# Maemo Diablo Implementing and using D-Bus signals

Training Material

February 9, 2009

# Contents

Chapter 1

# Implementing and using D-Bus signals

## 1.1 D-Bus Signal properties

Doing remote method invocations over the D-Bus is only one half of D-Bus capabilities. As was noted before, D-Bus also supports a _broadcast_ method of communication, which is also _asynchronous_. This mechanism is called a _signal_ (in D-Bus terminology) and is useful when you need to notify a lot of receivers about a state change that could affect them. Some examples where signals could be useful are notifying a lot of receivers if the system is being shut down, network connectivity has been lost and similar system wide conditions. This way, the receivers do not need to poll for the status continuously.

However, signals are not the solution to all problems. If a receiver is not processing its D-Bus messages quickly enough (or there just are too many), a signal might get lost on its way to the receiver. There might also be other complications, as with any RPC mechanism. For these reasons, if your application requires extra reliability, you will need to think on how to arrange it. One possibility would be to occasionally check the state that your application is interested in, assuming it can be checked over the D-Bus. Just do not do it too often (once a minute or less often and try to do it only when your application is already active for other reasons). This model will lead to reduction in battery life, so think hard before adopting it.

Signals in D-Bus are able to carry information. Each signal has its own name (specified in the interface XML) as well as "arguments". In signal's case, the argument list is actually just a list of information that is passed along the signal, and shouldn't be confused with method call parameters (although both are delivered in the same manner).

Signals do not "return", meaning that when a D-Bus signal is sent, no reply will be received, nor will be expected. If the signal emitter wants to be sure that the signal was delivered, additional mechanisms need to be constructed for this (D-Bus does not include them directly). A D-Bus signal is very similar to most datagram based network protocols, for example UDP. Sending a signal will succeed even if there are no receivers interested in that specific signal.

Most D-Bus language bindings will attempt to map D-Bus signals into something more natural in the target language. Since GLib already supports the notion of signals (as GLib signals), this mapping is quite natural. So in practise, your client will register for GLib signals and then handle the signals in callback functions (a special wrapper function must be used to register for the wrapped signals: `dbus_g_proxy_connect_signal`).

## 1.2 Declaring signals in the interface XML

We'll next extend our Value object so that it will contain two threshold values (minimum and maximum) and the object will emit signals whenever a set operation will fall outside the thresholds.

We'll also emit a signal whenever a value is changed (the binary content of the new value is different from the old one).

In order to make the signals available to introspection data, we modify the interface XML file accordingly:

```
<node>
  <interface name="org.maemo.Value">

  <!-- ... Listing cut for brevity ... -->

    <!-- Signal (D-Bus) definitions -->

    <!-- NOTE: The current version of dbus-bindings-tool doesn't
         actually enforce the signal arguments _at_all_. Signals need
         to be declared in order to be passed through the bus itself,
         but otherwise no checks are done! For example, you could
         leave the signal arguments unspecified completely, and the
         code would still work. -->

    <!-- Signals to tell interested clients about state change.
         We send a string parameter with them. They never can have
         arguments with direction=in. -->
    <signal name="changed_value1">
      <arg type="s" name="change_source_name" direction="out"/>
    </signal>

    <signal name="changed_value2">
      <arg type="s" name="change_source_name" direction="out"/>
    </signal>

    <!-- Signals to tell interested clients that values are outside
         the internally configured range (thresholds). -->
    <signal name="outofrange_value1">
      <arg type="s" name="outofrange_source_name" direction="out"/>
    </signal>
    <signal name="outofrange_value2">
      <arg type="s" name="outofrange_source_name" direction="out"/>
    </signal>

  </interface>
</node>
```

Listing 1.1: Adding the signal definitions to the interface XML (glib-dbus-signals/value-dbus-interface.xml)

The signal definitions are required if you're planning to use the `dbus-bindings-tool`, however, the argument specification for each signal is not required by the tool.

In-fact, it will just ignore all argument specifications, and as you'll see below, we have to do a lot of "manual coding" in order to implement and use the signals (on both client and server side). `dbus-bindings-tool` might get more features in the future, but for now we'll have to sweat a bit.

