# Maemo Diablo Platform Development

# Training Material

# for maemo 4.1

February 9, 2009

# Contents

# Preface

## Legal notice



## Disclaimer

The information in this document is provided "as is," with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. This document is provided for informational purposes only. Nokia Corporation disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information presented in this document. Nokia Corporation does not warrant or represent that such use will not infringe such rights. Nokia Corporation retains the right to make changes to this material at any time, without notice.

## Licenses

# Chapter 1

# Introduction

## 1.1 Introduction to Maemo Platform Development

D-Bus is an important middleware solution in the maemo™ platform that allows multiple processes to communicate with each other. A lot of existing services are already available using D-Bus, and this material will introduce you to the different ways that you can utilise these services from your programs. Implementing new services so that other programs may use them using D-Bus is equally important, and different techniques and approaches to do this are covered as well.

The material assumes knowledge of the C programming language, general Linux programming concepts and knowledge of the topics covered in the "maemo Getting Started" and "maemo Application Development" materials.

This version of the material covers maemo SDK version 4.1.x, Diablo.

More information about the maemo training material is available from maemo training wiki pages (http://wiki.maemo.org/Maemo_Training) maintained by maemo community. Notice that the information in maemo wiki is not verified by Nokia and thus Nokia cannot be responsible of that information.

# Chapter 2

# D-Bus, The Message Bus System

## 2.1 Introduction to D-Bus

D-Bus (the D originally stood for "Desktop") is a relatively new inter process communication (IPC) mechanism designed to be used as a unified middleware layer in free desktop environments. Some example projects where D-Bus is used are GNOME and Hildon. Compared to other middleware layers for IPC, D-Bus lacks many of the more refined (and complicated) features and for that reason, is faster and simpler.

D-Bus does not directly compete with low level IPC mechanisms like sockets, shared memory or message queues. Each of these mechanisms have their uses, which normally do not overlap the ones in D-Bus. Instead, D-Bus aims to provide higher level functionality, like:

- Structured name spaces
- Architecture independent data formatting
- Support for the most common data elements in messages
- A generic remote call interface with support for exceptions (errors)
- A generic signalling interface to support "broadcast" type communication
- Clear separation of per-user and system-wide scopes, which is important when dealing with multi-user systems
- Not bound to any specific programming language (while providing a design that readily maps to most higher level languages, via language specific bindings)

The design of D-Bus benefits from the long experience of using other middleware IPC solutions in the desktop arena and this has allowed the design to be optimised. It also doesn't yet suffer from "creeping featurism" (having extra features just to satisfy niche use cases).

All this said, the main problem area that D-Bus aims to solve is facilitating easy IPC between related (often graphical) desktop software applications.

D-Bus has a very important role in maemo, as it is the IPC mechanism to use when using the services provided in the platform (and devices). Providing services over D-Bus is also the easiest way to assure component re-use from other applications.

## 2.2 D-Bus architecture and terminology

In D-Bus, the *bus* is a central concept. It is the channel through which applications can do the method calls, send signals and listen to signals. There are two predefined buses: the *session bus* and the *system bus*.

- The session bus is meant for communication between applications that are connected to the same desktop session, and normally started and run by one user (using the same user identifier, or UID).
- The system bus is meant for communication when applications (or services) running with disparate sessions wish to communicate with each other. Most common use for this bus is sending system wide notifications when system wide events occur. Adding a new storage device, network connectivity change events and shutdown related events are all examples of when system bus would be the more suitable bus for communication.

Normally only one system bus will exist, but there might be several session buses (one per each desktop session). Since in Internet Tablets, all user applications will run with the same user id (`user`), there will only be one session bus as well.

A bus exists in the system in the form of a *bus daemon*, a process that specialises in passing messages from a process to another. The daemon will also forward notifications to all applications on the bus. At the lowest level, D-Bus only supports point-to-point communication, normally using the local domain sockets (`AF_UNIX`) between the application and the bus daemon. The point-to-point aspect of D-Bus is however abstracted by the bus daemon, which will implement addressing and message passing functionality so that each application doesn't need to care about which specific process will receive each method call or notification.

The above means that sending a message using D-Bus will always involve the following steps (under normal conditions):

- Creation and sending of the message to the bus daemon. This will cause at minimum two context switches.
- Processing of the message by the bus daemon and forwarding it to the target process. This will again cause at minimum two context switches.
- The target application will receive the message. Depending on the message type, it will either need to acknowledge it, respond with a reply or ignore it. The last case is only possible with notifications (i.e., *signals* in D-Bus terminology). Acknowledgement or replies will cause further context switches.

Coupled together, the above rules mean that if you plan to transfer large amounts of data between processes, D-Bus won't be the most efficient way to do it. The most efficient way would be using some kind of shared memory arrangement, but is often quite complex to implement correctly.

## 2.3 Addressing and names in D-Bus

In order for the messages to reach the intended recipient, the IPC mechanism needs to support some form of addressing. The addressing scheme in D-Bus has been designed to be flexible but at the same time efficient. Each bus has its private name space which is not directly related to any other bus.

In order to send a message, you will need an destination address and it is formed in a hierarchical manner from the following elements:

- The bus on which the message is to be sent. A bus is normally opened only once per application lifetime. One will then use the bus connection for sending and receiving messages for as long as necessary. This way, the target bus will form a transparent part of the message address (i.e., it is not specified separately for each message sent).
- The *well-known name* for the service provided by the recipient. A close analogy to this would be the DNS system in Internet, where people normally use names to connect to services, instead of specific IP addresses providing the services. The idea in D-Bus well-known names is very similar, since the same service might be implemented in different ways in different applications. It should be noted however that currently most of the existing D-Bus services are "unique" in that each of them provides their own well-known name, and replacing one implementation with another is not common.
  - A well-known name consists of A-Z characters (lower- or uppercase), dot characters, dashes and underscores. There must be at least two dot separated elements in the well-known name. Unlike DNS, the dots do not carry any additional information about management (zones) meaning that the well-known names are **NOT** hierarchical.
  - In order to reduce clashes in the D-Bus name space, it is recommended you form the name by reversing the order of labels of a DNS domain that you own. Similar approach is used in Java for package names.
  - Examples: `org.maemo.Alert` and `org.freedesktop.Notifications`.
- Each service can contain multiple different objects, each of which provides a different (or same) service. In order to separate one object from another, *object paths* are used. A PIM information store for example, might include separate objects to manage the contact information and synchronisation.
  - Object paths look like file paths (elements separated with the / -character).

  - In D-Bus, it is also possible to do "lazy binding", so that a specific function in the recipient will be called on all remote method calls, irrespective of object paths in the calls. This allows one to do on-demand targeting of method calls, so that an user might remove a specific object in an address book service (using an object path similar to `/org/maemo/AddressBook/Contacts/ShortName`). Due to the limitations in characters that you can put into the object path, this is not recommended. A better way would be to supply the `ShortName` as a method call argument instead (as an UTF-8 formatted string).
  - It is common to form the object path using the same elements as in the well-known name, but replacing the dots with slashes and appending a specific object name to the end. For example: `/org/maemo/Alert/Alerter`. It is a convention, but also solves a specific problem when a process might re-use an existing D-Bus connection without explicitly knowing about it (using a library that encapsulates D-Bus functionality). Using short names here would increase the risk of name-space collisions within that process.
  - Similar to well-known names, object paths do not have inherent hierarchy, even if the path separator is used. The only place where you might see some hierarchy because of path components is the introspection interface (which is out of scope of this material).

- In order to support object oriented mapping where objects are the units providing the service, D-Bus also implements a naming unit called the interface. The interface specifies the legal (i.e., defined and implemented) method calls, their parameters (called *arguments* in D-Bus) and possible signals. It is then possible to re-use the same interface across multiple separate objects implementing the same service, or more commonly, that a single object implements multiple different services. An example of latter is the implementation of the `org.freedesktop.DBus.Introspectable` interface which defines the method necessary to support D-Bus introspection (more on this later on). If you're going to use the GLib/D-Bus wrappers to generate parts of your D-Bus code, your objects will automatically also support the introspection interface.
  - Interface names use the same naming rules as well-known names. This might seem somewhat confusing at start since well-known names serve a completely different purpose, but with time, you'll get used to it.
  - For simple services, it is common to repeat the well-known name in interface name. This is the most common scenario with existing services.

- The last part of the message address is the *member name*. When dealing with remote procedure calls, this is also sometimes called *method name* and when dealing with signals, *signal name*. The member name selects which procedure to call, or which signal to emit. It needs to be unique only within the interface that an object will implement.

  - Member names can have letters, digits and underscores in them. For example: `RetrieveQuote`.

- For a more in-depth review on these, please see the Introduction to D-Bus page.

That about covers the most important rules in D-Bus addresses that you're likely to encounter. Below is an example of all four components that we'll also use shortly to send a simple message (a method call) in the SDK:

```
#define SYSNOTE_NAME  "org.freedesktop.Notifications"
#define SYSNOTE_OPATH "/org/freedesktop/Notifications"
#define SYSNOTE_IFACE "org.freedesktop.Notifications"
#define SYSNOTE_NOTE  "SystemNoteDialog"
```

Listing 2.1: D-Bus naming components

Even if you later decide to use the LibOSSO RPC functions (which encapsulate a lot of the D-Bus machinery), you will still operate with all of the D-Bus naming components.

## 2.4 Role of D-Bus in maemo

D-Bus has been selected as de facto IPC mechanism in maemo, to carry messages between the various software components. The main reason for this is that a lot of software developed for the GNOME environment is already exposing its functionality through D-Bus. Using a generic interface which is not bound to any specific service makes it also easier to deal with different software license requirements.

The SDK unfortunately does not come with a lot of software that is exposed via D-Bus, but we'll be using one component of the application framework as demonstration (it works also in the SDK).

We're particularly interested in asking the notification framework component to display a Note dialog. The dialog is modal, which means that users cannot proceed in their graphical environment unless they first acknowledge the dialog. Normally you would try to avoid such GUI decisions, but later on we'll see why and when this feature can be useful. Please note that the `SystemNoteDialog` member is an extension to the draft `org.freedesktop.Notifications` specification, and as such, is not documented in that draft.

