

Maemo Diablo Source code for the GLib
D-Bus signal example
Training Material

February 9, 2009

Contents

1	Source code for the GLib D-Bus signal example	2
1.1	glib-dbus-signals/common-defs.h	2
1.2	glib-dbus-signals/value-dbus-interface.xml	3
1.3	glib-dbus-signals/server.c	4
1.4	glib-dbus-signals/client.c	15
1.5	glib-dbus-signals/Makefile	20

Chapter 1

Source code for the GLib D-Bus signal example

1.1 glib-dbus-signals/common-defs.h

```
#ifndef INCLUDE_COMMON_DEFS_H
#define INCLUDE_COMMON_DEFS_H
/**
 * This maemo code example is licensed under a MIT-style license,
 * that can be found in the file called "License" in the same
 * directory as this file.
 * Copyright (c) 2007-2008 Nokia Corporation. All rights reserved.
 *
 * This file includes the common symbolic defines for both client and
 * the server. Normally this kind of information would be part of the
 * object usage documentation, but in this example we take the easy
 * way out.
 *
 * To re-iterate: You could just as easily use strings in both client
 * and server, and that would be the more common way.
 */

/* Well-known name for this service. */
#define VALUE_SERVICE_NAME "org.maemo.Platdev_ex"
/* Object path to the provided object. */
#define VALUE_SERVICE_OBJECT_PATH "/GlobalValue"
/* And we're interested in using it through this interface.
   This must match the entry in the interface definition XML. */
#define VALUE_SERVICE_INTERFACE "org.maemo.Value"

/* 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"

#endif
```

Listing 1.1: glib-dbus-signals/common-defs.h

1.2 glib-dbus-signals/value-dbus-interface.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- This maemo code example is licensed under a MIT-style license,
that can be found in the file called "License" in the same
directory as this file.
Copyright (c) 2007-2008 Nokia Corporation. All rights reserved. --
>

<!-- If you keep the following DOCTYPE tag in your interface
specification, xmllint can fetch the DTD over the Internet
for validation automatically. -->
<!DOCTYPE node PUBLIC
"-//freedesktop//DTD D-Bus Object Introspection 1.0//EN"
"http://standards.freedesktop.org/dbus/1.0/introspect.dtd">

<!-- This file defines the D-Bus interface for a simple object, that
will hold a simple state consisting of two values (one a 32-bit
integer, the other a double).

The object will always generate a signal when a value is changed
(changed_value1 or changed_value2).

It has also a min and max thresholds: when a client tries to
set the value too high or too low, the object will generate a
signal (outofrange_value1 or outofrange_value2).

The thresholds are not modifiable (nor viewable) via this
interface. They are specified in integers and apply to both
internal values. Adding per-value thresholds would be a good
idea. Generalizing the whole interface to support multiple
concurrent values would be another good idea.

The interface name is "org.maemo.Value".
One known reference implementation is provided for it by the
"/GlobalValue" object found via a well-known name of
"org.maemo.Platdev_ex". -->

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

    <!-- Method definitions -->

    <!-- getvalue1(): returns the first value (int) -->
    <method name="getvalue1">
      <!-- NOTE Naming arguments is not mandatory, but is recommended
so that D-Bus introspection tools are more useful.
Otherwise the arguments will be automatically named
"arg0", "arg1" and so on. -->
      <arg type="i" name="cur_value" direction="out"/>
    </method>

    <!-- getvalue2(): returns the second value (double) -->
    <method name="getvalue2">
      <arg type="d" name="cur_value" direction="out"/>
    </method>

    <!-- setvalue1(int newValue): sets value1 -->
    <method name="setvalue1">
      <arg type="i" name="new_value" direction="in"/>
    </method>
  </interface>
</node>
```

```

<!-- setvalue2(double newValue): sets value2 -->
<method name="setvalue2">
  <arg type="d" name="new_value" direction="in"/>
</method>

<!-- 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 1.2: glib-dbus-signals/value-dbus-interface.xml