## 1.3 Emitting signals from a GObject

So that we can later change easily the signal names, we'll define them in a header file and use the header file in both server and client. This is the section with the signal names:

```
/* Symbolic constants for the signal names to use with GLib.
   These need to map into the D-Bus signal names. */
#define SIGNAL_CHANGED_VALUE1    "changed_value1"
#define SIGNAL_CHANGED_VALUE2    "changed_value2"
#define SIGNAL_OUTOFRANGE_VALUE1 "outofrange_value1"
#define SIGNAL_OUTOFRANGE_VALUE2 "outofrange_value2"
```

Listing 1.2: Defining symbols for signal names for easier renaming (glib-dbus-signals/common-defs.h)

Before an GObject can emit a GLib signal, the signal itself needs to be defined and created. This is best done in the class constructor code (since the signal types need to be created only once):

```
/**
 * Define enumerations for the different signals that we can generate
 * (so that we can refer to them within the signals-array [below]
 * using symbolic names). These are not the same as the signal name
 * strings.
 *
 * NOTE: E_SIGNAL_COUNT is NOT a signal enum. We use it as a
 *       convenient constant giving the number of signals defined so
 *       far. It needs to be listed last.
 */
typedef enum {
  E_SIGNAL_CHANGED_VALUE1,
  E_SIGNAL_CHANGED_VALUE2,
  E_SIGNAL_OUTOFRANGE_VALUE1,
  E_SIGNAL_OUTOFRANGE_VALUE2,
  E_SIGNAL_COUNT
} ValueSignalNumber;

  /*... Listing cut for brevity ...*/

typedef struct {
  /* The parent class state. */
  GObjectClass parent;
  /* The minimum number under which values will cause signals to be
     emitted. */
  gint thresholdMin;
  /* The maximum number over which values will cause signals to be
     emitted. */
  gint thresholdMax;
  /* Signals created for this class. */
  guint signals[E_SIGNAL_COUNT];
} ValueObjectClass;
```

```
  /*... Listing cut for brevity ...*/

/**
 * Per class initializer
 *
 * Sets up the thresholds (-100 .. 100), creates the signals that we
 * can emit from any object of this class and finally registers the
 * type into the GLib/D-Bus wrapper so that it may add its own magic.
 */
static void value_object_class_init(ValueObjectClass* klass) {

  /* Since all signals have the same prototype (each will get one
     string as a parameter), we create them in a loop below. The only
     difference between them is the index into the klass->signals
     array, and the signal name.

     Since the index goes from 0 to E_SIGNAL_COUNT-1, we just specify
     the signal names into an array and iterate over it.

     Note that the order here must correspond to the order of the
     enumerations before. */
  const gchar* signalNames[E_SIGNAL_COUNT] = {
    SIGNAL_CHANGED_VALUE1,
    SIGNAL_CHANGED_VALUE2,
    SIGNAL_OUTOFRANGE_VALUE1,
    SIGNAL_OUTOFRANGE_VALUE2 };
  /* Loop variable */
  int i;

  dbg("Called");
  g_assert(klass != NULL);

  /* Setup sane minimums and maximums for the thresholds. There is no
     way to change these afterwards (currently), so you can consider
     them as constants. */
  klass->thresholdMin = -100;
  klass->thresholdMax = 100;

  dbg("Creating signals");

  /* Create the signals in one loop, since they all are similar
     (except for the names). */
  for (i = 0; i < E_SIGNAL_COUNT; i++) {
    guint signalId;

    /* Most of the time you will encounter the following code without
       comments. This is why all the parameters are documented
       directly below. */
    signalId =
      g_signal_new(signalNames[i], /* str name of the signal */
                   /* GType to which signal is bound to */
                   G_OBJECT_CLASS_TYPE(klass),
                   /* Combination of GSignalFlags which tell the
                      signal dispatch machinery how and when to
                      dispatch this signal. The most common is the
                      G_SIGNAL_RUN_LAST specification. */
                   G_SIGNAL_RUN_LAST,
                   /* Offset into the class structure for the type
                      function pointer. Since we're implementing a
                      simple class/type, we'll leave this at zero. */
                   0,
                   /* GSignalAccumulator to use. We don't need one. */
```

```
                    NULL,
                    /* User-data to pass to the accumulator. */
                    NULL,
                    /* Function to use to marshal the signal data into
                       the parameters of the signal call. Luckily for
                       us, GLib (GCClosure) already defines just the
                       function that we want for a signal handler that
                       we don't expect any return values (void) and
                       one that will accept one string as parameter
                       (besides the instance pointer and pointer to
                       user-data).

                       If no such function would exist, you would need
                       to create a new one (by using glib-genmarshal
                       tool). */
                    g_cclosure_marshal_VOID__STRING,
                    /* Return GType of the return value. The handler
                       does not return anything, so we use G_TYPE_NONE
                       to mark that. */
                    G_TYPE_NONE,
                    /* Number of parameter GTypes to follow. */
                    1,
                    /* GType(s) of the parameters. We only have one. */
                    G_TYPE_STRING);
    /* Store the signal Id into the class state, so that we can use
       it later. */
    klass->signals[i] = signalId;

    /* Proceed with the next signal creation. */
  }
  /* All signals created. */

  dbg("Binding to GLib/D-Bus");

  /*... Listing cut for brevity ...*/

}
```

Listing 1.3: Signal enumerations their storage and their creation (glib-dbus-signals/server.c)

The signal types will be kept in the class structure so that they can be referenced easily by the signal emitting utility (covered next). The class constructor code will also set up the threshold limits, which in our implementation will be immutable (they cannot be changed). You might want to experiment with adding more methods to adjust the thresholds at your option.