The notification server is listening for method calls on the `org.freedesktop.Notifications` well-known name. The object that implements the necessary interface is located at `/org/freedesktop/Notifications` object path. The method to display the note dialog is called `SystemNoteDialog` and is defined in the `org.freedesktop.Notifications` D-Bus interface.

D-Bus comes with a handy tool to experiment with method calls and signals: `dbus-send`. We'll attempt to use it in the following snippet to display the dialog:

```
[sbox-DIABLO_X86: ~] > run-standalone.sh dbus-send --print-reply  \
 --type=method_call --dest=org.freedesktop.Notifications  \
 /org/freedesktop/Notifications org.freedesktop.Notifications
Error org.freedesktop.DBus.Error.UnknownMethod: Method "Notifications" with
 signature "" on interface "org.freedesktop" doesn't exist
```

Invoking dbus-send without a method name (dbus-send thinks that `Notifications` is the method name)

Parameters for `dbus-send`:

- `--session`: (implicit since default) which bus to use for sending (the other option being `--system`)
- `--print-reply`: ask the tool to wait for a reply to the method call, and print out the results (if any)
- `--type=method_call`: instead of sending a signal (which is the default), make a method call
- `--dest=org.freedesktop.Notifications`: the well-known name for the target service
- `/org/freedesktop/Notifications`: object path within the target process that implements the interface
- `org.freedesktop.Notifications`: (incorrectly specified) interface name defining the method

When using `dbus-send`, extra care needs to be taken when specifying the interface and member names. The tool expects both of them to be combined into one parameter (without spaces in between). We modify the command line a bit and try again:

```
[sbox-DIABLO_X86: ~] > run-standalone.sh dbus-send --print-reply  \
 --type=method_call --dest=org.freedesktop.Notifications  \
 /org/freedesktop/Notifications org.freedesktop.Notifications.SystemNoteDialog
Error org.freedesktop.DBus.Error.UnknownMethod: Method "SystemNoteDialog" with
 signature "" on interface "org.freedesktop.Notifications" doesn't exist
```

Correct method name, but insufficient arguments.

Seems that the RPC call is still missing something. Most RPC methods will expect a series of parameters (or arguments, as D-Bus calls them).

`SystemNoteDialog` expects these three parameters (in this order):

- `string`: The message to display
- `uint32`: An unsigned integer giving the style of the dialog. Styles 0-4 mean different icons and style 5 is a special animated "progress indicator" dialog.
- `string`: Message to use for the "Ok" button that the user needs to press to dismiss the dialog. Using an empty string will cause the default text to be used (which is "Ok").

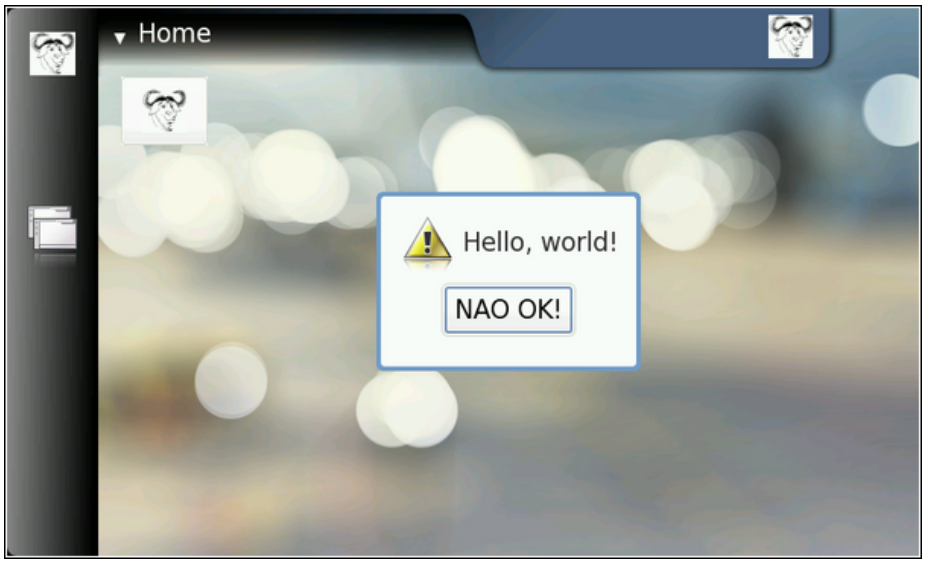
Arguments are specified by giving the argument type and its contents separated with a colon as follows:

```
[sbox-DIABLO_X86: ~] > run-standalone.sh dbus-send --print-reply  \
 --type=method_call --dest=org.freedesktop.Notifications  \
 /org/freedesktop/Notifications org.freedesktop.Notifications.SystemNoteDialog  \
 string:'Hello, world!' uint32:0 string:'NAO OK!'
method return sender=:1.1 -> dest=:1.15
   uint32 4
```

Correct method name and correct arguments

Since we asked `dbus-send` to print replies, we'll see the reply comes as a single unsigned integer, with value of 4. This is the unique number for this notification, and could be used with the `CloseNotification` method of the Notifications-interface to pre-emptively close the dialog. It might be especially useful if your software will notice that some warning condition has ended and there's no need to bother the user with the warning anymore.

Assuming that you run the above command while the application framework is already running, the end result should more or less look like this:




Result of the `SystemNoteDialog` method call

If you repeat the command multiple times, you will notice that the notification service is capable of displaying only one dialog at a time. This makes sense as the dialog is modal anyway. You will also notice that the method calls are queued somewhere and not lost (i.e., the notification service will display all of the requested dialogs). The service also acknowledges the RPC method call without delay (which is not always the obvious thing to do), giving a different return value each time (incrementing by one each time).

## 2.5 Programming directly with libdbus

The lowest level library to use for D-Bus programming is `libdbus`. Using this library directly is discouraged, mostly because it contains a lot of specific code to integrate into various main-loop designs that the higher level language bindings use.

The libdbus API reference documentation at maemo.org contains a helpful note:

```
/**
 * Uses the low-level libdbus which shouldn't be used directly.
 * As the D-Bus API reference puts it "If you use this low-level API
 * directly, you're signing up for some pain".
 */
```

Listing 2.2: Warning about using libdbus directly (libdbus-example/dbus-example.c)

We ignore the warnings and use the library to implement a simple program that will replicate the `dbus-send` example that we saw before. In order to do this with the minimum amount of code, the code will not process (nor expect) any responses to the method call. It will however demonstrate the bare minimum function calls that you'll need to use to send messages on the bus.

We'll start by introducing the necessary header files.

```
#include <dbus/dbus.h> /* Pull in all of D-Bus headers. */
#include <stdio.h>     /* printf, fprintf, stderr */
#include <stdlib.h>    /* EXIT_FAILURE, EXIT_SUCCESS */
#include <assert.h>    /* assert */

/* Symbolic defines for the D-Bus well-known name, interface, object
   path and method name that we're going to use. */

#define SYSNOTE_NAME  "org.freedesktop.Notifications"
#define SYSNOTE_OPATH "/org/freedesktop/Notifications"
#define SYSNOTE_IFACE "org.freedesktop.Notifications"
#define SYSNOTE_NOTE  "SystemNoteDialog"
```

Listing 2.3: Including the necessary headers and the symbolic constants for the method call (libdbus-example/dbus-example.c)

Unlike the rest of the code in this material, `dbus-example` does not use GLib nor other support libraries (other than `libdbus`). This explains why we'll use printf and other functions that you'd normally replace with GLib equivalents.

Connecting to the session bus will (hopefully) yield a `DBusConnection` structure:

```
/**
 * The main program that demonstrates a simple "fire & forget" RPC
 * method invocation.
 */
int main(int argc, char** argv) {

  /* Structure representing the connection to a bus. */
  DBusConnection* bus = NULL;
  /* The method call message. */
  DBusMessage* msg = NULL;

  /* D-Bus will report problems and exceptions using the DBusError
     structure. We'll allocate one in stack (so that we don't need to
     free it explicitly. */
  DBusError error;

  /* Message to display. */
  const char* dispMsg = "Hello World!";
  /* Text to use for the acknowledgement button. "" means default. */
  const char* buttonText = "";
```

```
  /* Type of icon to use in the dialog (1 = OSSO_GN_ERROR). We could
     have just used the symbolic version here as well, but that would
     have required pulling the LibOSSO-header files. And this example
     must work without LibOSSO, so this is why a number is used. */
  int iconType = 1;

  /* Clean the error state. */
  dbus_error_init(&error);

  printf("Connecting to Session D-Bus\n");
  bus = dbus_bus_get(DBUS_BUS_SESSION, &error);
  terminateOnError("Failed to open Session bus\n", &error);
  assert(bus != NULL);
```

Listing 2.4: Connecting to the Session bus (libdbus-example/dbus-example.c)

Note that `libdbus` will attempt to share existing connection structures when the same process is connecting to the same bus. This is done to avoid the somewhat costly connection setup time. Sharing connections is beneficial when your program is using libraries which would also open their own connections to the same buses.

