Maemo Diablo Source code for the GLib D-Bus asynchronous examples

Training Material

February 9, 2009

Contents

Chapter 1

Source code for the GLib D-Bus asynchronous examples

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

1.2 glib-dbus-async/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>

    <!-- 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-async/value-dbus-interface.xml

1.3 glib-dbus-async/server.c

```
/**
 * Same server as in glib-dbus-signals, but with a 5 second sleep
 * between each client operation (in order to demonstrate the benefits
 * of using the async interface at the client end).
 *
 * 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.
 */

#include <glib.h>
```

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

/* How many microseconds to delay between each client operation. */
#define SERVER_DELAY_USEC (5*1000000UL)

/* 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 =
                  /* str name of the signal */
      g_signal_new(signalNames[i],
                  /* 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");

  /* 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_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 marshaling 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);

  dbg("Delaying operation");
```

```
  g_usleep(SERVER_DELAY_USEC);

  /* 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);

  dbg("Delaying operation");
  g_usleep(SERVER_DELAY_USEC);

  /* 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);

  dbg("Delaying operation");
  g_usleep(SERVER_DELAY_USEC);

  /* 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);

  dbg("Delaying operation");
  g_usleep(SERVER_DELAY_USEC);

  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
     requsts. */
  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");

#ifndef 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;
}
```

Listing 1.3: glib-dbus-async/server.c

1.4 glib-dbus-async/client-stubs.c

```
/**
 * An approach for issuing RPC method calls using the
 * dbus-binding-tool generated wrappers.
 *
 * 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 order to shorten the code, the code for handling signals has
 * been removed as well as code dealing with value2.
 *
 * This program also demonstrates the inherent complexity when moving
 * from sync to async calls (more problem scenarios to handle). As
 * such, it is NOT suitable to be copy-pasted!
 */

#include <glib.h>
#include <dbus/dbus-glib.h>
#include <stdlib.h>   /* exit */
#include <sys/time.h> /* struct timeval and friends */
#include <time.h>     /* gettimeofday */

/* 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);
  }
}

/**
 * Utility to return a pointer to a statically allocated buffer that
 * holds the text representation of seconds since this program was
 * started. Not safe to use in threaded programs!
 */
static const gchar* timestamp(void) {
  /* Holds the "seconds since starting" string. */
  static gchar buffer[200] = {};
  /* Holds the starting timestamp. 0 means that it has not been
     initialized. */
  static guint64 startTimestamp = 0;
  /* Holds the current timestamp. */
  guint64 curTimestamp = 0;
```

```
  /* Temp storage for the secs + microseconds time. */
  struct timeval tv;
  /* Temp storage for the difference between start and now. */
  guint64 delta;

  /* Get current time and convert into microseconds flat. */
  gettimeofday(&tv, NULL);
  /* Convert into microseconds. */
  curTimestamp = (guint64)tv.tv_usec + ((guint64)tv.tv_sec)*1000000ULL;

  /* Running for the first time? */
  if (startTimestamp == 0) {
    /* Copy to prev so that we get 0 delta. */
    startTimestamp = curTimestamp;
  }

  /* Calculate the delta (in microseconds). */
  delta = curTimestamp - startTimestamp;

  /* Create the string giving offset from start in seconds. */
  g_snprintf(buffer, sizeof(buffer), "%2u.%.2u",
    (guint)delta / 1000000,
    ((guint)delta % 1000000) / 10000);

  /* Return pointer to the buffer (will always return a pointer to the
     same location. */
  return buffer;
}

/**
 * This function will be called when the async setvalue1 will either
 * complete, timeout or fail (our server however does not signal
 * errors, but the client D-Bus library might). When this example
 * program is left running for a while, you will see all three cases.
 *
 * The prototype must match the one generated by the dbus-binding-tool
 * (org_maemo_Value_setvalue1_reply).
 *
 * Since there is no return value from the RPC, the only useful
 * parameter that we get is the error object, which we'll check.
 * If error is NULL, that means no error. Otherwise the RPC call
 * failed and we should check what the cause was.
 */
static void setValue1Completed(DBusGProxy* proxy, GError *error,
                                                  gpointer userData) {

  g_print(PROGNAME ":%s:setValue1Completed\n", timestamp());
  if (error != NULL) {
    g_printerr(PROGNAME "       ERROR: %s\n", error->message);
    /* We need to release the error object since the stub code does
       not do it automatically. */
    g_error_free(error);
  } else {
    g_print(PROGNAME "       SUCCESS\n");
  }
}

/**
 * This function will be called repeatedly from within the mainloop
 * timer launch code.
 *
 * It will launch asynchronous RPC method to set value1 with ever
```

```
 * increasing argument.
 */
static gboolean timerCallback(DBusGProxy* remoteobj) {

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

  /* Start the RPC.
     This is done by calling the stub function that will take the new
     value and the callback function to call on reply getting back.