Emitting the signals is then quite easy, but in order to reduce code amount, we'll create an utility function that will launch a given signal based on its enumeration:

```
/**
 * Utility helper to emit a signal given with internal enumeration and
 * the passed string as the signal data.
 *
 * Used in the setter functions below.
 */
static void value_object_emitSignal(ValueObject* obj,
                                    ValueSignalNumber num,
                                    const gchar* message) {
```

```
  /* In order to access the signal identifiers, we need to get a hold
     of the class structure first. */
  ValueObjectClass* klass = VALUE_OBJECT_GET_CLASS(obj);

  /* Check that the given num is valid (abort if not).
     Given that this file is the module actually using this utility,
     you can consider this check superfluous (but useful for
     development work). */
  g_assert((num < E_SIGNAL_COUNT) && (num >= 0));

  dbg("Emitting signal id %d, with message '%s'", num, message);

  /* This is the simplest way of emitting signals. */
  g_signal_emit(/* Instance of the object that is generating this
                   signal. This will be passed as the first parameter
                   to the signal handler (eventually). But obviously
                   when speaking about D-Bus, a signal caught on the
                   other side of D-Bus will be first processed by
                   the GLib-wrappers (the object proxy) and only then
                   processed by the signal handler. */
                obj,
                /* Signal id for the signal to generate. These are
                   stored inside the class state structure. */
                klass->signals[num],
                /* Detail of signal. Since we are not using detailed
                   signals, we leave this at zero (default). */
                0,
                /* Data to marshal into the signal. In our case it's
                   just one string. */
                message);
  /* g_signal_emit returns void, so we cannot check for success. */

  /* Done emitting signal. */
}
```

Listing 1.4: Utility function to emit a signal with specified message (glib-dbus-signals/server.c)

So that we don't need to check the threshold values in multiple places in the source code, we also implement that as a separate function. Emitting the "threshold exceeded" signal is still up to the caller.

```
/**
 * Utility to check the given integer against the thresholds.
 * Will return TRUE if thresholds are not exceeded, FALSE otherwise.
 *
 * Used in the setter functions below.
 */
static gboolean value_object_thresholdsOk(ValueObject* obj,
                                          gint value) {

  /* Thresholds are in class state data, get access to it */
  ValueObjectClass* klass = VALUE_OBJECT_GET_CLASS(obj);

  return ((value >= klass->thresholdMin) &&
          (value <= klass->thresholdMax));
}
```

Listing 1.5: Utility function to check whether given value is within thresholds or not (glib-dbus-signals/server.c)

Both utility functions are then used from within the respective set functions, one of which is presented below:

```c
/**
 * Function that gets called when someone tries to execute "setvalue1"
 * over the D-Bus. (Actually the marshalling code from the stubs gets
 * executed first, but they will eventually execute this function.)
 */
gboolean value_object_setvalue1(ValueObject* obj, gint valueIn,
                                                  GError** error) {

  dbg("Called (valueIn=%d)", valueIn);

  g_assert(obj != NULL);

  /* Compare the current value against old one. If they're the same,
     we don't need to do anything (except return success). */
  if (obj->value1 != valueIn) {
    /* Change the value. */
    obj->value1 = valueIn;

    /* Emit the "changed_value1" signal. */
    value_object_emitSignal(obj, E_SIGNAL_CHANGED_VALUE1, "value1");

    /* If new value falls outside the thresholds, emit
       "outofrange_value1" signal as well. */
    if (!value_object_thresholdsOk(obj, valueIn)) {
      value_object_emitSignal(obj, E_SIGNAL_OUTOFRANGE_VALUE1,
                                   "value1");
    }
  }
  /* Return success to GLib/D-Bus wrappers. In this case we don't need
     to touch the supplied error pointer-pointer. */
  return TRUE;
}
```

Listing 1.6: setvalue1 with "value-changed" and "outofrange" signal support (glib-dbus-signals/server.c)

You might be wondering the role of the "`value1`" string parameter that is sent along both of the signals above. Sending the signal origin name with the signal allows one to reuse the same callback function in the client. It's quite rare that this kind of "source naming" would be useful, but it allows us to write a slightly shorter client program.