In order to communicate errors, `libdbus` uses `DBusError` structures, whose contents are pretty simple. We'll use the `dbus_error_init` to guarantee that the error structure contains a non-error state before connecting to the bus. If there's an error, it will be handled in `terminateOnError`:

```
/**
 * Utility to terminate if given DBusError is set.
 * Will print out the message and error before terminating.
 *
 * If error is not set, will do nothing.
 *
 * NOTE: In real applications you should spend a moment or two
 *       thinking about the exit-paths from your application and
 *       whether you need to close/unreference all resources that you
 *       have allocated. In this program, we rely on the kernel to do
 *       all necessary cleanup (closing sockets, releasing memory),
 *       but in real life you need to be more careful.
 *
 *       One possible solution model to this is implemented in
 *       "flashlight", a simple program that is presented later.
 */
static void terminateOnError(const char* msg,
                             const DBusError* error) {

  assert(msg != NULL);
  assert(error != NULL);

  if (dbus_error_is_set(error)) {
    fprintf(stderr, msg);
    fprintf(stderr, "DBusError.name: %s\n", error->name);
    fprintf(stderr, "DBusError.message: %s\n", error->message);
    /* If the program wouldn't exit because of the error, freeing the
       DBusError needs to be done (with dbus_error_free(error)).
       NOTE:
         dbus_error_free(error) would only free the error if it was
         set, so it is safe to use even when you're unsure. */
    exit(EXIT_FAILURE);
  }
}
```

Listing 2.5: Processing DBusErrors (libdbus-example/dbus-example.c)

`libdbus` also contains some utility functions, so that you don't have to code everything manually. One such utility is `dbus_bus_name_has_owner` which checks whether there is at least some process who owns the given well known name at that moment:

```
/* Normally one would just do the RPC call immediately without
   checking for name existence first. However, sometimes it's useful
   to check whether a specific name even exists on a platform on
   which you're planning to use D-Bus.

   In our case it acts as a reminder to run this program using the
   run-standalone.sh script when running in the SDK.

   The existence check is not necessary if the recipient is
   startable/activateable by D-Bus. In that case, if the recipient
   is not already running, the D-Bus daemon will start the
   recipient (a process that has been registered for that
   well-known name) and then passes the message to it. This
   automatic starting mechanism will avoid the race condition
   discussed below and also makes sure that only one instance of
   the service is running at any given time. */
printf("Checking whether the target name exists ("
       SYSNOTE_NAME ")\n");
if (!dbus_bus_name_has_owner(bus, SYSNOTE_NAME, &error)) {
  fprintf(stderr, "Name has no owner on the bus!\n");
  return EXIT_FAILURE;
}
terminateOnError("Failed to check for name ownership\n", &error);
/* Someone on the Session bus owns the name. So we can proceed in
   relative safety. There is a chance of a race. If the name owner
   decides to drop out from the bus just after we check that it is
   owned, our RPC call (below) will fail anyway. */
```

Listing 2.6: Checking for well-known name availability (libdbus-example/dbus-example.c)

Creating a method call using `libdbus` is slightly more tedious than using the higher-level interfaces, but not very difficult. The process is separated into two steps: creating a message structure, and appending the arguments to the message:

```
/* Construct a DBusMessage that represents a method call.
   Parameters will be added later. The internal type of the message
   will be DBUS_MESSAGE_TYPE_METHOD_CALL. */
printf("Creating a message object\n");
msg = dbus_message_new_method_call(SYSNOTE_NAME, /* destination */
                                   SYSNOTE_OPATH,  /* obj. path */
                                   SYSNOTE_IFACE,  /* interface */
                                   SYSNOTE_NOTE); /* method str */
if (msg == NULL) {
  fprintf(stderr, "Ran out of memory when creating a message\n");
  exit(EXIT_FAILURE);
}

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

/* Add the arguments to the message. For the Note dialog, we need
```

```
    three arguments:
      arg0: (STRING) "message to display, in UTF-8"
      arg1: (UINT32) type of dialog to display. We will use 1.
                     (libosso.h/OSSO_GN_ERROR).
      arg2: (STRING) "text to use for the ack button". "" means
                     default text (OK in our case).

    When listing the arguments, the type needs to be specified first
    (by using the libdbus constants) and then a pointer to the
    argument content needs to be given.

    NOTE: It is always a pointer to the argument value, not the value
          itself!

    We terminate the list with DBUS_TYPE_INVALID. */
  printf("Appending arguments to the message\n");
  if (!dbus_message_append_args(msg,
                                DBUS_TYPE_STRING, &dispMsg,
                                DBUS_TYPE_UINT32, &iconType,
                                DBUS_TYPE_STRING, &buttonText,
                                DBUS_TYPE_INVALID)) {
    fprintf(stderr, "Ran out of memory while constructing args\n");
    exit(EXIT_FAILURE);
  }
```

Listing 2.7: Constructing a D-Bus message (method call in our case) (libdbus-example/dbus-example.c)

When arguments are appended to the message, their content is copied and possible converted into a format that will be sent over the connection to the daemon. This process is called *marshaling* and is a common feature to most RPC systems. Our method call will require two parameters (as before), the first being the text to display and the second one being the style of the icon to use. Parameters passed to `libdbus` are always passed by address. This is different from the higher level libraries, and we'll return to this later.

The arguments are encoded so that their type code is followed by the pointer where the marshaling functions can find the content. The argument list is terminated with `DBUS_TYPE_INVALID` so that the function knows where the argument list ends (since the function prototype ends with an ellipsis, ...).

```
  /* Set the "no-reply-wanted" flag into the message. This also means
     that we cannot reliably know whether the message was delivered or
     not, but since we don't have reply message handling here, it
     doesn't matter. The "no-reply" is a potential flag for the remote
     end so that they know that they don't need to respond to us.

     If the no-reply flag is set, the D-Bus daemon makes sure that the
     possible reply is discarded and not sent to us. */
  dbus_message_set_no_reply(msg, TRUE);
```

Listing 2.8: Specifying that we don't expect a reply from to the message (libdbus-example/dbus-example.c)

By setting the no-reply-flag, we're effectively telling the bus daemon that even if there is a reply coming back for this RPC method, we don't want it. The daemon will not send one to us in this case.

Once the message is fully constructed, it can be added to the sending queue of our program. Messages are not sent immediately by `libdbus`. Normally this

allows the message queue to accumulate more than one message and all of the messages to be sent at once to the daemon. This in turn cuts down the number of context switches necessary. In our case, this will be the only message that the program ever sends, so we ask the send queue to be flushed immediately, and this will instruct the library to send all messages to the daemon immediately:

```
  printf("Adding message to client's send-queue\n");
  /* We could also get a serial number (dbus_uint32_t) for the message
     so that we could correlate responses to sent messages later. In
     our case there won't be a response anyway, so we don't care about
     the serial, so we pass a NULL as the last parameter. */
  if (!dbus_connection_send(bus, msg, NULL)) {
    fprintf(stderr, "Ran out of memory while queueing message\n");
    exit(EXIT_FAILURE);
  }

  printf("Waiting for send-queue to be sent out\n");
  dbus_connection_flush(bus);

  printf("Queue is now empty\n");
```

Listing 2.9: Sending the message (libdbus-example/dbus-example.c)

After we're done sending the message, we start tearing down the reserved resources. We'll first free up the message and then free up the connection structure.

```
  printf("Cleaning up\n");

  /* Free up the allocated message. Most D-Bus objects have internal
     reference count and sharing possibility, so _unref() functions
     are quite common. */
  dbus_message_unref(msg);
  msg = NULL;

  /* Free-up the connection. libdbus attempts to share existing
     connections for the same client, so instead of closing down a
     connection object, it is unreferenced. The D-Bus library will
     keep an internal reference to each shared connection, to
     prevent accidental closing of shared connections before the
     library is finalized. */
  dbus_connection_unref(bus);
  bus = NULL;

  printf("Quitting (success)\n");

  return EXIT_SUCCESS;
}
```

Listing 2.10: D-Bus cleanup (libdbus-example/dbus-example.c)

After building the program, we'll attempt to run it:

```
[sbox-DIABLO_X86: ~/libdbus-example] > ./dbus-example
Connecting to Session D-Bus
process 6120: D-Bus library appears to be incorrectly set up;
 failed to read machine uuid:
  Failed to open "/var/lib/dbus/machine-id": No such file or directory
See the manual page for dbus-uuidgen to correct this issue.
  D-Bus not built with -rdynamic so unable to print a backtrace
Aborted (core dumped)
```

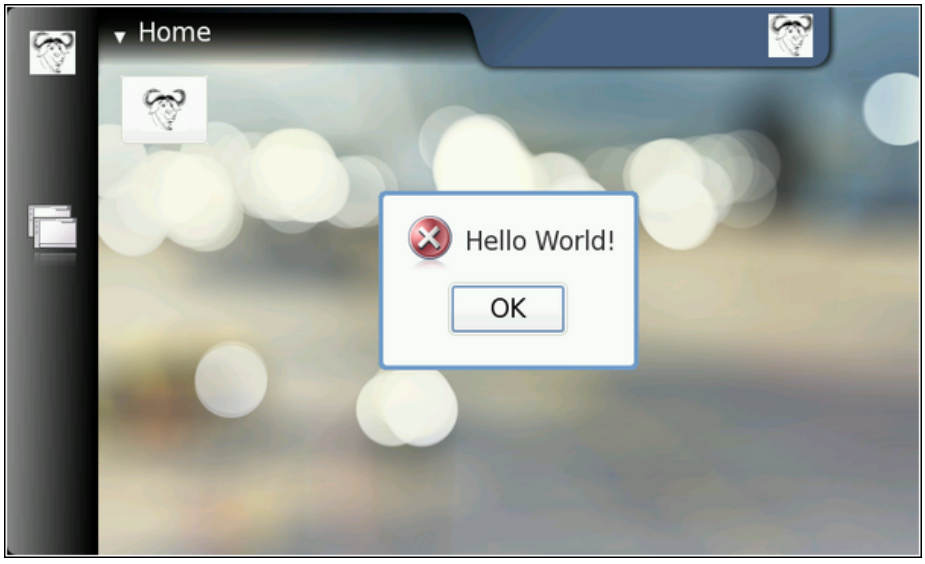
Running the example, but without the proper environmental variables

The D-Bus library needs environmental variables set correctly in order to locate the session daemon. We forgot to prepend the command with **run-standalone.sh** and this caused the library internally abort the execution. Normally, `dbus_bus_get` would have returned a `NULL` pointer and set the error structure, but the version on the 4.0 SDK will assert internally in this condition and programs cannot avoid the abort. Correcting the small mishap, we try again:

```
[sbox-DIABLO_X86: ~/libdbus-example] > run-standalone.sh ./dbus-example
Connecting to Session D-Bus
Checking whether the target name exists (org.freedesktop.Notifications)
Creating a message object
Appending arguments to the message
Adding message to client's send-queue
Waiting for send-queue to be sent out
Queue is now empty
Cleaning up
Quitting (success)
/dev/dsp: No such file or directory
```

Running the example with proper environmental variables

The error message (about **/dev/dsp**) printed to the same terminal where AF was started is normal (in SDK). Displaying the Note dialog normally also causes an "Alert" sound to be played. The sound system has not been setup in the SDK, so the notification component complains about failing to open the sound device.




