it will be doing so through BlueZ, the bluetooth linux stack. Unlike other Linux subsystems, such as ALSA, that provide their userspace API as a C library, BlueZ exposes its userspace API through D-Bus (Desktop Bus). D-Bus is an inter-process communication system that allows for flexible, language-agnostic communication between system services and applications.

While BlueZ and D-Bus are each individually fairly well-documented, information on using them together is quite sparse, especially for application's written in C. This guide strives to collect the information necessary to get started with writing C applications that use D-Bus, and more specifically, interact with the BlueZ bluetooth daemon through the GLib D-Bus binding.

D-Bus Basics

Architecture

The D-Bus system can be thought of as a network between processes within a machine. Moreover, processes on the D-Bus network follow a client-server architecture, where, for example, a user application is a client, and a system-daemon like BlueZ is a server.

Message Buses

The D-Bus network of a machine running D-Bus is exposed to processes as a system daemon called a *message-bus*, which is like a central router that takes a message from a process and routes it to its intended recipient process. In practice, there are two such message buses available to processes on a machine: the *session-bus* and the *system-bus*. The session-bus is for user-level applications to talk amongst each other, and the system-bus is for user-level applications to talk to system-level services.

The BlueZ daemon, being a system-level service, is found on the system-bus.

Service Names

To distinguish processes connected to a message bus, each process is assigned a *unique connection name*. In networking terms, this is like a device address. Unique connection names always begin with a colon character (:). To make discovering and connecting to specific services more convenient, applications can also reserve for themselves *well-known names*. In networking terms, the well-known name is like a website URL. It acts as a memorable, human readable identifier for the process.

The BlueZ daemon goes by the well-known name `org.bluez`.

To see the names of all processes currently connected to both the session and system busses, we can use the `busctl list` command:

```
$ busctl list
NAME                                 PID PROCESS         USER            CONNECTION
:1.0                                 395 systemd-resolve systemd-resolve :1.0
:1.1                                 379 systemd-network systemd-network :1.1
:1.12                                961 systemd         user            :1.12
:1.13                                982 pipewire-pulse  user            :1.13
:1.14                                980 pipewire        user            :1.14
:1.15                                981 wireplumber     user            :1.15
:1.16                                979 pipewire        user            :1.16
:1.17                                993 rtkit-daemon    root            :1.17
:1.18                               1003 polkitd         polkitd         :1.18
:1.19                                981 wireplumber     user            :1.19
:1.2                                   1 systemd         root            :1.2
:1.3                                 514 avahi-daemon    avahi           :1.3
:1.37                               1413 busctl          user            :1.37
:1.4                                 640 systemd-logind  root            :1.4
:1.5                                 550 iwd             root            :1.5
:1.6                                 826 bluetoothd      root            :1.6
:1.7                                 790 unattended-upgr root            :1.7
:1.9                                 861 mender          root            :1.9
com.ubuntu.SoftwareProperties          - -               -               (activatable)
io.mender.AuthenticationManager      861 mender          root            :1.9
io.mender.UpdateManager              861 mender          root            :1.9
net.connman.iwd                      550 iwd             root            :1.5
org.bluez                            826 bluetoothd      root            :1.6
```

(Some columns and rows were ommited from the above output)

Objects

A process on a message-bus can expose a number of *objects* to other processes. Examples of objects exposed by the BlueZ daemon process include bluetooth adapters, connected bluetooth devices, and the BlueZ daemon itself. Objects in a process are distinguished by their unique *object path*. An object path looks like a filesystem filepath, with strings separated by forward slashes. While an object path does not need to follow any specific structure, it is often formatted to reflect the tree-like ownerships between objects within a process, as is the case with BlueZ's object paths.

The object path of a bluetooth adapter in the BlueZ daemon process looks like `/org/bluez/hci0`, where `hci0` is the name of the adapter's object instance name, and `/org/bluez/` is the parent BlueZ daemon object that owns it.