The implementation of `setvalue2` is almost identical, but deals with the `gdouble` parameter.

The `getvalue`-functions are identical to the versions before as is the Makefile.

We next build the server and start it on the background (in preparation for testing with `dbus-send`):

```
[sbox-DIABLO_X86: ~/glib-dbus-signals] > make server
dbus-binding-tool --prefix=value_object --mode=glib-server \
  value-dbus-interface.xml > value-server-stub.h
cc -I/usr/include/dbus-1.0 -I/usr/lib/dbus-1.0/include \
 -I/usr/include/glib-2.0 -I/usr/lib/glib-2.0/include -g -Wall \
 -DG_DISABLE_DEPRECATED -DNO_DAEMON -DPROGNAME=\"server\" \
 -c server.c -o server.o
cc server.o -o server -ldbus-glib-1 -ldbus-1 -lgobject-2.0 -lglib-2.0
[sbox-DIABLO_X86: ~/glib-dbus-signals] > run-standalone.sh ./server &
[1] 15293
server:main Connecting to the Session D-Bus.
server:main Registering the well-known name (org.maemo.Platdev_ex)
server:main RequestName returned 1.
server:main Creating one Value object.
server:value_object_class_init: Called
server:value_object_class_init: Creating signals
server:value_object_class_init: Binding to GLib/D-Bus
server:value_object_class_init: Done
server:value_object_init: Called
server:main Registering it on the D-Bus.
server:main Ready to serve requests (daemonizing).
server: Not daemonizing (built with NO_DAEMON-build define)
[sbox-DIABLO_X86: ~/glib-dbus-signals] >
```

Building and starting the server with signals

We then proceed to test the `setvalue1` method:

```
[sbox-DIABLO_X86: ~/glib-dbus-signals] > run-standalone.sh dbus-send \
 --type=method_call --print-reply --dest=org.maemo.Platdev_ex \
 /GlobalValue org.maemo.Value.setvalue1 int32:10
server:value_object_setvalue1: Called (valueIn=10)
server:value_object_emitSignal: Emitting signal id 0, with message 'value1'
method return sender=:1.38 -> dest=:1.41
[sbox-DIABLO_X86: ~/glib-dbus-signals] > run-standalone.sh dbus-send \
 --type=method_call --print-reply --dest=org.maemo.Platdev_ex \
 /GlobalValue org.maemo.Value.setvalue1 int32:-200
server:value_object_setvalue1: Called (valueIn=-200)
server:value_object_emitSignal: Emitting signal id 0, with message 'value1'
server:value_object_emitSignal: Emitting signal id 2, with message 'value1'
method return sender=:1.38 -> dest=:1.42
```

And then `setvalue2` (with doubles). You might notice something fishy in the threshold triggering at this point:

```
[sbox-DIABLO_X86: ~/glib-dbus-signals] > run-standalone.sh dbus-send \
 --type=method_call --print-reply --dest=org.maemo.Platdev_ex \
 /GlobalValue org.maemo.Value.setvalue2 double:100.5
server:value_object_setvalue2: Called (valueIn=100.500)
server:value_object_emitSignal: Emitting signal id 1, with message 'value2'
method return sender=:1.38 -> dest=:1.44
[sbox-DIABLO_X86: ~/glib-dbus-signals] > run-standalone.sh dbus-send \
 --type=method_call --print-reply --dest=org.maemo.Platdev_ex \
 /GlobalValue org.maemo.Value.setvalue2 double:101
server:value_object_setvalue2: Called (valueIn=101.000)
server:value_object_emitSignal: Emitting signal id 1, with message 'value2'
server:value_object_emitSignal: Emitting signal id 3, with message 'value2'
method return sender=:1.38 -> dest=:1.45
```

Since the threshold testing logic will truncate the `gdouble` before testing against the (integer) thresholds, a value of 100.5 will be detected as 100, and will still fit within the thresholds.

Instead of printing out the emitted signal names, their enumeration values are printed. This could be rectified with a small enumeration to string table, but it was emitted from the program for simplicity.

You will also notice that other than seeing the server messages about emitting the signals, there's not a trace of them being sent or received. This is because dbus-send does not listen for signals. There is a separate tool for tracing signals and it will be covered at the end of this chapter (dbus-monitor).

## 1.4 Catching signals in GLib/D-Bus clients

In order to receive D-Bus signals in the client, one needs to do quite a bit of work per signal. This is because dbus-bindings-tool doesn't generate any code for signals (at this moment). The aim is to make the GLib wrappers emit GSignals whenever an interesting D-Bus signal will arrive. This also means that we'll need to register our interest for a particular D-Bus signal.

