Matthew Johnson

Using the DBUS C API

THE CANONICAL ADDRESS OF THIS DOCUMENT IS NOW http://dbus.freedesktop.org/doc/dbus/libdbus-tutorial.html

D-BUS is a message bus system, a simple way for applications to talk to one another The low-level API for DBUS is written in C but most of the documentation and code is written for a higher level binding, such as Python or GLib. Here I provide tutorial/howto for a basic server and client using the C API directly, including example code. The DBUS website has Doxygen documentation for the C API.

NOTE: you should not use this API unless you absolutely have to. Application developers should use one of the bindings if at all possible. If you are writing in C then the GLib bindings are recommended.

Common Code

A lot of the code is common no matter what you want to do on the Bus. First you need to connect to the bus. There is normally a system and a session bus. The DBUS config may restrict who can connect to the system bus. Secondly, you need to request a name on the bus. For simplicity I don't cope with the situation where someone already owns that name.

```
   DBusError err;
   DBusConnection* conn;
   int ret;
   // initialise the errors
   dbus_error_init(&err);
```

```
// connect to the bus
   conn = dbus_bus_get(DBUS_BUS_SESSION, &err);
   if (dbus_error_is_set(&err)) {
      fprintf(stderr, "Connection Error (%s)\n", err.message);
      dbus_error_free(&err);
   }
   if (NULL == conn) {
      exit(1);
   }
```

```
// request a name on the bus
   ret = dbus_bus_request_name(conn, "test.method.server",
         DBUS_NAME_FLAG_REPLACE_EXISTING
         , &err);
   if (dbus_error_is_set(&err)) {
      fprintf(stderr, "Name Error (%s)\n", err.message);
      dbus_error_free(&err);
   }
   if (DBUS_REQUEST_NAME_REPLY_PRIMARY_OWNER != ret) {
      exit(1);
   }
```

After you have finished with the bus you should close the connection. Note that this is a shared connection to the bus so you probably only want to do this just before the application terminates.

```
   dbus_connection_close(conn);
```

Sending a Signal

The simplest operation is sending a broadcast signal to the bus. To do this you need to create a DBusMessage object representing the signal, and specifying what object and interface the signal represents. You then need to add any parameters to it as appropriate, and send it to the bus. Finally you need to free the message. Several methods return false on out of memory, this should be checked for and handled.

```
dbus_uint32_t serial = 0; // unique number to associate replies w
DBusMessage* msg;
DBusMessageIter args;

// create a signal and check for errors
msg = dbus_message_new_signal("/test/signal/Object", // object na
      "test.signal.Type", // interface name of the signal
      "Test"); // name of the signal
if (NULL == msg)
{
   fprintf(stderr, "Message Null\n");
   exit(1);
}

// append arguments onto signal
dbus_message_iter_init_append(msg, &args);
if (!dbus_message_iter_append_basic(&args, DBUS_TYPE_STRING, &sig
   fprintf(stderr, "Out Of Memory!\n");
   exit(1);
}

// send the message and flush the connection
if (!dbus_connection_send(conn, msg, &serial)) {
   fprintf(stderr, "Out Of Memory!\n");
   exit(1);
}
dbus_connection_flush(conn);

// free the message
dbus_message_unref(msg);
```

Calling a Method

Calling a remote method is very similar to sending a signal. You need to create the message and specify the name on the bus it is being send to, and what object, interface and method is being called. You then add parameters to the message as above. If you need a reply to the message you need to use a different method to send the message. This gives you a pending reply object which you can block on to wait for the reply and get another `DBusMessage*` as the reply. From this you can read the parameters returned by the remote method.

```
DBusMessage* msg;
DBusMessageIter args;
DBusPendingCall* pending;

msg = dbus_message_new_method_call("test.method.server", // targe
      "/test/method/Object", // object to call on
      "test.method.Type", // interface to call on
      "Method"); // method name
if (NULL == msg) {
   fprintf(stderr, "Message Null\n");
   exit(1);
}

// append arguments
dbus_message_iter_init_append(msg, &args);
if (!dbus_message_iter_append_basic(&args, DBUS_TYPE_STRING, &para
   fprintf(stderr, "Out Of Memory!\n");
   exit(1);
}

// send message and get a handle for a reply
if (!dbus_connection_send_with_reply (conn, msg, &pending, -1)) {
```

```
      fprintf(stderr, "Out Of Memory!\n");
      exit(1);
   }
   if (NULL == pending) {
      fprintf(stderr, "Pending Call Null\n");
      exit(1);
   }
   dbus_connection_flush(conn);

   // free message
   dbus_message_unref(msg);
```

```
   bool stat;
   dbus_uint32_t level;

   // block until we receive a reply
   dbus_pending_call_block(pending);

   // get the reply message
   msg = dbus_pending_call_steal_reply(pending);
   if (NULL == msg) {
      fprintf(stderr, "Reply Null\n");
      exit(1);
   }
   // free the pending message handle
   dbus_pending_call_unref(pending);

   // read the parameters
   if (!dbus_message_iter_init(msg, &args))
      fprintf(stderr, "Message has no arguments!\n");
   else if (DBUS_TYPE_BOOLEAN != dbus_message_iter_get_arg_type(&args
      fprintf(stderr, "Argument is not boolean!\n");
   else
      dbus_message_iter_get_basic(&args, &stat);

   if (!dbus_message_iter_next(&args))
      fprintf(stderr, "Message has too few arguments!\n");
   else if (DBUS_TYPE_UINT32 != dbus_message_iter_get_arg_type(&args)
      fprintf(stderr, "Argument is not int!\n");
   else
      dbus_message_iter_get_basic(&args, &level);

   printf("Got Reply: %d, %d\n", stat, level);

   // free reply and close connection
   dbus_message_unref(msg);
```