The object path of a bluetooth device looks like `/org/bluez/hci0/dev_0C_02_BD_78_41_0B`, where `dev_0C_02_BD_78_41_0B` is the device object itself, and the path prefix `/org/bluez/hci0/` indicates that the device was connected and owned by the adapter `hci0`.

To see the all the objects under a particular connection, we can use the `busctl tree` command:

```
$ busctl tree org.bluez
└─ /org
  └─ /org/bluez
    └─ /org/bluez/hci0
      └─ /org/bluez/hci0/dev_AB_CD_EF_GH_IJ_KL
        ├─ /org/bluez/hci0/dev_AB_CD_EF_GH_IJ_KL/fd0
        ├─ /org/bluez/hci0/dev_AB_CD_EF_GH_IJ_KL/player0
        ├─ /org/bluez/hci0/dev_AB_CD_EF_GH_IJ_KL/sep1
        ├─ /org/bluez/hci0/dev_AB_CD_EF_GH_IJ_KL/sep2
        ├─ /org/bluez/hci0/dev_AB_CD_EF_GH_IJ_KL/sep3
        ├─ /org/bluez/hci0/dev_AB_CD_EF_GH_IJ_KL/sep4
        └─ /org/bluez/hci0/dev_AB_CD_EF_GH_IJ_KL/sep5
```

Interfaces

A D-Bus interface defines a set of related methods, signals, and properties that an object on the D-Bus can expose. It's similar to an interface in object-oriented programming: it specifies what an object can do, without defining how it's implemented.

Each interface has a unique name following a reverse domain name style. Examples of interfaces implemented by BlueZ device objects are `org.bluez.Device1` and `org.freedesktop.DBus.Properties`. Note that the names for these interfaces are namespaced, in that `Device1` and `Properties` are the actual interface names, and the `org.bluez.` and `org.freedesktop.DBus.` prefixes distinguish who defined the interfaces. Moreover, there are number of *standard interfaces* specified by D-Bus, of which `org.freedesktop.DBus.Properties` is an example.

By sending the appropriate message to the message-bus, an application can call a method on an object's interface in another process, or read/write one of its properties. An example of a method on the `org.freedesktop.DBus.Properties` interface is the `Get` method, which takes the name of a property of an object and returns its value.

Additionally, an application can subscribe to signals on an object. An example of a signal on the `org.freedesktop.DBus.Properties` interface is the `PropertiesChanged` signal, which notifies subscribers when any one of the object's properties has changed.

To see the definition of an interface implemented by an object, as well as the current values of properties on that interface, use the command `busctl introspect <bus name> <object path> <interface name>`:

```
$ busctl introspect org.bluez /org/bluez/hci0/dev_0C_02_BD_78_41_0B org.bluez.Device1
NAME                TYPE      SIGNATURE RESULT/VALUE                         FLAGS
.CancelPairing      method    -         -                                    -
.Connect            method    -         -                                    -
.ConnectProfile     method    s         -                                    -
.Disconnect         method    -         -                                    -
.DisconnectProfile  method    s         -                                    -
.Pair               method    -         -                                    -
.Adapter            property  o         "/org/bluez/hci0"                    emits-change
.Address            property  s         "AB:CD:EF:GH:IJ:KL"                  emits-change
.AddressType        property  s         "public"                             emits-change
.Alias              property  s         "Samsung Phone"                      emits-change writable
.Appearance         property  q         -                                    emits-change
.Blocked            property  b         false                                emits-change writable
.Bonded             property  b         true                                 emits-change
.Class              property  u         5898764                              emits-change
.Connected          property  b         true                                 emits-change
.Icon               property  s         "phone"                              emits-change
.LegacyPairing      property  b         false                                emits-change
.ManufacturerData   property  a{qv}     -                                    emits-change
.Modalias           property  s         "bluetooth:v0075p0100d0201"          emits-change
.Name               property  s         "Samsung Phone"                      emits-change
```

```
.Paired            property  b         true                                    emits-change
.RSSI              property  n         -                                       emits-change
.ServiceData       property  a{sv}     -                                       emits-change
.ServicesResolved  property  b         true                                    emits-change
.Trusted           property  b         false                                   emits-change writable
.TxPower           property  n         -                                       emits-change
.UUIDs             property  as        18 "00001105-0000-1000-8000-00805f9b34f… emits-change
.WakeAllowed       property  b         -                                       emits-change writable
```

To see all the interfaces of an object and their definitions/values, use `busctl introspect` without specifying an interface.