Our friendly error message, using low-level D-Bus

In order to get `libdbus` integrated into makefiles, one will have to use `pkg-config` and one possible solution is presented below:

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

PKG_CFLAGS  := $(shell pkg-config --cflags $(pkg_packages))
PKG_LDFLAGS := $(shell pkg-config --libs $(pkg_packages))
# Additional flags for the compiler:
#    -g : Add debugging symbols
# -Wall : Enable most gcc warnings
ADD_CFLAGS := -g -Wall

# Combine user supplied, additional, and pkg-config flags
CFLAGS  := $(PKG_CFLAGS) $(ADD_CFLAGS) $(CFLAGS)
LDFLAGS := $(PKG_LDFLAGS) $(LDFLAGS)
```

Listing 2.11: Integrating libdbus into Makefiles (libdbus-example/Makefile)

Above is one possibility to integrate user-supplied variables into makefiles so that they will still be passed along the toolchain. This allows the user to execute `make` with custom flags, overriding those that are introduced via other means. For example: `"CFLAGS='-g0' make"` would result in `-g0` being interpreted after the `-g` that is in the **Makefile**, and this would lead to debugging symbols being disabled. Environmental variables can be taken into account in exactly the same way.

For more complicated programs, it's likely that you'll require multiple different `CFLAGS` settings for different object files that you're building (or multiple different programs). In that case you'd do the combining in each target rule separately. In this material all the example programs are self-contained and rather simple, so we'll be using the above mechanism in all the example makefiles.

You might also want to consider using GNU autotools for some larger projects, but this material will use GNU make as the software building tool.

# Chapter 3

# LibOSSO

## 3.1 Introduction to LibOSSO

LibOSSO is a library that all applications designed for maemo are expected to use. Mainly because it automatically allows the application to survive the task killing process. This task killing is done by the Desktop environment when an application launched from the Task navigator doesn't register the proper D-Bus name on the bus within a certain time limit after the launch. LibOSSO also conveniently isolates the application from possible implementation changes on D-Bus level. D-Bus used to be not API stable before as well, so LibOSSO provided "version isolation" with respect D-Bus. Since D-Bus has reached maturity (1.0), no API changes are expected for the low level library, but the GLib/D-Bus wrapper might still change at some point.

Besides the protection and isolation services, LibOSSO also provides useful utility functions to handle autosaving and state saving features of the platform, process hardware state and device mode changes and other important events happening in Internet Tablets. It also provides convenient utility wrapper functions to send RPC method calls over the D-Bus. The feature set is aimed at covering the most common GUI application needs, and as such, will not be enough in all cases. In these cases it will be necessary to use the GLib/D-Bus wrapper functions (or libdbus directly, which is not recommended).

In this material we will be concentrating on the RPC aspects of LibOSSO (and use an utility function or two as well).

## 3.2 Using LibOSSO for D-Bus method calls

We'll start by re-implementing the functionality from the libdbus example that we used before, but use LibOSSO functions instead of direct libdbus ones. The new version will use exactly the same D-Bus name-space components to pop up a Note dialog. LibOSSO also contains a function to do all this for you (`osso_system_note_dialog`) which we'll use directly later on. It is however instructive to see what LibOSSO provides in terms of RPC support and using a familiar RPC method is the easiest way to achieve this.

We'll start with the header section of the example program:

```c
#include <libosso.h>

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

#define SYSNOTE_NAME  "org.freedesktop.Notifications"
#define SYSNOTE_OPATH "/org/freedesktop/Notifications"
#define SYSNOTE_IFACE "org.freedesktop.Notifications"
#define SYSNOTE_NOTE  "SystemNoteDialog"
```

Listing 3.1: Header files for the simple example (libosso-example-sync/libosso-rpc-sync.c)

LibOSSO by itself only requires the libosso.h header file to be included. We'll also use the exact same D-Bus well-known name, object path, interface name and method name as before.

When reading other source code that implements or uses D-Bus services, you might sometimes wonder why the D-Bus interface name is using the same symbolic constant as the well-known name (in the above example `SYSNOTE_IFACE` would be omitted, and `SYSNOTE_NAME` would be used whenever an interface name would be required). If the service in question is not easily reusable or re-implementable, it might make sense to use an interface name that is as unique as the well-known name. This goes against the idea of defining interfaces, but is still quite common (and is the easy way out without bothering with difficult design decisions).

We continue by looking at how LibOSSO contexts are created and how they're eventually released:

```c
int main(int argc, char** argv) {

  /* The LibOSSO context that we need to do RPC. */
  osso_context_t* ossoContext = NULL;

  g_print("Initializing LibOSSO\n");
  /* The program name for registration is communicated from the
     Makefile via a -D preprocessor directive. Since it doesn't
     contain any dots in it, a prefix of "com.nokia." will be added
     to it internally within osso_initialize(). */
  ossoContext = osso_initialize(ProgName, "1.0", FALSE, NULL);
  if (ossoContext == NULL) {
    g_error("Failed to initialize LibOSSO\n");
  }

  g_print("Invoking the method call\n");
  runRPC(ossoContext);

  g_print("Shutting down LibOSSO\n");
  /* Deinitialize LibOSSO. The function doesn't return status code so
     we cannot know whether it succeeded or failed. We assume that it
     always succeeds. */
  osso_deinitialize(ossoContext);
  ossoContext = NULL;

  g_print("Quitting\n");
  return EXIT_SUCCESS;
}
```

Listing 3.2: Initialising and closing LibOSSO (libosso-example-sync/libosso-rpc-sync.c)

An LibOSSO context is a small structure containing the necessary information for the LibOSSO functions to communicate over D-Bus (both session and system buses). When a context is created, you'll need to pass your "application name" to `osso_initialize`. This name is used to register a name on the D-Bus and this will keep the task killer from killing your process later on (assuming the application was started via the Task navigator). If your application name does not contain any dot characters in it, `com.nokia.` will be prepended to it automatically in LibOSSO. The application name is normally not visible to users, so this shouldn't be a big problem. You might run into application name collisions if some other application will use the same name (and also without the dots), so it might be a good idea to provide a proper name based on a DNS domain you own or control. If you plan to implement a service to clients over the D-Bus (with `osso_rpc_set_cb`-functions) you'll need to be extra careful about the application name used here.

The version number is currently still unused, but `1.0` is recommended for now. The second to last parameter is obsolete and has no effect while the last parameter tells LibOSSO which mainloop structure to integrate to. Using `NULL` here means that LibOSSO event processing will integrate to the default GMainLoop object created (which is what you most often want).

Releasing the LibOSSO context will automatically close the connections to the D-Bus buses and release all allocated memory related to the connections and LibOSSO state. If you want to use LibOSSO functions after this, you'll have to initialize a context again.

The following snippet shows the RPC call using LibOSSO, and also contains the code suitable for using to deal with possible errors in the launch as well as the result of the RPC. The `ossoErrorStr` function is covered shortly as is the utility function to print out the result structure.

```
/**
 * Do the RPC call.
 *
 * Note that this function will block until the method call either
 * succeeds, or fails. If the method call would take a long time to
 * run, this would block the GUI of the program (which we don't have).
 *
 * Needs the LibOSSO state to do the launch.
 */
static void runRPC(osso_context_t* ctx) {

  /* Message to display. */
  const char* dispMsg = "Hello SystemNote!\nVia LibOSSO/sync.";
  /* Icon type to use. */
  gint iconType = OSSO_GN_ERROR;
  /* Button label text to use, "" means leaving the defaults. */
  const char* labelText = "";

  /* Will hold the result from the RPC invocation function. */
  osso_return_t result;
  /* Will hold the result of the method call (or error). */
  osso_rpc_t methodResult = {};

  g_print("runRPC called\n");

  g_assert(ctx != NULL);
```

```
  /* Compared to the libdbus functions, LibOSSO provides conveniently
     a function that will do the dispatch and also allows us to pass
     the arguments all with one call.

     The arguments for the "SystemNoteDialog" are the same as in
     dbus-example.c (since it is the same service). You might also
     notice that even if LibOSSO provides some convenience, it does
     not completely isolate us from libdbus. We still supply the
     argument types using D-Bus constants.

     NOTE Do not pass the argument values by pointers as with libdbus,
          instead pass them by value (as below). */
  result = osso_rpc_run(ctx,
                        SYSNOTE_NAME,      /* well-known name */
                        SYSNOTE_OPATH,         /* object path */
                        SYSNOTE_IFACE,           /* interface */
                        SYSNOTE_NOTE,          /* method name */
                        &methodResult, /* method return value */
                        /* The arguments for the RPC. The types
                           are unchanged, but instead of passing
                           them via pointers, they're passed by
                           "value" instead. */
                        DBUS_TYPE_STRING, dispMsg,
                        DBUS_TYPE_UINT32, iconType,
                        DBUS_TYPE_STRING, labelText,
                        DBUS_TYPE_INVALID);
  /* Check whether launching the RPC succeeded. */
  if (result != OSSO_OK) {
    g_error("Error launching the RPC (%s)\n",
            ossoErrorStr(result));
    /* We also terminate right away since there's nothing to do. */
  }
  g_print("RPC launched successfully\n");

  /* Now decode the return data from the method call.
     NOTE: If there is an error during RPC delivery, the return value
           will be a string. It is not possible to differentiate that
           condition from an RPC call that returns a string.