When implementing the callbacks for the signals, care needs to be taken in order to implement the prototype correctly. Since our signals will be sent with one attached string value, our callbacks will at least receive the string parameter. Besides the signal attached arguments, the callback will receive the proxy object through which the signal was received, and optional user specified data (which we don't use in our example, so it will be always NULL).

```
/**
 * Signal handler for the "changed" signals. These will be sent by the
 * Value-object whenever its contents change (whether within
 * thresholds or not).
 *
 * Like before, we use strcmp to differentiate between the two
 * values, and act accordingly (in this case by retrieving
 * synchronously the values using getvalue1 or getvalue2.
 *
 * NOTE: Since we use synchronous getvalues, it is possible to get
 *       this code stuck if for some reason the server would be stuck
 *       in an eternal loop.
 */
static void valueChangedSignalHandler(DBusGProxy* proxy,
                                      const char* valueName,
                                      gpointer userData) {
  /* Since method calls over D-Bus can fail, we'll need to check
     for failures. The server might be shut down in the middle of
     things, or might act badly in other ways. */
  GError* error = NULL;

  g_print(PROGNAME ":value-changed (%s)\n", valueName);

  /* Find out which value changed, and act accordingly. */
  if (strcmp(valueName, "value1") == 0) {
    gint v = 0;
    /* Execute the RPC to get value1. */
    org_maemo_Value_getvalue1(proxy, &v, &error);
    if (error == NULL) {
      g_print(PROGNAME ":value-changed Value1 now %d\n", v);
    } else {
      /* You could interrogate the GError further, to find out exactly
         what the error was, but in our case, we'll just ignore the
         error with the hope that some day (preferably soon), the
         RPC will succeed again (server comes back on the bus). */
      handleError("Failed to retrieve value1", error->message, FALSE);
    }
  } else {
    gdouble v = 0.0;
    org_maemo_Value_getvalue2(proxy, &v, &error);
    if (error == NULL) {
      g_print(PROGNAME ":value-changed Value2 now %.3f\n", v);
    } else {
      handleError("Failed to retrieve value2", error->message, FALSE);
    }
  }
  /* Free up error object if one was allocated. */
  g_clear_error(&error);
}
```

Listing 1.7: Signal handling callback for the changed_value signals (glib-dbus-signals/client.c)

The callback will first determine which was the source value which caused the signal to be generated. For this, it uses the string argument of the signal. It will then retrieve the current value using the respective RPC methods (`getvalue1` or `getvalue2`) and print out the value.

If any errors occur during the method calls, the errors will be printed out, but the program will continue to run. If an error does occur, the GError object will need to be freed (done with `g_clear_error`). We do not terminate the program on RPC errors since the condition might be temporary (the Value object server might be restarted later).

The code for the `outOfRangeSignalHandler` callback has been omitted since it doesn't contain anything beyond what `valueChangedSignalHandler` demonstrates.