1.3 glib-dbus-signals/server.c

```

/**
 * This program implements a GObject for a simple class, then
 * registers the object on the D-Bus and starts serving requests.
 *
 * This maemo code example is licensed under a MIT-style license,
 * that can be found in the file called "License" in the same
 * directory as this file.
 * Copyright (c) 2007-2008 Nokia Corporation. All rights reserved.
 *
 * In more complicated code, the GObject definition, implementation
 * and D-Bus registration would all live in separate files. In this
 * server, they're all present in this file.
 *
 * This program will pull the automatically generated D-Bus/GLib stub
 * code (which contains the marshaling code as well as a binding/call
 * table.
 *
 * This program also might serve as an introduction into implementing

```

```

* custom GType/GObjects, but it is not the primary purpose here.
* Important things like object life-time management (freeing of
* objects), sub-classing and GObject properties are not covered at
* all.
*
* Demonstrates a simple implementation of "tracing" as well (via the
* NO_DAEMON define, as when built as non-daemonizing version, will
* output information about what is happening and where. Adding
* timestamps to each trace message is left out (see gettimeofday()
* system call for a simple solution).
*/

#include <glib.h>
#include <dbus/dbus-glib.h>
#include <stdlib.h> /* exit, EXIT_FAILURE */
#include <unistd.h> /* daemon */

/* Pull symbolic constants that are shared (in this example) between
the client and the server. */
#include "common-defs.h"

/**
 * 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;

typedef struct {
    /* The parent class object state. */
    GObject parent;
    /* Our first per-object state variable. */
    gint value1;
    /* Our second per-object state variable. */
    gdouble value2;
} ValueObject;

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;

/* Forward declaration of the function that will return the GType of

```

```

    the Value implementation. Not used in this program. */
GType value_object_get_type(void);

/* Macro for the above. It is common to define macros using the
naming convention (seen below) for all GType implementations,
and that's why we're going to do that here as well. */
#define VALUE_TYPE_OBJECT                (value_object_get_type())

#define VALUE_OBJECT(object) \
(G_TYPE_CHECK_INSTANCE_CAST ((object), \
VALUE_TYPE_OBJECT, ValueObject))
#define VALUE_OBJECT_CLASS(klass) \
(G_TYPE_CHECK_CLASS_CAST ((klass), \
VALUE_TYPE_OBJECT, ValueObjectClass))
#define VALUE_IS_OBJECT(object) \
(G_TYPE_CHECK_INSTANCE_TYPE ((object), \
VALUE_TYPE_OBJECT))
#define VALUE_IS_OBJECT_CLASS(klass) \
(G_TYPE_CHECK_CLASS_TYPE ((klass), \
VALUE_TYPE_OBJECT))
#define VALUE_OBJECT_GET_CLASS(obj) \
(G_TYPE_INSTANCE_GET_CLASS ((obj), \
VALUE_TYPE_OBJECT, ValueObjectClass))

G_DEFINE_TYPE(ValueObject, value_object, G_TYPE_OBJECT)

/**
 * Since the stub generator will reference the functions from a call
 * table, the functions must be declared before the stub is included.
 */
gboolean value_object_getvalue1(ValueObject* obj, gint* value_out,
                                GError** error);
gboolean value_object_getvalue2(ValueObject* obj, gdouble* value_out,
                                GError** error);
gboolean value_object_setvalue1(ValueObject* obj, gint value_in,
                                GError** error);
gboolean value_object_setvalue2(ValueObject* obj, gdouble value_in,
                                GError** error);

/**
 * Pull in the stub for the server side.
 */
#include "value-server-stub.h"

/* A small macro that will wrap g_print and expand to empty when
server will daemonize. We use this to add debugging info on
the server side, but if server will be daemonized, it doesn't
make sense to even compile the code in.

The macro is quite "hairy", but very convenient. */
#ifdef NO_DAEMON
#define dbg(fmtstr, args...) \
(g_print(PROGNAME ":%s: " fmtstr "\n", __func__, ##args))
#else
#define dbg(dummy...)
#endif

/**
 * Per object initializer
 *
 * Only sets up internal state (both values set to zero)
 */