     If a method returns "void", the type-field in the methodResult
     will be set to DBUS_TYPE_INVALID. This is not an error. */
  g_print("Method returns: ");
  printOssoValue(&methodResult);
  g_print("\n");

  g_print("runRPC ending\n");
}
```

Listing 3.3: Issuing the method call and waiting for response (libosso-example-sync/libosso-rpc-sync.c)

It is important to note that `osso_rpc_run` is a synchronous (blocking) call which will wait for either the response from the method call, a timeout or an error. In our case the method will be handled quickly so it's not a big problem, but in many cases the methods take some time to execute (and might require loading external resources) so you should keep this in mind. Asynchronous LibOSSO RPC functions will be covered shortly.

If your method call will return more than one return value (this is possible in D-Bus), LibOSSO currently does not provide a mechanism to return all of them (it will return the first value only).

Decoding the result code from the LibOSSO RPC functions is pretty straight forward and is done in a separate utility:

```
/**
 * Utility to return a pointer to a statically allocated string giving
 * the textural representation of LibOSSO errors. Has no internal
 * state (safe to use from threads).
 *
 * LibOSSO does not come with a function for this, so we define one
 * ourselves.
 */
static const gchar* ossoErrorStr(osso_return_t errCode) {

  switch (errCode) {
    case OSSO_OK:
      return "No error (OSSO_OK)";
    case OSSO_ERROR:
      return "Some kind of error occurred (OSSO_ERROR)";
    case OSSO_INVALID:
      return "At least one parameter is invalid (OSSO_INVALID)";
    case OSSO_RPC_ERROR:
      return "Osso RPC method returned an error (OSSO_RPC_ERROR)";
    case OSSO_ERROR_NAME:
      return "(undocumented error) (OSSO_ERROR_NAME)";
    case OSSO_ERROR_NO_STATE:
      return "No state file found to read (OSSO_ERROR_NO_STATE)";
    case OSSO_ERROR_STATE_SIZE:
      return "Size of state file unexpected (OSSO_ERROR_STATE_SIZE)";
    default:
      return "Unknown/Undefined";
  }
}
```

Listing 3.4: Decoding LibOSSO errors (libosso-example-sync/libosso-rpc-sync.c)

Decoding the RPC return value is however slightly more complex, as the return value is a structure which contains a typed union (type is encoded in the `type` field of the structure):

```
/**
 * Utility to print out the type and content of given osso_rpc_t.
 * It also demonstrates the types available when using LibOSSO for
 * the RPC. Most simple types are available, but arrays are not
 * (unfortunately).
 */
static void printOssoValue(const osso_rpc_t* val) {

  g_assert(val != NULL);

  switch (val->type) {
    case DBUS_TYPE_BOOLEAN:
      g_print("boolean:%s", (val->value.b == TRUE)?"TRUE":"FALSE");
      break;
    case DBUS_TYPE_DOUBLE:
      g_print("double:%.3f", val->value.d);
      break;
    case DBUS_TYPE_INT32:
      g_print("int32:%d", val->value.i);
      break;
    case DBUS_TYPE_UINT32:
```

```
      g_print("uint32:%u", val->value.u);
      break;
    case DBUS_TYPE_STRING:
      g_print("string:'%s'", val->value.s);
      break;
    case DBUS_TYPE_INVALID:
      g_print("invalid/void");
      break;
    default:
      g_print("unknown(type=%d)", val->type);
      break;
  }
}
```

Listing 3.5: Decoding RPC results (libosso-example-sync/libosso-rpc-sync.c)

Note that LibOSSO RPC functions do not support array parameters either, so you're restricted to use method calls that only have simple parameters.

We build the example and then run it. The end result is the now familiar Note dialog.

```
[sbox-DIABLO_X86: ~/libosso-example-sync] > run-standalone.sh ./libosso-rpc-sync
Initializing LibOSSO
Invoking the method call
runRPC called
/dev/dsp: No such file or directory
RPC launched successfully
Method returns: uint32:8
runRPC ending
Shutting down LibOSSO
Quitting
```

Running the example

The only difference is the location of the audio device error message. It will now appear before `runRPC` returns, since `runRPC` waits for RPC completion. You never should rely on this kind of ordering, because the RPC execution could also be delayed (and the message might appear at a later location when you try this program).

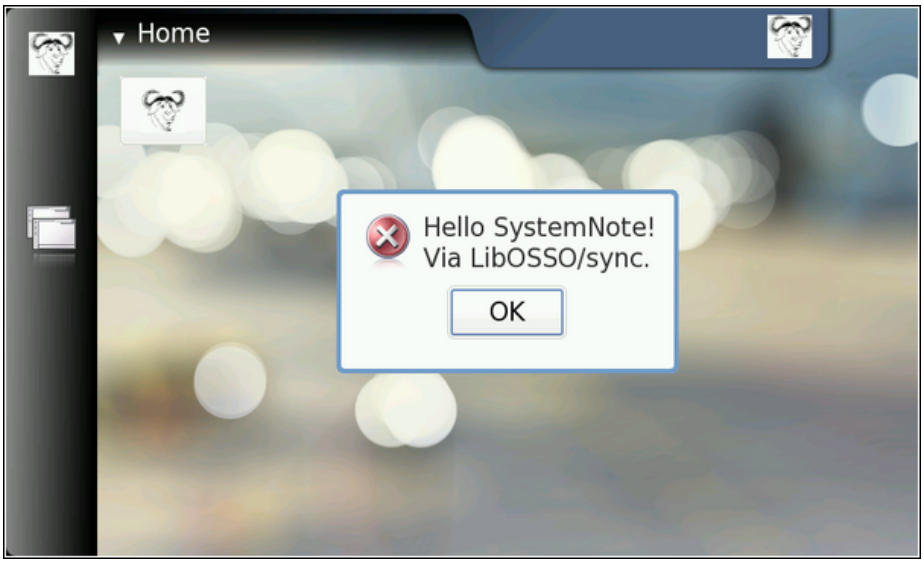



Figure 3.1: The end result

One point of interest in the Makefile is the mechanism by which the `ProgName` define is set. It is often useful to separate the program name related information outside the source code, so that the code fragment may then be re-used more easily. In this case, we control the application name that is used when LibOSSO is initialised from the Makefile.

```
# define a list of pkg-config packages we want to use
pkg_packages := glib-2.0 libosso

# ... Listing cut for brevity ...

libosso-rpc-sync: libosso-rpc-sync.c
  $(CC) $(CFLAGS) -DProgName=\"LibOSSOExample\" \
    $< -o $@ $(LDFLAGS)
```

Listing 3.6: Integrating LibOSSO into Makefiles (libosso-example-sync/Makefile)

## 3.3 Asynchronous method calls with LibOSSO

Sometimes the method call will take long time to run (or you're not sure whether it might take long time to run). In these cases you should use the asynchronous RPC utility functions in LibOSSO instead of the synchronous ones. The biggest difference is that the method call will be split into two parts: launching of the RPC and handling its result in a callback function. The same limitations with respect to method parameter types and the number of return values still apply.

In order for the callback to use LibOSSO functions and control the mainloop object, we'll need to create a small application state. The state will be passed to the callback when necessary.

```
/**
 * Small application state so that we can pass both LibOSSO context
 * and the mainloop around to the callbacks.
 */
typedef struct {
  /* A mainloop object that will "drive" our example. */
  GMainLoop* mainloop;
  /* The LibOSSO context which we use to do RPC. */
  osso_context_t* ossoContext;
} ApplicationState;
```

Listing 3.7: Application state for the example (libosso-example-async/libosso-rpc-async.c)

The `osso_rpc_async_run` function is used to launch the method call and it will normally return immediately. If it returns an error, it will be probably a client-side error (since the RPC method hasn't returned by then). The callback function to handle the RPC response will be registered with the function, as will the name-space related parameters and the method call arguments:

```
/**
 * We launch the RPC call from within a timer callback in order to
 * make sure that a mainloop object will be running when the RPC will
 * return (to avoid a nasty race condition).
 *
 * So, in essence this is a one-shot timer callback.
 *
 * In order to launch the RPC, it will need to get a valid LibOSSO
 * context (which is carried via the userData/application state
 * parameter).
 */
static gboolean launchRPC(gpointer userData) {

  ApplicationState* state = (ApplicationState*)userData;
  /* Message to display. */
  const char* dispMsg = "Hello SystemNote!\nVia LibOSSO/async.";
  /* Icon type to use. */
  gint iconType = OSSO_GN_ERROR;
  /* Button label text to use. */
  const char* labelText = "Execute!";

  /* Will hold the result from the RPC launch call. */
  osso_return_t result;

  g_print("launchRPC called\n");

  g_assert(state != NULL);

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

  /* The only difference compared to the synchronous version is the
     addition of the callback function parameter, and the user-data
     parameter for data that will be passed to the callback. */
  result = osso_rpc_async_run(state->ossoContext,
                              SYSNOTE_NAME,     /* well-known name */
                              SYSNOTE_OPATH,      /* object path */
                              SYSNOTE_IFACE,        /* interface */
                              SYSNOTE_NOTE,       /* method name */
                              rpcCompletedCallback,    /* async cb */
                              state,          /* user-data for cb */
```

```
                              /* The arguments for the RPC. */
                              DBUS_TYPE_STRING, dispMsg,
                              DBUS_TYPE_UINT32, iconType,
                              DBUS_TYPE_STRING, labelText,
                              DBUS_TYPE_INVALID);
  /* Check whether launching the RPC succeeded (we don't know the
     result from the RPC itself). */
  if (result != OSSO_OK) {
    g_error("Error launching the RPC (%s)\n",
            ossoErrorStr(result));
    /* We also terminate right away since there's nothing to do. */
  }
  g_print("RPC launched successfully\n");

  g_print("launchRPC ending\n");

  /* We only want to be called once, so ask the caller to remove this
     callback from the timer launch list by returning FALSE. */
  return FALSE;
}
```

Listing 3.8: Starting the RPC call (libosso-example-async/libosso-rpc-async.c)

Handling the return from the RPC method is handled by a simple callback function that will need to always use the same parameter prototype. It will receive the return value as well as the interface and method names. The latter two are useful as you can then use the same callback function to handle returns from multiple different (and simultaneous) RPC method calls.

The return value structure is allocated by LibOSSO and will be freed once your callback will return, so you don't need to handle that manually.

```
/**
 * Will be called from LibOSSO when the RPC return data is available.
 * Will print out the result, and return. Note that it must not free
 * the value, since it does not own it.
 *
 * The prototype (for reference) must be osso_rpc_async_f().
 *
 * The parameters for the callback are the D-Bus interface and method
 * names (note that object path and well-known name are NOT
 * communicated). The idea is that you can then reuse the same
 * callback to process completions from multiple simple RPC calls.
 */
static void rpcCompletedCallback(const gchar* interface,
                                 const gchar* method,
                                 osso_rpc_t* retVal,
                                 gpointer userData) {

  ApplicationState* state = (ApplicationState*)userData;

  g_print("rpcCompletedCallback called\n");

  g_assert(interface != NULL);
  g_assert(method != NULL);
  g_assert(retVal != NULL);
  g_assert(state != NULL);

  g_print(" interface: %s\n", interface);
  g_print(" method: %s\n", method);
  /* NOTE If there is an error in the RPC delivery, the return value
          will be a string. This is unfortunate if your RPC call is
```

```
          supposed to return a string as well, since it is not
          possible to differentiate between the two cases.