     The stub returns a DBusGProxyCall object, but we don't need it
     so we'll ignore the return value. The return value could be used
     to cancel a pending request (from client side) with
     dbus_g_proxy_cancel_call. We could also pass a pointer to
     user-data (last parameter), but we don't need one in this example.
     It would normally be used to "carry around" the application state.
     */
  g_print(PROGNAME ":%s:timerCallback launching setvalue1\n",
          timestamp());
  org_maemo_Value_setvalue1_async(remoteobj, localValue1,
                                  setValue1Completed, NULL);
  g_print(PROGNAME ":%s:timerCallback setvalue1 launched\n",
          timestamp());

  /* Step the local value forward. */
  localValue1 += 10;

  /* Repeat timer later. */
  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) Start a timer that will launch timerCallback once per second.
 * 6) Run main-loop (forever)
 */
int main(int argc, char** argv) {
  DBusGConnection* bus;
  /* The proxy object. */
  DBusGProxy* remoteValue;
  GMainLoop* mainloop;
  GError* error = NULL;

  g_type_init();

  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);
```

```
  }

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

  g_print(PROGNAME
          ":%s:main Starting main loop (first timer in 1s).\n",
          timestamp());
  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 1;
}
```

Listing 1.4: glib-dbus-async/client-stubs.c

1.5 glib-dbus-async/client-glib.c

```
/**
 * Same example as client-stubs, but implementing the asynchronous
 * logic by using GLib/D-Bus wrapper functions directly. This version
 * of the asynchronous client does not require the generated stubs.
 *
 * 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.
 */

#include <glib.h>
#include <dbus/dbus-glib.h>
#include <stdlib.h> /* exit */

/* Pull the common symbolic defines. */
#include "common-defs.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);
  }
}
```

```
/**
 * This function will be called when the async setvalue1 will either
 * complete, timeout or fail (same as before). The main difference in
 * using GLib/D-Bus wrappers is that we need to "collect" the return
 * value (or error). This is done with the _end_call function.
 *
 * Note that all callbacks that are to be registered for RPC async
 * notifications using dbus_g_proxy_begin_call must follow the
 * following prototype: DBusGProxyCallNotify .
 */
static void setValue1Completed(DBusGProxy* proxy,
                               DBusGProxyCall* call,
                               gpointer userData) {

  /* This will hold the GError object (if any). */
  GError* error = NULL;

  g_print(PROGNAME ":setValue1Completed\n");

  /* We next need to collect the results from the RPC call.
     The function returns FALSE on errors (which we check), although
     we could also check whether error-ptr is still NULL. */
  if (!dbus_g_proxy_end_call(proxy,
                             /* The call that we're collecting. */
                             call,
                             /* Where to store the error (if any). */
                             &error,
                             /* Next we list the GType codes for all
                                the arguments we expect back. In our
                                case there are none, so set to
                                invalid. */
                             G_TYPE_INVALID)) {
    /* Some error occurred while collecting the result. */
    g_printerr(PROGNAME " ERROR: %s\n", error->message);
    g_error_free(error);
  } else {
    g_print(PROGNAME " SUCCESS\n");
  }
}

/**
 * This function will be called repeatedly from within the mainloop
 * timer launch code.
 *
 * It will launch asynchronous RPC method to set value1 with ever
 * increasing argument.
 */
static gboolean timerCallback(DBusGProxy* remoteobj) {

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

  /* Start the first RPC.
     The call using GLib/D-Bus is only slightly more complex than the
     stubs. The overall operation is the same. */
  g_print(PROGNAME ":timerCallback launching setvalue1\n");
  dbus_g_proxy_begin_call(remoteobj,
                          /* Method name. */
                          "setvalue1",
                          /* Callback to call on "completion". */
                          setValue1Completed,
```

```
                          /* User-data to pass to callback. */
                          NULL,
                          /* Function to call to free userData after
                             callback returns. */
                          NULL,
                          /* First argument GType. */
                          G_TYPE_INT,
                          /* First argument value (passed by value) */
                          localValue1,
                          /* Terminate argument list. */
                          G_TYPE_INVALID);
  g_print(PROGNAME ":timerCallback setvalue1 launched\n");

  /* Step the local value forward. */
  localValue1 += 10;

  /* Repeat timer later. */
  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) Start a timer that will launch timerCallback once per second.
 * 6) Run main-loop (forever)
 */
int main(int argc, char** argv) {
  DBusGConnection* bus;
  /* The proxy object. */
  DBusGProxy* remoteValue;
  GMainLoop* mainloop;
  GError* error = NULL;

  g_type_init();

  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);
  }

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

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

Listing 1.5: glib-dbus-async/client-glib.c

1.6 glib-dbus-async/Makefile

```
# Similar Makefile than before for the synchronous glib-dbus-example,
# but this time for two clients. client-stubs uses the generated stub
# functions, client-glib uses the GLib wrapper functions. Comment
# verbosity has also been reduced.

# 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

# Get compilation flags for necessary libraries from pkg-config
PKG_CFLAGS  := $(shell pkg-config --cflags $(pkg_packages))
# Get linking flags
PKG_LDFLAGS := $(shell pkg-config --libs $(pkg_packages))

# Add debugging, full warnings and GLib deprecation
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 user supplied, additional and pkg-config 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-stubs client-glib

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

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

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

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

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

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

# Note that the GLib client doesn't need the stub code.
client-glib.o: client-glib.c common-defs.h
  $(CC) $(CFLAGS) -DPROGNAME=\"$(basename $@)\" -c $< -o $@

# The stub header dependencies
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 \
    $< > $@

# XML interface validation
checkxml: $(interface_xml)
  @xmllint --valid --noout $<
  @echo $< checks out ok

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

# Changing Makefile will cause rebuild
server.o client-stubs.o clients-glib.o: Makefile
```

Listing 1.6: glib-dbus-async/Makefile