```

```

static void value_object_init(ValueObject* obj) {
    dbg("Called");
    g_assert(obj != NULL);
    obj->value1 = 0;
    obj->value2 = 0.0;
}

/**
 * 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. */
    class->signals[i] = signalId;

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

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

/* Time to bind this GType into the GLib/D-Bus wrappers.
NOTE: This is not yet "publishing" the object on the D-Bus, but
since it is only allowed to do this once per class
creation, the safest place to put it is in the class
initializer.
Specifically, this function adds "method introspection
data" to the class so that methods can be called over
the D-Bus. */
dbus_g_object_type_install_info(VALUE_TYPE_OBJECT,
                                &dbus_glib_value_object_info);

dbg("Done");
/* All done. Class is ready to be used for instantiating objects */
}

/**
 * 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. */
}

/**
 * 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));
}

/**
 * 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.)
 *
 * NOTE: If you change the name of this function, the generated
 * stubs will no longer find it! On the other hand, if you
 * decide to modify the interface XML, this is one of the places
 * that you'll have to modify as well.

```

```

/* This applies to the next four functions (including this one). */
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;
}

/**
 * Function that gets executed on "setvalue2".
 * Other than this function operating with different type input
 * parameter (and different internal value), all the comments from
 * set_value1 apply here as well.
 */
gboolean value_object_setvalue2(ValueObject* obj, gdouble valueIn,
                                GError** error) {

    dbg("Called (valueIn=%.3f)", valueIn);

    g_assert(obj != NULL);

    /* Normally comparing doubles against other doubles is a bad idea,
       since multiple values can "collide" into one binary
       representation. In our case, it is not a real problem, as we're
       not interested in numeric comparison, but testing whether the
       binary content is about to change. Also, as the value has been
       sent by client over the D-Bus, it has already been reduced. */
    if (obj->value2 != valueIn) {
        obj->value2 = valueIn;

        value_object_emitSignal(obj, E_SIGNAL_CHANGED_VALUE2, "value2");

        if (!value_object_thresholdsOk(obj, valueIn)) {
            value_object_emitSignal(obj, E_SIGNAL_OUTOFRANGE_VALUE2,
                                    "value2");
        }
    }
    return TRUE;
}

```

```

/**
 * Function that gets executed on "getvalue1".
 * We don't signal the get operations, so this will be simple.
 */
gboolean value_object_getvalue1(ValueObject* obj, gint* valueOut,
                                GError** error) {

    dbg("Called (internal value1 is %d)", obj->value1);

    g_assert(obj != NULL);

    /* Check that the target pointer is not NULL.
     Even is the only caller for this will be the GLib-wrapper code,
     we cannot trust the stub generated code and should handle the
     situation. We will terminate with an error in this case.

     Another option would be to create a new GError, and store
     the error condition there. */
    g_assert(valueOut != NULL);

    /* Copy the current first value to caller specified memory. */
    *valueOut = obj->value1;

    /* Return success. */
    return TRUE;
}

/**
 * Function that gets executed on "getvalue2".
 * (Again, similar to "getvalue1").
 */
gboolean value_object_getvalue2(ValueObject* obj, gdouble* valueOut,
                                GError** error) {

    dbg("Called (internal value2 is %.3f)", obj->value2);

    g_assert(obj != NULL);
    g_assert(valueOut != NULL);

    *valueOut = obj->value2;
    return TRUE;
}

/**
 * Print out an error message and optionally quit (if fatal is TRUE)
 */
static void handleError(const char* msg, const char* reason,
                        gboolean fatal) {
    g_printerr(PROGNAME ": ERROR: %s (%s)\n", msg, reason);
    if (fatal) {
        exit(EXIT_FAILURE);
    }
}

/**
 * The main server code
 *
 * 1) Init GType/GObject
 * 2) Create a mainloop
 * 3) Connect to the Session bus
 * 4) Get a proxy object representing the bus itself
 * 5) Register the well-known name by which clients can find us.
 * 6) Create one Value object that will handle all client requests.