Registering for the signals is a two-step process. We first need to register our interest in the D-Bus signals, and then install the callbacks for the respective GLib signals. This is done within `main`:

```
/**
 * The test program itself.
 *
 * 1) Setup GType/GSignal
 * 2) Create GMainLoop object
 * 3) Connect to the Session D-Bus
 * 4) Create a proxy GObject for the remote Value object
 * 5) Register signals that we're interested from the Value object
 * 6) Register signal handlers for them
 * 7) Start a timer that will launch timerCallback once per second.
 * 8) Run main-loop (forever)
 */
int main(int argc, char** argv) {

  /*... Listing cut for brevity ...*/

  remoteValue =
    dbus_g_proxy_new_for_name(bus,
                              VALUE_SERVICE_NAME, /* name */
                              VALUE_SERVICE_OBJECT_PATH, /* obj path */
                              VALUE_SERVICE_INTERFACE /* interface */);
  if (remoteValue == NULL) {
    handleError("Couldn't create the proxy object",
                "Unknown(dbus_g_proxy_new_for_name)", TRUE);
  }

  /* Register the signatures for the signal handlers.
     In our case, we'll have one string parameter passed to use along
     the signal itself. The parameter list is terminated with
     G_TYPE_INVALID (i.e., the GType for string objects. */

  g_print(PROGNAME ":main Registering signal handler signatures.\n");

  /* Add the argument signatures for the signals (needs to be done
     before connecting the signals). This might go away in the future,
     when the GLib-bindings will do automatic introspection over the
     D-Bus, but for now we need the registration phase. */
  { /* Create a local scope for variables. */

    int i;
    const gchar* signalNames[] = { SIGNAL_CHANGED_VALUE1,
                                   SIGNAL_CHANGED_VALUE2,
                                   SIGNAL_OUTOFRANGE_VALUE1,
                                   SIGNAL_OUTOFRANGE_VALUE2 };
    /* Iterate over all the entries in the above array.
       The upper limit for i might seem strange at first glance,
       but is quite common idiom to extract the number of elements
       in a statically allocated arrays in C.
       NOTE: The idiom will not work with dynamically allocated
             arrays. (Or rather it will, but the result is probably
             not what you expect.) */
    for (i = 0; i < sizeof(signalNames)/sizeof(signalNames[0]); i++) {
      /* Since the function doesn't return anything, we cannot check
         for errors here. */
      dbus_g_proxy_add_signal(/* Proxy to use */
                              remoteValue,
                              /* Signal name */
                              signalNames[i],
                              /* Will receive one string argument */
                              G_TYPE_STRING,
                              /* Termination of the argument list */
                              G_TYPE_INVALID);
    }
  } /* end of local scope */

  g_print(PROGNAME ":main Registering D-Bus signal handlers.\n");

  /* We connect each of the following signals one at a time,
     since we'll be using two different callbacks. */

  /* Again, no return values, cannot hence check for errors. */
  dbus_g_proxy_connect_signal(/* Proxy object */
                              remoteValue,
                              /* Signal name */
                              SIGNAL_CHANGED_VALUE1,
                              /* Signal handler to use. Note that the
                                 typecast is just to make the compiler
                                 happy about the function, since the
                                 prototype is not compatible with
                                 regular signal handlers. */
                              G_CALLBACK(valueChangedSignalHandler),
                              /* User-data (we don't use any). */
                              NULL,
                              /* GClosureNotify function that is
                                 responsible in freeing the passed
                                 user-data (we have no data). */
                              NULL);

  dbus_g_proxy_connect_signal(remoteValue, SIGNAL_CHANGED_VALUE2,
                              G_CALLBACK(valueChangedSignalHandler),
                              NULL, NULL);
  dbus_g_proxy_connect_signal(remoteValue, SIGNAL_OUTOFRANGE_VALUE1,
                              G_CALLBACK(outOfRangeSignalHandler),
                              NULL, NULL);

  dbus_g_proxy_connect_signal(remoteValue, SIGNAL_OUTOFRANGE_VALUE2,
                              G_CALLBACK(outOfRangeSignalHandler),
                              NULL, NULL);

  /* All signals are now registered and we're ready to handle them. */

  g_print(PROGNAME ":main Starting main loop (first timer in 1s).\n");

  /* Register a timer callback that will do RPC sets on the values.
     The userdata pointer is used to pass the proxy object to the
     callback so that it can launch modifications to the object. */
  g_timeout_add(1000, (GSourceFunc)timerCallback, remoteValue);

  /* Run the program. */
  g_main_loop_run(mainloop);
  /* Since the main loop is not stopped (by this code), we shouldn't
     ever get here. The program might abort() for other reasons. */

  /* If it does, return failure as exit code. */
  return EXIT_FAILURE;
}
```

Listing 1.8: Register interest in D-Bus signals and installing callbacks to handle them (glib-dbus-signals/client.c)

When adding the argument signatures for the signals (with `dbus_g_proxy_add_signal`) one needs to be very careful with the parameter list. The signal argument types must be exactly the same as are sent from the server (irrespective of the argument specification in the interface XML). This is because the current version of `dbus-bindings-tool` does not generate any checks to enforce signal arguments based on the interface. In our simple case we only receive one string with each different signal, so this is not a big issue. The implementation for the callback function will need to match the argument specification given to the `_add_signal`-function, otherwise data layout on the stack will be incorrect, and bad things will happen.

Building the client happens in the same manner as before (`make client`). Since the server is still (hopefully) running on the background, we'll now start the client in the same session:

```
[sbox-DIABLO_X86: ~/glib-dbus-signals] > run-standalone.sh ./client
client:main Connecting to Session D-Bus.
client:main Creating a GLib proxy object for Value.
client:main Registering signal handler signatures.
client:main Registering D-Bus signal handlers.
client:main Starting main loop (first timer in 1s).
server:value_object_setvalue1: Called (valueIn=-80)
server:value_object_emitSignal: Emitting signal id 0, with message 'value1'
client:timerCallback Set value1 to -80
server:value_object_setvalue2: Called (valueIn=-120.000)
server:value_object_emitSignal: Emitting signal id 1, with message 'value2'
server:value_object_emitSignal: Emitting signal id 3, with message 'value2'
client:timerCallback Set value2 to -120.000
client:value-changed (value1)
server:value_object_getvalue1: Called (internal value1 is -80)
client:value-changed Value1 now -80
client:value-changed (value2)
server:value_object_getvalue2: Called (internal value2 is -120.000)
client:value-changed Value2 now -120.000
client:out-of-range (value2)!
client:out-of-range Value 2 is outside threshold
server:value_object_setvalue1: Called (valueIn=-70)
server:value_object_emitSignal: Emitting signal id 0, with message 'value1'
client:timerCallback Set value1 to -70
server:value_object_setvalue2: Called (valueIn=-110.000)
server:value_object_emitSignal: Emitting signal id 1, with message 'value2'
server:value_object_emitSignal: Emitting signal id 3, with message 'value2'
client:timerCallback Set value2 to -110.000
client:value-changed (value1)
server:value_object_getvalue1: Called (internal value1 is -70)
client:value-changed Value1 now -70
client:value-changed (value2)
server:value_object_getvalue2: Called (internal value2 is -110.000)
client:value-changed Value2 now -110.000
...
```