Types

The D-Bus type system defines how data is encoded and transmitted between processes. It's a compact, strongly-typed system designed to be language-agnostic, while at the same time having close mappings to types available in most programming languages.

Every value sent over D-Bus is associated with a type signature: a string of characters which describes the format and structure of the type. For example, the type signature of a lone string is `s`, and the type signature of an array of strings is `a{s}`.

From the output

```
$ busctl introspect org.bluez /org/bluez/hci0 org.freedesktop.DBus.Properties
NAME                                TYPE      SIGNATURE RESULT/VALUE FLAGS
.Get                                method    ss        v            -
.GetAll                             method    s         a{sv}        -
.Set                                method    ssv       -            -
.PropertiesChanged                  signal    sa{sv}as  -            -
```

, we can see that the type signature of the `Get` method's parameters is `ss`, and the type signature of its return value is `v`. That is, the `Get` method takes two parameters of type string, and returns a single value of type variant.

Proxies

A D-Bus proxy is a local, in-process object that acts as a client-side representation of a remote D-Bus object. Instead of manually constructing and sending D-Bus messages, you interact with the proxy as if it were a normal object with methods, properties, and signals. The proxy handles all the message-passing under the hood. Moreover, proxies facilitate the mapping of D-Bus objects to objects in a programming framework's type system, such as a `java.lang.Object` in java, or a `GObject` in glib.

GDBus: The GLib D-Bus Binding

D-Bus bindings are available for many languages to make developing an application that interfaces with D-Bus more convenient. A D-Bus binding is a library that interfaces a programming language or framework to the D-Bus system, making it easier to send and receive messages without dealing with low-level D-Bus message protocol details. Moreover, bindings provide high-level APIs that map D-Bus concepts like buses, object paths, interfaces, methods, and signals into language-native constructs such as classes, objects, and function calls.

GDBus, which is the GLib library's D-Bus binding, is today's standard D-Bus binding for applications written in C. While C itself has no builtin OOP constructs, GLib provides the GObject type system that implements OOP in C, thus allowing GDBus to have a convenience almost on par with the D-Bus bindings for other languages.

GLib Installation

Installing on Target

To install glib on the BeagleY-AI, connect with ssh and run

```
sudo apt update
sudo apt install libglib2.0-0
```

Cross-compiling

To cross-compile an application using glib on a Debian host, we need to add the arm64 architecture:

```
sudo dpkg --add-architecture arm64
```

, install the arm64 version of the libglib dev package:

```
sudo apt update
sudo apt install libglib2.0-dev:arm64
```

and install the arm64 cross compiling packages:

```
sudo apt install crossbuild-essential-arm64
```

To link the glib and gio libraries to a target in our CMakeLists.txt file, we use `pkg-config`:

```
# File: CMakeLists.txt

...
find_package(PkgConfig REQUIRED)
pkg_check_modules(deps REQUIRED IMPORTED_TARGET glib-2.0 gio-2.0)
target_link_libraries(target PkgConfig::deps)
...
```