```

```

* 7) Register it on the bus (will be found via "/GlobalValue" object
*     path)
* 8) Daemonize the process (if not built with NO_DAEMON)
* 9) Start processing requests (run GMainLoop)
*
* This program will not exit (unless it encounters critical errors).
*/
int main(int argc, char** argv) {
    /* The GObject representing a D-Bus connection. */
    DBusGConnection* bus = NULL;
    /* Proxy object representing the D-Bus service object. */
    DBusGProxy* busProxy = NULL;
    /* Will hold one instance of ValueObject that will serve all the
    requests. */
    ValueObject* valueObj = NULL;
    /* GMainLoop for our program. */
    GMainLoop* mainloop = NULL;
    /* Will store the result of the RequestName RPC here. */
    guint result;
    GError* error = NULL;

    /* Initialize the GType/GObject system. */
    g_type_init();

    /* Create a main loop that will dispatch callbacks. */
    mainloop = g_main_loop_new(NULL, FALSE);
    if (mainloop == NULL) {
        /* Print error and terminate. */
        handleError("Couldn't create GMainLoop", "Unknown(OOM?)", TRUE);
    }

    g_print(PROGNAME ":main Connecting to the Session D-Bus.\n");
    bus = dbus_g_bus_get(DBUS_BUS_SESSION, &error);
    if (error != NULL) {
        /* Print error and terminate. */
        handleError("Couldn't connect to session bus", error->message, TRUE
    );
    }

    g_print(PROGNAME ":main Registering the well-known name (%s)\n",
    VALUE_SERVICE_NAME);

    /* In order to register a well-known name, we need to use the
    "RequestMethod" of the /org/freedesktop/DBus interface. Each
    bus provides an object that will implement this interface.

    In order to do the call, we need a proxy object first.
    DBUS_SERVICE_DBUS = "org.freedesktop.DBus"
    DBUS_PATH_DBUS = "/org/freedesktop/DBus"
    DBUS_INTERFACE_DBUS = "org.freedesktop.DBus" */
    busProxy = dbus_g_proxy_new_for_name(bus,
    DBUS_SERVICE_DBUS,
    DBUS_PATH_DBUS,
    DBUS_INTERFACE_DBUS);

    if (busProxy == NULL) {
        handleError("Failed to get a proxy for D-Bus",
    "Unknown(dbus_g_proxy_new_for_name)", TRUE);
    }

    /* Attempt to register the well-known name.
    The RPC call requires two parameters:
    - arg0: (D-Bus STRING): name to request

```

```

- arg1: (D-Bus UINT32): flags for registration.
  (please see "org.freedesktop.DBus.RequestName" in
  http://dbus.freedesktop.org/doc/dbus-specification.html)
Will return one uint32 giving the result of the RPC call.
We're interested in 1 (we're now the primary owner of the name)
or 4 (we were already the owner of the name, however in this
application it wouldn't make much sense).

The function will return FALSE if it sets the GError. */
if (!dbus_g_proxy_call(busProxy,
    /* Method name to call. */
    "RequestName",
    /* Where to store the GError. */
    &error,
    /* Parameter type of argument 0. Note that
    since we're dealing with GLib/D-Bus
    wrappers, you will need to find a suitable
    GType instead of using the "native" D-Bus
    type codes. */
    G_TYPE_STRING,
    /* Data of argument 0. In our case, this is
    the well-known name for our server
    example ("org.maemo.Platdev_ex"). */
    VALUE_SERVICE_NAME,
    /* Parameter type of argument 1. */
    G_TYPE_UINT,
    /* Data of argument 0. This is the "flags"
    argument of the "RequestName" method which
    can be use to specify what the bus service
    should do when the name already exists on
    the bus. We'll go with defaults. */
    0,
    /* Input arguments are terminated with a
    special GType marker. */
    G_TYPE_INVALID,
    /* Parameter type of return value 0.
    For "RequestName" it is UINT32 so we pick
    the GType that maps into UINT32 in the
    wrappers. */
    G_TYPE_UINT,
    /* Data of return value 0. These will always
    be pointers to the locations where the
    proxy can copy the results. */
    &result,
    /* Terminate list of return values. */
    G_TYPE_INVALID)) {
    handleError("D-Bus.RequestName RPC failed", error->message,
        TRUE);
    /* Note that the whole call failed, not "just" the name
    registration (we deal with that below). This means that
    something bad probably has happened and there's not much we can
    do (hence program termination). */
}
/* Check the result code of the registration RPC. */
g_print(PROGNAME ":main RequestName returned %d.\n", result);
if (result != 1) {
    handleError("Failed to get the primary well-known name.",
        "RequestName result != 1", TRUE);
    /* In this case we could also continue instead of terminating.
    We could retry the RPC later. Or "lurk" on the bus waiting for
    someone to tell us what to do. If we would be publishing
    multiple services and/or interfaces, it even might make sense