Client calling RPC methods and receiving change and out-of-range signals

The client will start with the timer callback being executed once per second (like before). Each iteration it will call the `setvalue1` and `setvalue2` RPC methods with increasing values. The number for `value2` is intentionally set below the minimum threshold so that that will cause an `outofrange_value2` signal to be emitted. For each set, the `changed_value` signals will also be emitted. Whenever the client will receive either of the value change signals, it will do an `getvalue` RPC method call to retrieve the current value and print it out.

This will continue until the client is terminated.

## 1.5 Tracing D-Bus signals

Sometimes it's useful to see which signals are actually carried on the buses, especially when adding signal handlers for signals that are emitted from undocumented interfaces. The `dbus-monitor` tool will attach to the D-Bus daemon and ask it to watch for signals and report them back to the tool, so that it can decode the signals automatically as they appear on the bus.

While the server and client are still running, we next start the `dbus-monitor` (in a separate session this time) to see whether the signals are transmitted correctly. You should note that signals will be appear on the bus even if there are no clients currently interested in them. In our case signals are emitted by the server based on client issued RPC methods, so if you terminate the client, signals will cease.

```
[sbox-DIABLO_X86: ~/glib-dbus-signals] > run-standalone.sh dbus-monitor type='signal'
signal sender=:1.38 -> dest=(null destination)
 interface=org.maemo.Value; member=changed_value1
 string "value1"
signal sender=:1.38 -> dest=(null destination)
 interface=org.maemo.Value; member=changed_value2
 string "value2"
signal sender=:1.38 -> dest=(null destination)
 interface=org.maemo.Value; member=outofrange_value2
 string "value2"
signal sender=:1.38 -> dest=(null destination)
 interface=org.maemo.Value; member=changed_value1
 string "value1"
signal sender=:1.38 -> dest=(null destination)
 interface=org.maemo.Value; member=changed_value2
 string "value2"
signal sender=:1.38 -> dest=(null destination)
 interface=org.maemo.Value; member=outofrange_value2
 string "value2"
signal sender=:1.38 -> dest=(null destination)
 interface=org.maemo.Value; member=changed_value1
 string "value1"
signal sender=:1.38 -> dest=(null destination)
 interface=org.maemo.Value; member=changed_value2
 string "value2"
signal sender=:1.38 -> dest=(null destination)
 interface=org.maemo.Value; member=changed_value1
 string "value1"
signal sender=:1.38 -> dest=(null destination)
 interface=org.maemo.Value; member=changed_value2
 string "value2"
```

Running `dbus-monitor` to follow what signals are passed when the client runs

The tool will automatically decode the parameters to best of its ability (the `string` parameter for the signals above). It does not know the semantic meaning for the different signals, so sometimes you'll need to do some additional testing to decide what they actually mean. This is especially true when mapping out undocumented interfaces (for which you might not have the source code).

Some examples of displaying signals on the system bus on a device follow:

```
Nokia-N810-42-18:~# run-standalone.sh dbus-monitor --system
signal sender=:1.3 -> dest=(null destination) path=/com/nokia/mce/signal;
 interface=com.nokia.mce.signal; member=display_status_ind
 string "dimmed"
signal sender=:1.3 -> dest=(null destination) path=/com/nokia/mce/signal;
 interface=com.nokia.mce.signal; member=system_inactivity_ind
 boolean true
signal sender=:1.3 -> dest=(null destination) path=/com/nokia/mce/signal;
 interface=com.nokia.mce.signal; member=save_unsaved_data_ind
signal sender=:1.3 -> dest=(null destination) path=/com/nokia/mce/signal;
 interface=com.nokia.mce.signal; member=display_status_ind
 string "off"
```