Receiving a Signal

These next two operations require reading messages from the bus and handling them. There is not a simple way in the C API as of 0.50 to do this. Because DBUS is designed to use OO bindings and an event based model all the methods that handle the messages on the wire use callbacks. I have submitted a patch which should make it into the next release which allows messages to be read and marshalled from the wire in a non-blocking operation. This is the `dbus_connection_read_write()` method, which is required to use the code on this page.

To receive a signal you have to tell the bus what signals you are interested in so they are sent to your application, then you read messages off the bus and can check what type they are and what interfaces and signal names each signal represents.

```
// add a rule for which messages we want to see
   dbus_bus_add_match(conn,
         "type='signal',interface='test.signal.Type'",
         &err); // see signals from the given interface
   dbus_connection_flush(conn);
   if (dbus_error_is_set(&err)) {
      fprintf(stderr, "Match Error (%s)\n", err.message);
      exit(1);
```

```
    }
```

```
// loop listening for signals being emmitted
    while (true) {

        // non blocking read of the next available message
        dbus_connection_read_write(conn, 0);
        msg = dbus_connection_pop_message(conn);

        // loop again if we haven't read a message
        if (NULL == msg) {
            sleep(1);
            continue;
        }

        // check if the message is a signal from the correct interface
        if (dbus_message_is_signal(msg, "test.signal.Type", "Test")) {
            // read the parameters
            if (!dbus_message_iter_init(msg, &args))
                fprintf(stderr, "Message has no arguments!\n");
            else if (DBUS_TYPE_STRING != dbus_message_iter_get_arg_type
                fprintf(stderr, "Argument is not string!\n");
            else {
                dbus_message_iter_get_basic(&args, &sigvalue);
                printf("Got Signal with value %s\n", sigvalue);
            }
        }

        // free the message
        dbus_message_unref(msg);
    }
```

Note that the use of `sleep` in this example is to simulate some other thing we are blocking on, such as `select`. If the only thing happening in this thread is the DBus communication then the `dbus_connection_read_write` call can take a timeout parameter, which is how many ms to block for. -1 blocks forever.

Exposing a Method to be called

To expose a method which may be called by other DBUS applications you have to listen for messages as above, then when you get a method call corresponding to the method you exposed you parse out the parameters, construct a reply message from the original and populate its parameters with the return value. Finally you have to send and free the reply.

```
// loop, testing for new messages
    while (true) {
        // non blocking read of the next available message
        dbus_connection_read_write(conn, 0);
        msg = dbus_connection_pop_message(conn);

        // loop again if we haven't got a message
        if (NULL == msg) {
            sleep(1);
            continue;
        }

        // check this is a method call for the right interface and meti
        if (dbus_message_is_method_call(msg, "test.method.Type", "Metho
            reply_to_method_call(msg, conn);

        // free the message
        dbus_message_unref(msg);
    }
```

```
void reply_to_method_call(DBusMessage* msg, DBusConnection* conn)
{
    DBusMessage* reply;
    DBusMessageIter args;
```

```
    DBusConnection* conn;
    bool stat = true;
    dbus_uint32_t level = 21614;
    dbus_uint32_t serial = 0;
    char* param = "";

    // read the arguments
    if (!dbus_message_iter_init(msg, &args))
       fprintf(stderr, "Message has no arguments!\n");
    else if (DBUS_TYPE_STRING != dbus_message_iter_get_arg_type(&args)
       fprintf(stderr, "Argument is not string!\n");
    else
       dbus_message_iter_get_basic(&args, &param);
    printf("Method called with %s\n", param);

    // create a reply from the message
    reply = dbus_message_new_method_return(msg);

    // add the arguments to the reply
    dbus_message_iter_init_append(reply, &args);
    if (!dbus_message_iter_append_basic(&args, DBUS_TYPE_BOOLEAN, &sta
       fprintf(stderr, "Out Of Memory!\n");
       exit(1);
    }
    if (!dbus_message_iter_append_basic(&args, DBUS_TYPE_UINT32, &leve
       fprintf(stderr, "Out Of Memory!\n");
       exit(1);
    }

    // send the reply && flush the connection
    if (!dbus_connection_send(conn, reply, &serial)) {
       fprintf(stderr, "Out Of Memory!\n");
       exit(1);
    }
    dbus_connection_flush(conn);

    // free the reply
    dbus_message_unref(reply);
}
```

Code Examples

That should be all you need to write a simple server and client for DBUS using the C API. The code snippets above come from dbus-example.c which you can download and test. It contains code for all four operations above.

- To receive signals: `dbus-example receive`
- To send a signal: `dbus-example send param`
- To listen for method calls: `dbus-example listen`
- To call the method: `dbus-example query param`