Now, when cross-compiling, we need to make sure CMake uses the arm64 version of `pkg-config` to discover the arm64 dev libraries. We do this by setting the `PKG_CONFIG_EXECUTABLE` variable in a toolchain file:

```
# File: aarch64-linux-gnu.cmake

# the name of the target operating system
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR aarch64)

# which compilers to use for C and C++
set(CMAKE_C_COMPILER   /usr/bin/aarch64-linux-gnu-gcc)
set(CMAKE_CXX_COMPILER /usr/bin/aarch64-linux-gnu-g++)

# where is the target environment located
set(CMAKE_FIND_ROOT_PATH /usr/aarch64-linux-gnu)
set(CMAKE_INCLUDE_PATH  /usr/include/aarch64-linux-gnu)
set(CMAKE_LIBRARY_PATH  /usr/lib/aarch64-linux-gnu)
set(CMAKE_PROGRAM_PATH  /usr/bin/aarch64-linux-gnu)

# adjust the default behavior of the FIND_XXX() commands:
# search programs in the host environment
set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)

set(PKG_CONFIG_EXECUTABLE aarch64-linux-gnu-pkg-config)

# search headers and libraries in the target environment
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY BOTH)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE BOTH)
```

and configure CMake with this toolchain file:

```
cmake -DCMAKE_TOOLCHAIN_FILE=path/to/toolchain/file/aarch64-linux-gnu.cmake -S . -B build
```

GDBusProxy

The type to represent a D-Bus proxy in `GDBus` is `GDBusProxy`. To create a `GDBusProxy` for an object with a known path, we can use one of the `g_dbus_proxy_new()` or `g_dbus_proxy_new_for_bus()` functions. For example,

```
GDBusProxy *adapter_proxy = g_dbus_proxy_new_for_bus_sync(
    G_BUS_TYPE_SYSTEM,              // Message Bus
    G_DBUS_PROXY_FLAGS_NONE,        // Flags
    NULL,                           // Interface info
    "org.bluez",                    // Bus name
    "/org/bluez/hci0",              // Object path
    "org.bluez.Adapter1",           // Interface Name
    NULL,                           // GCancellable
    &error
);
```

creates a proxy for the `org.bluez.Adapter1` interface on the `/org/bluez/hci0` adapter object instance.

With a proxy in hand, we can easily call methods for that proxy. To call the `StartDiscovery` method on `org.bluez.Adapter1` proxy, we use `g_dbus_proxy_call()`:

```
GVariant *result = g_dbus_proxy_call_sync(adapter_proxy,            // Proxy object
                                          "StartDiscovery",         // Method name
                                          NULL,                     // Parameters
                                          G_DBUS_CALL_FLAGS_NONE,   // Flags
                                          -1,                       // Timeout
                                          NULL,                     // GCancellable
                                          &error);
```

Something important to note here is that to wrap values for the D-Bus type system, GLib has the [GVariant] type. Method calls on objects/proxies take a `GVariant` as a parameter (we set parameters to `NULL` for `StartDiscovery` since it takes no parameters), and return a `GVariant`.

Setting/getting a property of a proxy can be done by calling the `Set` and `Get` methods of the `org.freedesktop.DBus.Properties` interface:

```
GVariant *param = g_variant_new("(ss)",
                                "org.bluez.Adapter1",
                                "Powered");

GVariant *property_value = g_dbus_proxy_call_sync(adapter_proxy,
                                                  "org.freedesktop.DBus.Properties.Get",
                                                  param,
                                                  G_DBUS_CALL_FLAGS_NONE,
                                                  -1,
                                                  NULL,
                                                  error);

g_variant_unref(param);
```

This gets the value of the `Powered` property on the `org.bluez.Adapter1` interface. From this example, we can see that it is possible to call a method on a different interface than the one that a proxy currently represents.