A device turning off the backlight after inactivity

```
signal sender=:1.3 -> dest=(null destination) path=/com/nokia/mce/signal;
 interface=com.nokia.mce.signal; member=system_inactivity_ind
 boolean false
signal sender=:1.3 -> dest=(null destination) path=/com/nokia/mce/signal;
 interface=com.nokia.mce.signal; member=display_status_ind
 string "on"
signal sender=:1.3 -> dest=(null destination) path=/com/nokia/mce/signal;
 interface=com.nokia.mce.signal; member=tklock_mode_ind
 string "unlocked"
```

A device coming back to life after a screen tap

```
signal sender=:1.0 -> dest=(null destination)
 path=/org/freedesktop/Hal/devices/computer_logicaldev_input_0;
 interface=org.freedesktop.Hal.Device; member=Condition
   string "ButtonPressed"
   string "power"
signal sender=:1.3 -> dest=(null destination) path=/com/nokia/mce/signal;
 interface=com.nokia.mce.signal; member=sig_device_mode_ind
   string "flight"
signal sender=:1.26 -> dest=(null destination)
 path=/com/nokia/wlancond/signal;
 interface=com.nokia.wlancond.signal; member=disconnected
   string "wlan0"
signal sender=:1.28 -> dest=(null destination) path=/com/nokia/icd;
 interface=com.nokia.icd; member=status_changed
   string "SSID"
   string "WLAN_INFRA"
   string "DISCONNECTING"
   string "com.nokia.icd.error.network_error"
signal sender=:1.22 -> dest=(null destination) path=/org/bluez/hci0;
 interface=org.bluez.Adapter; member=ModeChanged
   string "off"
signal sender=:1.25 -> dest=(null destination)
 path=/com/nokia/btcond/signal;
 interface=com.nokia.btcond.signal; member=hci_dev_down
   string "hci0"
```

A device going into offline mode

```
signal sender=:1.0 -> dest=(null destination)
 path=/org/freedesktop/Hal/devices/computer_logicaldev_input_0;
 interface=org.freedesktop.Hal.Device; member=Condition
   string "ButtonPressed"
   string "power"
signal sender=:1.3 -> dest=(null destination) path=/com/nokia/mce/signal;
 interface=com.nokia.mce.signal; member=sig_device_mode_ind
   string "normal"
signal sender=:1.39 -> dest=(null destination)
 path=/org/kernel/class/firmware/hci_h4p;
 interface=org.kernel.kevent; member=add
signal sender=:1.39 -> dest=(null destination)
 path=/org/kernel/class/firmware/hci_h4p;
 interface=org.kernel.kevent; member=remove
signal sender=:1.25 -> dest=(null destination)
 path=/com/nokia/btcond/signal;
 interface=com.nokia.btcond.signal; member=hci_dev_up
   string "hci0"
signal sender=:1.22 -> dest=(null destination) path=/org/bluez/hci0;
 interface=org.bluez.Adapter; member=ModeChanged
   string "connectable"
signal sender=:1.22 -> dest=(null destination) path=/org/bluez/hci0;
 interface=org.bluez.Adapter; member=NameChanged
   string "Nokia N810"
signal sender=:1.28 -> dest=(null destination) path=/com/nokia/icd;
 interface=com.nokia.icd; member=status_changed
   string "SSID"
   string "WLAN_INFRA"
   string "CONNECTING"
   string ""
signal sender=:1.26 -> dest=(null destination)
 path=/com/nokia/wlancond/signal;
 interface=com.nokia.wlancond.signal; member=connected
   string "wlan0"
   array [
      byte 0
      byte 13
      byte 157
      byte 198
      byte 120
      byte 175
   ]
   int32 536870912
signal sender=:1.100 -> dest=(null destination) path=/com/nokia/eap/signal;
 interface=com.nokia.eap.signal; member=auth_status
   uint32 4
signal sender=:1.28 -> dest=(null destination) path=/com/nokia/icd;
 interface=com.nokia.icd; member=proxies
   uint32 0
   string ""
   int32 0
   string ""
   int32 0
   string ""
   int32 0
   string ""
   int32 0
   string ""
   int32 0
   array [ ]
   string ""
signal sender=:1.28 -> dest=(null destination) path=/com/nokia/icd;
 interface=com.nokia.icd; member=status_changed
   string "SSID"
   string "WLAN_INFRA"
   string "CONNECTED"
   string ""
```

A device going back into normal mode.

It is also possible to send signals from the command line, which is useful when you want to emulate some feature of a device inside the SDK. This (like

RPC method calls) can be done with the `dbus-send` tool and an example of this kind of simulation was given in the LibOSSO chapter.