          If a method returns "void", the type-field in the retVal
          will be set to DBUS_TYPE_INVALID (it's not an error). */
  g_print(" result: ");
  printOssoValue(retVal);
  g_print("\n");

  /* Tell the main loop to terminate. */
  g_main_loop_quit(state->mainloop);

  g_print("rpcCompletedCallback done\n");
}
```

Listing 3.9: Handling the end of the RPC call (libosso-example-async/libosso-rpc-async.c)

In our case, receiving the response to the method call will cause the main program to be terminated.

The application setup logic is covered next:

```
int main(int argc, char** argv) {

  /* Keep the application state in main's stack. */
  ApplicationState state = {};
  /* Keeps the results from LibOSSO functions for decoding. */
  osso_return_t result;
  /* Default timeout for RPC calls in LibOSSO. */
  gint rpcTimeout;

  g_print("Initializing LibOSSO\n");
  state.ossoContext = osso_initialize(ProgName, "1.0", FALSE, NULL);
  if (state.ossoContext == NULL) {
    g_error("Failed to initialize LibOSSO\n");
  }

  /* Print out the default timeout value (which we don't change, but
     could, with osso_rpc_set_timeout()). */
  result = osso_rpc_get_timeout(state.ossoContext, &rpcTimeout);
  if (result != OSSO_OK) {
    g_error("Error getting default RPC timeout (%s)\n",
            ossoErrorStr(result));
  }
  /* Interestingly the timeout seems to be -1, but is something else
     (by default). -1 probably then means that "no timeout has been
     set". */
  g_print("Default RPC timeout is %d (units)\n", rpcTimeout);

  g_print("Creating a mainloop object\n");
  /* Create a GMainLoop with default context and initial condition of
     not running (FALSE). */
  state.mainloop = g_main_loop_new(NULL, FALSE);
  if (state.mainloop == NULL) {
    g_error("Failed to create a GMainLoop\n");
  }

  g_print("Adding timeout to launch the RPC in one second\n");
  /* This could be replaced by g_idle_add(cb, &state), in order to
     guarantee that the RPC would be launched only after the mainloop
     has started. We opt for a timeout here (for no particular
     reason). */
```

```
  g_timeout_add(1000, (GSourceFunc)launchRPC, &state);

  g_print("Starting mainloop processing\n");
  g_main_loop_run(state.mainloop);

  g_print("Out of mainloop, shutting down LibOSSO\n");
  /* Deinitialize LibOSSO. */
  osso_deinitialize(state.ossoContext);
  state.ossoContext = NULL;

  /* Free GMainLoop as well. */
  g_main_loop_unref(state.mainloop);
  state.mainloop = NULL;

  g_print("Quitting\n");
  return EXIT_SUCCESS;
}
```

Listing 3.10: Application setup timeout registration main loop and finalisation (libosso-example-async/libosso-rpc-async.c)

The code includes an example how to query the method call timeout value as well, however timeout values are left unchanged in our program.

The RPC method call is launched in a slightly unorthodox way, via a timeout call that will launch one second after mainloop processing starts. One could just as easily use `g_idle_add`, as long as the launching itself will be done after mainloop processing starts. Since the method return value callback will terminate the mainloop, the mainloop needs to be active at that point. The only way to guarantee this is to launch the RPC after the mainloop is active.

Testing the program yields little surprises (other than the default timeout value being -1):

```
[sbox-DIABLO_X86: ~/libosso-example-async] > run-standalone.sh ./libosso-rpc-async
Initializing LibOSSO
Default RPC timeout is -1 (units)
Creating a mainloop object
Adding timeout to launch the RPC in one second
Starting mainloop processing
launchRPC called
RPC launched successfully
launchRPC ending
rpcCompletedCallback called
 interface: org.freedesktop.Notifications
 method: SystemNoteDialog
 result: uint32:10
rpcCompletedCallback done
Out of mainloop, shutting down LibOSSO
Quitting
/dev/dsp: No such file or directory
```

You might notice another shift in the audio device error string. It is displayed now after all other messages (similar to the libdbus example). It seems that the audio playback is started "long after" the dialog itself is displayed, or maybe the method returns before SystemNote starts the dialog display. Again, one should not rely on exact timing when dealing with D-Bus remote method calls.

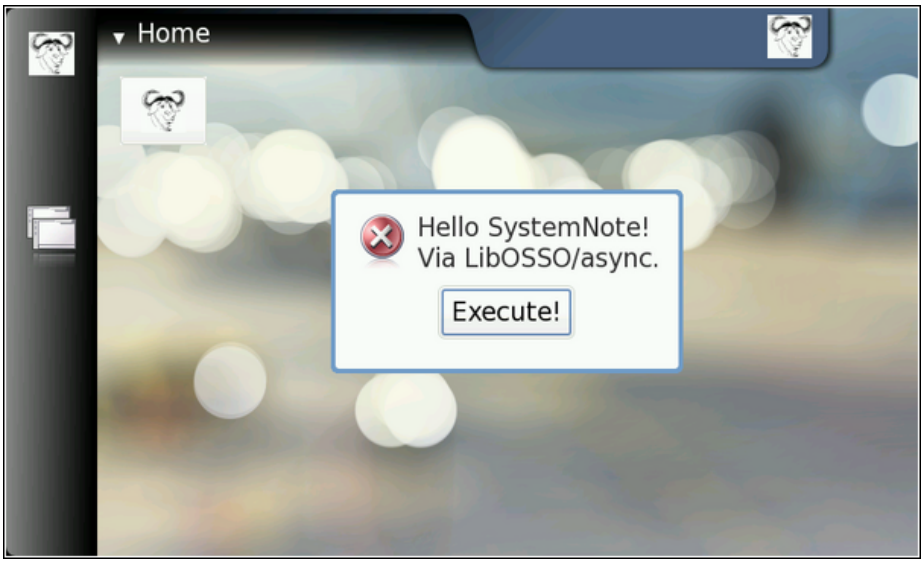



Figure 3.2: End result of the async example (no surprises)

The label text was modified slightly, to test out that non-default labels will work.

The Makefile for this example doesn't contain anything new nor special.

## 3.4 Device state and mode notifications

Since Internet Tablets are mobile devices, you should expect people to use them (and your software) while on the move, and also on airplanes and other places which might restrict network connectivity. If your program uses network connectivity, or wants to adapt to the conditions in the device better, you'll have to handle changes between the different devices states. The change between the states are normally initiated by the user of the device (when boarding an aircraft for example).

In order to demonstrate handling of the most important device state, we'll next implement a small utility program that will combine various utility functions from LibOSSO as well as handle the changes in the device state. The state that we're particularly interested in is the "flight" mode. This mode is initiated by the user by switching the device into "Offline"-mode. The internal name for this state however is "flight". Curiously enough there is also a mode called "offline" internally.

Our application is a simple utility program that keeps the backlight of the device turned on by periodically asking the system to delay the automatic display dimming functionality. Normally the backlight is turned off after short periods of inactivity, although this setting can be changed by the user. It is the goal of the application then to request a postponement of this mechanism (by 60 seconds at a time). We choose 45 seconds as an internal timer frequency so that we can always extend the time by 60 seconds (and be sure that we don't miss our opportunity by using a lower frequency than the maximum).

We also track the device mode, and once the device will enter the flight-mode, the program will terminate. Should the program be started when the device is already in flight-mode, the program will refuse to run.

Since the program has no GUI of its own, we'll also use Note dialogs and the infoprint facility to display status information to the user. The Note is used to remind the user that leaving the program running will exhaust the battery. The infoprints are used when the application will terminate or if it will refuse to run.

Most of the work required will be contained in the application setup logic, which allows us to reduce the code in `main` significantly:

```
/**
 * Main program:
 *
 * 1) Setup application state
 * 2) Start mainloop
 * 3) Release application state & terminate
 */
int main(int argc, char** argv) {

  /* We'll keep one application state in our program and allocate
     space for it from the stack. */
  ApplicationState state = {};

  g_print(PROGNAME ":main Starting\n");
  /* Attempt to setup the application state and if something goes
     wrong, do cleanup here and exit. */
  if (setupAppState(&state) == FALSE) {
    g_print(PROGNAME ":main Setup failed, doing cleanup\n");
    releaseAppState(&state);
    g_print(PROGNAME ":main Terminating with failure\n");
    return EXIT_FAILURE;
  }

  g_print(PROGNAME ":main Starting mainloop processing\n");
  g_main_loop_run(state.mainloop);
  g_print(PROGNAME ":main Out of main loop (shutting down)\n");
  /* We come here when the application state has the running flag set
     to FALSE and the device state changed callback has decided to
     terminate the program. Display a message to the user about
     termination next. */
  displayExitMessage(&state, ProgName " exiting");

  /* Release the state and exit with success. */
  releaseAppState(&state);

  g_print(PROGNAME ":main Quitting\n");
  return EXIT_SUCCESS;
}
```

Listing 3.11: Application main logic (libosso-flashlight/flashlight.c)

In order for the device state callbacks to force a quit of the application, we'll need to pass it the LibOSSO context. We also need access to the mainloop object and utilise a flag to tell when the timer should just quit (since timers cannot be removed externally in GLib).