GDBusProxy Signals

One of the great benefits of using proxies is being able to easily subscribe to signals on a D-Bus object. A very useful signal available to `GDBusProxy` instances is the [g-properties-changed] signal. This signal fires when any one of the proxy's properties changes value. To subscribe to this signal on our adapter proxy, we use

```
g_signal_connect(adapter_proxy,
                 "g-properties-changed",
                 G_CALLBACK(property_changed_cb),
                 NULL);
```

where `property_changed_cb` is the callback to be called when a property changes. Now, whenever a property changes, we can do something like print a message indicating such:

```
void property_changed_cb(GDBusProxy *proxy,
                         GVariant *changed_properties,
                         const gchar *const *invalidated_properties)
{
    (void)proxy;
    (void)invalidated_properties;

    GVariantIter *iter;
    const gchar *key;
    GVariant *value;

    g_variant_get(changed_properties, "a{sv}", &iter);

    while (g_variant_iter_loop(iter, "{&sv}", &key, &value))
    {
        gchar *value_str = g_variant_print(value, FALSE);
        g_print("Property %s changed value to: %s", key, value_str);
        g_free(value_str);
    }
    g_variant_iter_free(iter);
```

```
}
```

GDBusObjectManagerClient

The D-Bus standard interface `org.freedesktop.DBus.ObjectManager` is a very useful interface for seeing which objects are present on a D-Bus connection, and also for getting notified when objects appear or disappear on a connection. Since bluetooth devices frequently connect and disconnect, and bluetooth profiles on those devices appear and disappear, it would serve us well to make use of the `org.freedesktop.DBus.ObjectManager` interface within a bluetooth application. Luckily, the BlueZ daemon implements `org.freedesktop.DBus.ObjectManager`, and moreover, GDBus provides the `GDBusObjectManagerClient` type for accessing the interface more conveniently.

To create a `GDBusObjectManagerClient` for the BlueZ daemon, we do

```
GDBusObjectManagerClient *manager =
    g_dbus_object_manager_client_new_for_bus_sync(G_BUS_TYPE_SYSTEM,
                                                  G_DBUS_OBJECT_MANAGER_CLIENT_FLAGS_NONE,
                                                  "org.bluez",
                                                  "/",
                                                  NULL, NULL, NULL, NULL,
                                                  &error);
```

To get notified whenever objects are added or removed from BlueZ, we can connect to the [`object-added`] and [`object-removed`] signals of `GDBusObjectManagerClient`:

```
g_signal_connect(manager,
                 "object-added",
                 G_CALLBACK(on_object_added),
                 NULL);

g_signal_connect(manager,
                 "object-removed",
                 G_CALLBACK(on_object_removed),
                 NULL);
```

To avoid having to attach a callback on every proxy in our application, we can also use the `interface-proxy-properties-changed` signal of `GDBusObjectManagerClient`, and get notified whenever a property changes on any interface under the BlueZ daemon.

Resources

- DBus Tutorial: Good overview of DBus constructs
- DBus Specification
- GIO Documentation: Documentation for the GIO library of GLib, under which GDBus is implemented

Useful Tools

- `bluetoothctl`: A terminal client for BlueZ that demonstrates almost all of the capabilities available through BlueZ's DBus interface. Worth exploring how to setup agents, connect to devices, and use the player menu through this tool.
- `busctl`: As seen in the D-Bus basics section of the guide, `busctl` is a very useful D-Bus utility that abstracts away the nitty gritty of interacting with D-Bus. I found this useful for exploring the BlueZ object tree and testing method calls and getting and setting properties. Use tab completion judiciously with this tool.

Some BlueZ D-Bus Objects and Interfaces

Object Path	Description	Relevant Interfaces
`/`	BlueZ bus root	`org.freedesktop.DBus.ObjectManager`
`/org/bluez`	BlueZ managers	`org.bluez.AgentManager1`,
`/org/bluez/hciX`	Bluetooth Adapter	`org.bluez.Adapter1`
`/org/bluez/hciX/dev_XX_XX_XX_XX_XX_XX`	Remote Device	`org.bluez.Device1`
`/org/bluez/hciX/dev_XX_XX_XX_XX_XX_XX/playerX`	Media Controller	`org.bluez.MediaPlayer1`, `org.freedesktop.DBus.Properties`
`/org/bluez/hciX/dev_XX_XX_XX_XX_XX_XX/fdX`	Media Transport	`org.bluez.MediaTransport1`