```

```

        to continue with the rest anyway.

        In our simple program, we terminate. Not much left to do for
        this poor program if the clients won't be able to find the
        Value object using the well-known name. */
    }

    g_print(PROGNAME ":main Creating one Value object.\n");
    /* The NULL at the end means that we have stopped listing the
    property names and their values that would have been used to
    set the properties to initial values. Our simple Value
    implementation does not support GObject properties, and also
    doesn't inherit anything interesting from GObject directly, so
    there are no properties to set. For more examples on properties
    see the first GTK+ example programs from the Application
    Development material.

    NOTE: You need to keep at least one reference to the published
    object at all times, unless you want it to disappear from
    the D-Bus (implied by API reference for
    dbus_g_connection_register_g_object(). */
    valueObj = g_object_new(VALUE_TYPE_OBJECT, NULL);
    if (valueObj == NULL) {
        handleError("Failed to create one Value instance.",
            "Unknown(OOM?)", TRUE);
    }

    g_print(PROGNAME ":main Registering it on the D-Bus.\n");
    /* The function does not return any status, so can't check for
    errors here. */
    dbus_g_connection_register_g_object(bus,
        VALUE_SERVICE_OBJECT_PATH,
        G_OBJECT(valueObj));

    g_print(PROGNAME ":main Ready to serve requests (daemonizing).\n");
#ifdef NO_DAEMON
    /* This will attempt to daemonize this process. It will switch this
    process' working directory to / (chdir) and then reopen stdin,
    stdout and stderr to /dev/null. Which means that all printouts
    that would occur after this, will be lost. Obviously the
    daemonization will also detach the process from the controlling
    terminal as well. */
    if (daemon(0, 0) != 0) {
        g_error(PROGNAME ": Failed to daemonize.\n");
    }
#else
    g_print(PROGNAME
        ": Not daemonizing (built with NO_DAEMON-build define)\n");
#endif

    /* Start service requests on the D-Bus forever. */
    g_main_loop_run(mainloop);
    /* This program will not terminate unless there is a critical
    error which will cause abort() to be used. Hence it will never
    reach this point. */

    /* If it does, return failure exit code just in case. */
    return EXIT_FAILURE;
}

```

1.4 glib-dbus-signals/client.c

```
/**
 * A simple test program to test using of the Value object over the
 * D-Bus.
 *
 * This maemo code example is licensed under a MIT-style license,
 * that can be found in the file called "License" in the same
 * directory as this file.
 * Copyright (c) 2007-2008 Nokia Corporation. All rights reserved.
 *
 * It will first create a GLib/D-Bus proxy object connected to the
 * server's Value object, and then start firing away value set
 * methods.
 *
 * The methods are fired from a timer based callback once per second.
 * If the method calls fail (the ones that modify the values), the
 * program will still continue to run (so that the server can be
 * restarted if necessary).
 *
 * Also demonstrates listening to/catching of D-Bus signals (using the
 * GSignal mechanism, into which D-Bus signals are automatically
 * converted by the GLib-bindings.
 *
 * When value changed signals are received, will also request the
 * Value object for the current value and print that out.
 */

#include <glib.h>
#include <dbus/dbus-glib.h>
#include <stdlib.h> /* exit, EXIT_FAILURE */
#include <string.h> /* strcmp */

/* Pull the common symbolic defines. */
#include "common-defs.h"

/* Pull in the client stubs that were generated with
   dbus-binding-tool */
#include "value-client-stub.h"

/**
 * Print out an error message and optionally quit (if fatal is TRUE)
 */
static void handleError(const char* msg, const char* reason,
                       gboolean fatal) {
    g_printerr(PROGNAME ": ERROR: %s (%s)\n", msg, reason);
    if (fatal) {
        exit(EXIT_FAILURE);
    }
}

/**
 * Handles signals from Value-object when either value is outside
 * the thresholds (min or max).
 *
 * We use strcmp internally to differentiate between the values.
 */
```

```

* NOTE: This signal handler's parameters are "selected" by the server
* when it creates the signals, and filtered by the interface
* XML. Unfortunately dbus-binding-tool does not know how to
* enforce the argument specifications of signals (from XML),
* so you'll need to be careful to match this function with
* the exact signal creation (marshaling information).
*/
static void outOfRangeSignalHandler(DBusGProxy* proxy,
                                   const char* valueName,
                                   gpointer userData) {

    g_print(PROGNAME ":out-of-range (%s)!\n", valueName);
    if (strcmp(valueName, "value1") == 0) {
        g_print(PROGNAME ":out-of-range Value 1 is outside threshold\n");
    } else {
        g_print(PROGNAME ":out-of-range Value 2 is outside threshold\n");
    }
}

/**
 * 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);
}

/**
 * This function will be called repeatedly from within the mainloop
 * timer launch code.
 *
 * The function will start with two statically initialized variables
 * (int and double) which will be incremented after each time this
 * function runs and will use the setvalue* remote methods to set the
 * new values. If the set methods fail, program is not aborted, but an
 * message will be issued to the user describing the error.
 *
 * It will purposefully start value2 from below the default minimum
 * threshold (set in the server code).
 *
 * NOTE: There is a design issue in the implementation in the Value
 * object: it does not provide "adjust" method which would make
 * it possible for multiple clients to adjust the values,
 * instead of setting them separately. Now, if you launch
 * multiple clients (at different times), the object internal
 * values will end up fluctuating between the clients.
 */
static gboolean timerCallback(DBusGProxy* remoteobj) {

    /* Local values that we'll start updating to the remote object. */
    static gint localValue1 = -80;
    static gdouble localValue2 = -120.0;

    GError* error = NULL;

    /* Set the first value. */
    org_maemo_Value_setvalue1(remoteobj, localValue1, &error);
    if (error != NULL) {
        handleError("Failed to set value1", error->message, FALSE);
    } else {
        g_print(PROGNAME ":timerCallback Set value1 to %d\n", localValue1);
    }

    /* If there was an error with the first, release the error, and
     don't attempt the second time. Also, don't add to the local
     values. We assume that errors from the first set are caused by
     server going off the D-Bus, but are hopeful that it will come
     back, and hence keep trying (returning TRUE). */
    if (error != NULL) {
        g_clear_error(&error);
        return TRUE;
    }

    /* Now try to set the second value as well. */
    org_maemo_Value_setvalue2(remoteobj, localValue2, &error);
    if (error != NULL) {
        handleError("Failed to set value2", error->message, FALSE);
        g_clear_error(&error); /* Or g_error_free in this case. */
    } else {
        g_print(PROGNAME ":timerCallback Set value2 to %.3lf\n",
            localValue2);
    }
}

```

```

    /* Step the local values forward. */
    localValue1 += 10;
    localValue2 += 10.0;

    /* Tell the timer launcher that we want to remain on the timer
       call list in the future as well. Returning FALSE here would
       stop the launch of this timer callback. */
    return TRUE;
}

/**
 * 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) {
    /* The D-Bus connection object. Provided by GLib/D-Bus wrappers. */
    DBusGConnection* bus;
    /* This will represent the Value object locally (acting as a proxy
       for all method calls and signal delivery. */
    DBusGProxy* remoteValue;
    /* This will refer to the GMainLoop object */
    GMainLoop* mainloop;
    GError* error = NULL;

    /* Initialize the GType/GObject system. */
    g_type_init();

    /* Create a new GMainLoop with default context (NULL) and initial
       state of "not running" (FALSE). */
    mainloop = g_main_loop_new(NULL, FALSE);
    /* Failure to create the main loop is fatal (for us). */
    if (mainloop == NULL) {
        handleError("Failed to create the mainloop", "Unknown (OOM?)",
            TRUE);
    }

    g_print(PROGNAME ":main Connecting to Session D-Bus.\n");
    bus = dbus_g_bus_get(DBUS_BUS_SESSION, &error);
    if (error != NULL) {
        handleError("Couldn't connect to the Session bus", error->message,
            TRUE);
        /* Normally you'd have to also g_error_free() the error object
           but since the program will terminate within handleError,
           it is not necessary here. */
    }

    g_print(PROGNAME ":main Creating a GLib proxy object for Value.\n");

    /* Create the proxy object that we'll be using to access the object
       on the server. If you would use dbus_g_proxy_for_name_owner(),
       you would be also notified when the server that implements the
       object is removed (or rather, the interface is removed). Since
       we don't care who actually implements the interface, we'll use the

```

```

    more common function. See the API documentation at
    http://maemo.org/api_refs/4.0/dbus/ for more details. */
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 1.4: glib-dbus-signals/client.c

1.5 glib-dbus-signals/Makefile

```

# This is a relatively simplistic Makefile suitable for projects which
# use the GLib bindings for D-Bus.
#
# One extra target (which requires xmllint, from package libxml2-utils)
# is available to verify the well-formedness and the structure of the
# interface definition xml file.
#
# Use the 'checkxml' target to run the interface XML through xmllint
# verification. You'll need to be connected to the Internet in order
# for xmllint to retrieve the DTD from fd.o (unless you setup local
# catalogs, which are not covered here).
#
# If you want to make the server daemonized, please see below for the

```

```

# 'NO_DAEMON' setting. Commenting that line will disabling tracing in
# the server AND make it into a daemon.

# Interface XML name (used in multiple targets)
interface_xml := value-dbus-interface.xml

# Define a list of pkg-config packages we want to use
pkg_packages := dbus-1 dbus-glib-1

PKG_CFLAGS := $(shell pkg-config --cflags $(pkg_packages))
PKG_LDFLAGS := $(shell pkg-config --libs $(pkg_packages))
# Add additional flags:
# -g : add debugging symbols
# -Wall : enable most gcc warnings
# -DG_DISABLE_DEPRECATED : disable GLib functions marked as deprecated
ADD_CFLAGS := -g -Wall -DG_DISABLE_DEPRECATED
# -DNO_DAEMON : do not daemonize the server (on a separate line so can
# be disabled just by commenting the line)
ADD_CFLAGS += -DNO_DAEMON

# Combine flags
CFLAGS := $(PKG_CFLAGS) $(ADD_CFLAGS) $(CFLAGS)
LDFLAGS := $(PKG_LDFLAGS) $(LDFLAGS)

# Define a list of generated files so that they can be cleaned as well
cleanfiles := value-client-stub.h \
              value-server-stub.h

targets = server client

.PHONY: all clean checkxml
all: $(targets)

# We don't use the implicit pattern rules built into GNU make, since
# they put the LDFLAGS in the wrong location (and linking consequently
# fails sometimes).
#
# NOTE: You could actually collapse the compilation and linking phases
# together, but this arrangement is much more common.

server: server.o
$(CC) $^ -o $@ $(LDFLAGS)

client: client.o
$(CC) $^ -o $@ $(LDFLAGS)

# The server and client depend on the respective implementation source
# files, but also on the common interface as well as the generated
# stub interfaces.
server.o: server.c common-defs.h value-server-stub.h
$(CC) $(CFLAGS) -DPROGRAMNAME="\$(basename $@)" -c $< -o $@

client.o: client.c common-defs.h value-client-stub.h
$(CC) $(CFLAGS) -DPROGRAMNAME="\$(basename $@)" -c $< -o $@

# If the interface XML changes, the respective stub interfaces will be
# automatically regenerated. Normally this would also mean that your
# builds would fail after this since you'd be missing implementation
# code.
value-server-stub.h: $(interface_xml)
dbus-binding-tool --prefix=value_object --mode=glib-server \
$< > $@

```

```

value-client-stub.h: $(interface_xml)
    dbus-binding-tool --prefix=value_object --mode=glib-client \
    $< > $@

# Special target to run DTD validation on the interface XML. Not run
# automatically (since xmllint isn't always available and also needs
# Internet connectivity).
checkxml: $(interface_xml)
    @xmllint --valid --noout $<
    @echo $< checks out ok

clean:
    $(RM) $(targets) $(cleanfiles) *.o

# In order to force a rebuild if this file is modified, we add the
# Makefile as a dependency to all low-level targets. Adding the same
# dependency to multiple files on the same line is allowed in GNU make
# syntax as follows. Just make sure that additional dependencies are
# listed after explicit rules, or that no implicit pattern rules will
# match the dependency. Otherwise funny things happen. Placing the
# Makefile dependency at the very end is often the safest solution.
server.o client.o: Makefile

```

Listing 1.5: glib-dbus-signals/Makefile