```
/* Application state.
```

```
   Contains the necessary state to control the application lifetime
   and use LibOSSO. */
typedef struct {
  /* The GMainLoop that will run our code. */
  GMainLoop* mainloop;
  /* LibOSSO context that is necessary to use LibOSSO functions. */
  osso_context_t* ossoContext;
  /* Flag to tell the timer that it should stop running. Also utilized
     to tell the main program that the device is already in Flight-
     mode and the program shouldn't continue startup. */
  gboolean running;
} ApplicationState;
```

Listing 3.12: Application state (libosso-flashlight/flashlight.c)

All of the setup and start logic is implemented in `setupAppState`, and contains a significant number of steps that are all necessary:

```
/**
 * Utility to setup the application state.
 *
 * 1) Initialize LibOSSO (this will connect to D-Bus)
 * 2) Create a mainloop object
 * 3) Register the device state change callback
 *    The callback will be called once immediately on registration.
 *    The callback will reset the state->running to FALSE when the
 *    program needs to terminate so we'll know whether the program
 *    should run at all. If not, display an error dialog.
 *    (This is the case if the device will be in "Flight"-mode when
 *    the program starts.)
 * 4) Register the timer callback (which will keep the screen from
 *    blanking).
 * 5) Un-blank the screen.
 * 6) Display a dialog to the user (on the background) warning about
 *    battery drain.
 * 7) Send the first "delay backlight dimming" command.
 *
 * Returns TRUE when everything went ok, FALSE when caller should call
 * releaseAppState and terminate. The code below will print out the
 * errors if necessary.
 */
static gboolean setupAppState(ApplicationState* state) {

  osso_return_t result;

  g_assert(state != NULL);

  g_print(PROGNAME ":setupAppState starting\n");

  /* Zero out the state. Has the benefit of setting all pointers to
     NULLs and all gbooleans to FALSE. This is useful when we'll need
     to determine what to release later. */
  memset(state, 0, sizeof(ApplicationState));

  g_print(PROGNAME ":setupAppState Initializing LibOSSO\n");

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

  state->ossoContext = osso_initialize(ProgName, "1.0", FALSE, NULL);
  if (state->ossoContext == NULL) {
    g_printerr(PROGNAME ": Failed to initialize LibOSSO\n");
    return FALSE;
```

```
  }

  g_print(PROGNAME ":setupAppState Creating a GMainLoop object\n");
  /* Create a new GMainLoop object, with default context (NULL) and
     initial "running"-state set to FALSE. */
  state->mainloop = g_main_loop_new(NULL, FALSE);
  if (state->mainloop == NULL) {
    g_printerr(PROGNAME ": Failed to create a GMainLoop\n");
    return FALSE;
  }

  g_print(PROGNAME
          ":setupAddState Adding hw-state change callback.\n");
  /* The callback will be called immediately with the state, so we
     need to know whether we're in offline mode to start with. If so,
     the callback will set the running-member to FALSE (and we'll
     check it below). */
  state->running = TRUE;
  /* In order to receive information about device state and changes
     in it, we register our callback here.

     Parameters for the osso_hw_set_event_cb():
       osso_context_t* :  LibOSSO context object to use.
       osso_hw_state_t* : Pointer to a device state type that we're
                          interested in. NULL for "all states".
       osso_hw_cb_f* :    Function to call on state changes.
       gpointer :         User-data passed to callback. */
  result = osso_hw_set_event_cb(state->ossoContext,
                                NULL, /* We're interested in all. */
                                deviceStateChanged,
                                state);
  if (result != OSSO_OK) {
    g_printerr(PROGNAME
               ":setupAppState Failed to get state change CB\n");
    /* Since we cannot reliably know when to terminate later on
       without state information, we will refuse to run because of the
       error. */
    return FALSE;
  }
  /* We're in "Flight" mode? */
  if (state->running == FALSE) {
    g_print(PROGNAME ":setupAppState In offline, not continuing.\n");
    displayExitMessage(state, ProgName " not available in Offline mode"
        );
    return FALSE;
  }

  g_print(PROGNAME ":setupAppState Adding blanking delay timer.\n");
  if (g_timeout_add(45000,
                    (GSourceFunc)delayBlankingCallback,
                    state) == 0) {
    /* If g_timeout_add returns 0, it signifies an invalid event
       source id. This means that adding the timer failed. */
    g_printerr(PROGNAME ": Failed to create a new timer callback\n");
    return FALSE;
  }

  /* Un-blank the display (will always succeed in the SDK). */
  g_print(PROGNAME ":setupAppState Unblanking the display\n");
  result = osso_display_state_on(state->ossoContext);
  if (result != OSSO_OK) {
    g_printerr(PROGNAME ": Failed in osso_display_state_on (%s)\n",
```

```
                ossoErrorStr(result));
    /* If the RPC call fails, odds are that nothing else will work
       either, so we decide to quit instead. */
    return FALSE;
  }

  /* Display a "Note"-dialog with a WARNING icon.
     The Dialog is MODAL, so user cannot do anything with the stylus
     until the Ok is selected, or the Back-key is pressed. */

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

  /* Other icons available:
      OSSO_GN_NOTICE: For general notices.
     OSSO_GN_WARNING: For warning messages.
       OSSO_GN_ERROR: For error messages.
        OSSO_GN_WAIT: For messages about "delaying" for something (an
                      hourglass icon is displayed).
                   5: Animated progress indicator. */

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

  g_print(PROGNAME ":setupAppState Displaying Note dialog\n");
  result = osso_system_note_dialog(state->ossoContext,
                                   /* UTF-8 text into the dialog */
                                   "Started " ProgName ".\n"
                       "Please remember to stop it when you're done, "
                                "in order to conserve battery power.",
                                   /* Icon to use */
                                   OSSO_GN_WARNING,
                                   /* We're not interested in the RPC
                                      return value. */
                                   NULL);
  if (result != OSSO_OK) {
    g_error(PROGNAME ": Error displaying Note dialog (%s)\n",
            ossoErrorStr(result));
  }
  g_print(PROGNAME ":setupAppState Requested for the dialog\n");

  /* Then delay the blanking timeout so that our timer callback has a
     chance to run before that. */
  delayDisplayBlanking(state);

  g_print(PROGNAME ":setupAppState Completed\n");

  /* State set up. */
  return TRUE;
}
```

Listing 3.13: Setting up of the application state (libosso-flashlight/flashlight.c)

The callback to handle device state changes is registered with the `osso_hw_set_event_cb` and we also see how to force the backlight on (which is necessary so that the backlight dimming delay will accomplish something). We also register the timer callback which then will start firing away after 45 seconds and will keep delaying the backlight dimming and do the first delay so that the backlight isn't dimmed right away.

The callback function will always receive the new "hardware state" as well as the user-data. It is also somewhat interesting to note that just by registering the callback, it will be triggered immediately. This will happen even before we

have started our mainloop in order to tell the application the initial state of the device when the application starts. We utilise this to determine whether the device is already in flight-mode and refuse to start if it is. Since we don't always know whether the mainloop is active or not (the callback can be triggered later on as well), we also utilise an additional flag to communicate the timer callback that it should quit (eventually).

```
/**
 * This callback will be called by LibOSSO whenever the device state
 * changes. It will also be called right after registration to inform
 * the current device state.
 *
 * The device state structure contains flags telling about conditions
 * that might affect applications, as well as the mode of the device
 * (for example telling whether the device is in "in-flight"-mode).
 */
static void deviceStateChanged(osso_hw_state_t* hwState,
                               gpointer data) {
  ApplicationState* state = (ApplicationState*)data;

  g_print(PROGNAME ":deviceStateChanged Starting\n");

  printDeviceState(hwState);

  /* If device is in/going into "flight-mode" (called "Offline" on
     some devices), we stop our operation automatically. Obviously
     this makes flashlight useless (as an application) if someone gets
     stuck in a dark cargo bay of a plane with snakes.. But we still
     need a way to shut down the application and react to device
     changes, and this is the easiest state to test with.

     Note that since offline mode will terminate network connections,
     you will need to test this on the device itself, not over ssh. */
  if (hwState->sig_device_mode_ind == OSSO_DEVMODE_FLIGHT) {
    g_print(PROGNAME ":deviceStateChanged In/going into offline.\n");

    /* Terminate the mainloop.
       NOTE: Since this callback is executed immediately on
             registration, the mainloop object is not yet "running",
             hence calling quit on it will be ineffective! _quit only
             works when the mainloop is running. */
    g_main_loop_quit(state->mainloop);
    /* We also set the running to correct state to fix the above
       problem. */
    state->running = FALSE;
  }
}
```

Listing 3.14: Handling device state changes (libosso-flashlight/flashlight.c)

The `printDeviceState` is an utility function to decode the device state structure that the callback will be invoked with. The state contains the device mode, but also `gboolean` flags which tell the application to adapt to the environment in other ways (like memory pressure and other indicators):

```
/* Small macro to return "YES" or "no" based on given parameter.
   Used in printDeviceState below. YES is in capital letters in order
   for it to "stand out" in the program output (since it's much
   rarer). */
#define BOOLSTR(p) ((p)?"YES":"no")
```

```
/**
 * Utility to decode the hwstate structure and print it out in human
 * readable format. Mainly useful for debugging on the device.
 *
 * The mode constants unfortunately are not documented in LibOSSO.
 */
static void printDeviceState(osso_hw_state_t* hwState) {

  gchar* modeStr = "Unknown";

  g_assert(hwState != NULL);

  switch(hwState->sig_device_mode_ind) {
    case OSSO_DEVMODE_NORMAL:
      /* Non-flight-mode. */
      modeStr = "Normal";
      break;
    case OSSO_DEVMODE_FLIGHT:
      /* Power button -> "Offline mode". */
      modeStr = "Flight";
      break;
    case OSSO_DEVMODE_OFFLINE:
      /* Unknown. Even if all connections are severed, this mode will
         not be triggered. */
      modeStr = "Offline";
      break;
    case OSSO_DEVMODE_INVALID:
      /* Unknown. */
      modeStr = "Invalid(?)";
      break;
    default:
      /* Leave at "Unknown". */
      break;
  }
  g_print(
    "Mode: %s, Shutdown: %s, Save: %s, MemLow: %s, RedAct: %s\n",
    modeStr,
    /* Set to TRUE if the device is shutting down. */
    BOOLSTR(hwState->shutdown_ind),
    /* Set to TRUE if our program has registered for autosave and
       now is the moment to save user data. */
    BOOLSTR(hwState->save_unsaved_data_ind),
    /* Set to TRUE when device is running low on memory. If possible,
       memory should be freed by our program. */
    BOOLSTR(hwState->memory_low_ind),
    /* Set to TRUE when system wants us to be less active. */
    BOOLSTR(hwState->system_inactivity_ind));
}
```

Listing 3.15: Decoding the device state structure (libosso-flashlight/flashlight.c)

The delaying of the display blanking is achieved with an utility function of LibOSSO (`osso_display_blanking_pause`) which is implemented in a separate function since it's called from multiple places:

```
/**
 * Utility to ask the device to pause the screen blanking timeout.
 * Does not return success/status.
 */
static void delayDisplayBlanking(ApplicationState* state) {
```

```
  osso_return_t result;

  g_assert(state != NULL);

  result = osso_display_blanking_pause(state->ossoContext);
  if (result != OSSO_OK) {
    g_printerr(PROGNAME ":delayDisplayBlanking. Failed (%s)\n",
               ossoErrorStr(result));
    /* But continue anyway. */
  } else {
    g_print(PROGNAME ":delayDisplayBlanking RPC succeeded\n");
  }
}
```

Listing 3.16: Delaying the backlight blanking (libosso-flashlight/flashlight.c)

The timer callback will normally just ask the blanking delay to be further extended, but will also check whether the program is shutting down (by using the `running` field in the application state). If the application is indeed shutting down, the timer will ask itself to be removed from the timer queue by returning `FALSE`:

```
/**
 * Timer callback that will be called within 45 seconds after
 * installing the callback and will ask the platform to defer any
 * display blanking for another 60 seconds.
 *
 * This will also prevent the device from going into suspend
 * (according to LibOSSO documentation).
 *
 * It will continue doing this until the program is terminated.
 *
 * NOTE: Normally timers shouldn't be abused like this since they
 *       bring the CPU out from power saving state. Because the screen
 *       backlight dimming might be activated again after 60 seconds
 *       of receiving the "pause" message, we need to keep sending the
 *       "pause" messages more often than every 60 seconds. 45 seconds
 *       seems like a safe choice.
 */
static gboolean delayBlankingCallback(gpointer data) {
  ApplicationState* state = (ApplicationState*)data;

  g_print(PROGNAME ":delayBlankingCallback Starting\n");

  g_assert(state != NULL);
  /* If we're not supposed to be running anymore, return immediately
     and ask caller to remove the timeout. */
  if (state->running == FALSE) {
    g_print(PROGNAME ":delayBlankingCallback Removing\n");
    return FALSE;
  }

  /* Delay the blanking for a while still (60 seconds). */
  delayDisplayBlanking(state);

  g_print(PROGNAME ":delayBlankingCallback Done\n");

  /* We want the same callback to be invoked from the timer
     launch again, so we return TRUE. */
  return TRUE;
}
```

Listing 3.17: Application state reactive timer callback (libosso-flashlight/flashlight.c)

We also have a small utility function that will be used to display an exit message (we have two ways of exiting):

```
/**
 * Utility to display an "exiting" message to user using infoprint.
 */
static void displayExitMessage(ApplicationState* state,
                               const gchar* msg) {

  osso_return_t result;

  g_assert(state != NULL);

  g_print(PROGNAME ":displayExitMessage Displaying exit message\n");
  result = osso_system_note_infoprint(state->ossoContext, msg, NULL);
  if (result != OSSO_OK) {
    /* This is rather harsh, since we terminate the whole program if
       the infoprint RPC fails. It is used to display messages at
       program exit anyway, so this isn't a critical issue. */
    g_error(PROGNAME ": Error doing infoprint (%s)\n",
            ossoErrorStr(result));
  }
}
```

Listing 3.18: Using infoprint for non-modal notifications (libosso-flashlight/flashlight.c)

And finally we come to the application state tear-down function which will release all the resources that have been allocated by the setup function.

```
/**
 * Release all resources allocated by setupAppState in reverse order.
 */
static void releaseAppState(ApplicationState* state) {

  g_print(PROGNAME ":releaseAppState starting\n");

  g_assert(state != NULL);

  /* First set the running state to FALSE so that if the timer will
     (for some reason) be launched, it will remove itself from the
     timer call list. This shouldn't be possible since we are running
     only with one thread. */
  state->running = FALSE;

  /* Normally we would also release the timer, but since the only way
     to do that is from the timer callback itself, there's not much we
     can do about it here. */

  /* Remove the device state change callback. It is possible that we
     run this even if the callback was never installed, but it is not
     harmful. */
  if (state->ossoContext != NULL) {
    osso_hw_unset_event_cb(state->ossoContext, NULL);
  }
```

```
  /* Release the mainloop object. */
  if (state->mainloop != NULL) {
    g_print(PROGNAME ":releaseAppState Releasing mainloop object.\n");
    g_main_loop_unref(state->mainloop);
    state->mainloop = NULL;
  }

  /* Lastly, free up the LibOSSO context. */
  if (state->ossoContext != NULL) {
    g_print(PROGNAME ":releaseAppState De-init LibOSSO.\n");
    osso_deinitialize(state->ossoContext);
    state->ossoContext = NULL;
  }
  /* All resources released. */
}
```

Listing 3.19: Releasing the application state (libosso-flashlight/flashlight.c)

The Makefile for this program contains no surprises, so is not shown here. We build the program and then run it in the SDK:

```
[sbox-DIABLO_X86: ~/libosso-flashlight] > run-standalone.sh ./flashlight
flashlight:main Starting
flashlight:setupAppState starting
flashlight:setupAppState Initializing LibOSSO
flashlight:setupAppState Creating a GMainLoop object
flashlight:setupAddState Adding hw-state change callback.
flashlight:deviceStateChanged Starting
Mode: Normal, Shutdown: no, Save: no, MemLow: no, RedAct: no
flashlight:setupAppState Adding blanking delay timer.
flashlight:setupAppState Unblanking the display
flashlight:setupAppState Displaying Note dialog
flashlight:setupAppState Requested for the dialog
flashlight:delayDisplayBlanking RPC succeeded
flashlight:setupAppState Completed
flashlight:main Starting mainloop processing
/dev/dsp: No such file or directory
flashlight:delayBlankingCallback Starting
flashlight:delayDisplayBlanking RPC succeeded
flashlight:delayBlankingCallback Done
...
```

Starting the flashlight application


Figure 3.3: The application will display a modal dialog when it starts, to remind the user of the ramifications.

Since the SDK does not contain indications about backlight operations, you will not "notice" that the application is running in your screen. From the debugging messages you will see that it is. It just takes 45 seconds between each timer callback launch (and for new debug messages to appear).

## 3.5 Simulating device mode changes

In order to test the flashlight application without requiring a device, it is useful to know how to "simulate" device mode changes in the SDK. From the standpoint of LibOSSO (and programs that use it), it will feel and look exactly like it does on a device. When LibOSSO will receive the D-Bus signal, it will be exactly the same signal as it would be on a device. D-Bus signals are covered more thoroughly later on.

To see how this works, start flashlight in one session and leave it running (you might want to dismiss the modal dialog). Then open another session and send the signal using the system bus that signifies a device mode change (below).

```
[sbox-DIABLO_X86: ~/libosso-flashlight] > run-standalone.sh \
 dbus-send --system --type=signal /com/nokia/mce/signal \
 com.nokia.mce.signal.sig_device_mode_ind string:'flight'
```

Simulating a device mode change signal in the SDK

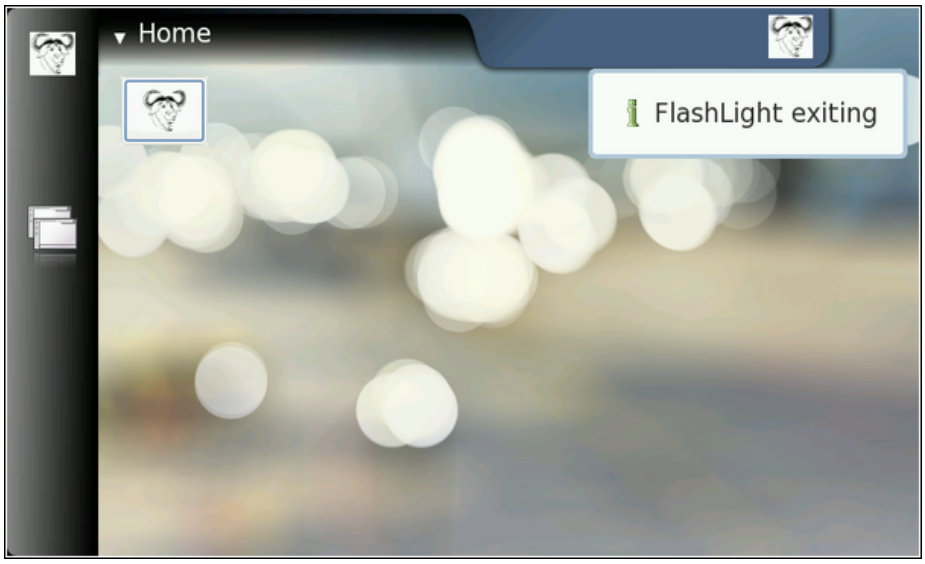
The argument for the signal is a string, stating the new device mode. They are defined (for LibOSSO) in the LibOSSO source code (src/osso-hw.c) which you can get with `apt-get source libosso`. The start of that file also defines other useful D-Bus well-known names, object paths and interfaces as well as method names relating to hardware and system-wide conditions. Here we are

only interested in switching the device mode from 'normal' to 'flight' and then back (see below).

```
flashlight:delayBlankingCallback Starting
flashlight:delayDisplayBlanking RPC succeeded
flashlight:delayBlankingCallback Done
...
flashlight:deviceStateChanged Starting
Mode: Flight, Shutdown: no, Save: no, MemLow: no, RedAct: no
flashlight:deviceStateChanged In/going into offline.
flashlight:main Out of main loop (shutting down)
flashlight:displayExitMessage Displaying exit message
flashlight:releaseAppState starting
flashlight:releaseAppState Releasing mainloop object.
flashlight:releaseAppState De-init LibOSSO.
flashlight:main Quitting
```

Reaction of flashlight to flight mode

Once the signal is sent, it will eventually be converted into a callback call from LibOSSO and our `deviceStateChanged` function gets to run. It will notice that the device mode is now `FLIGHT_MODE` and shutdown flashlight.

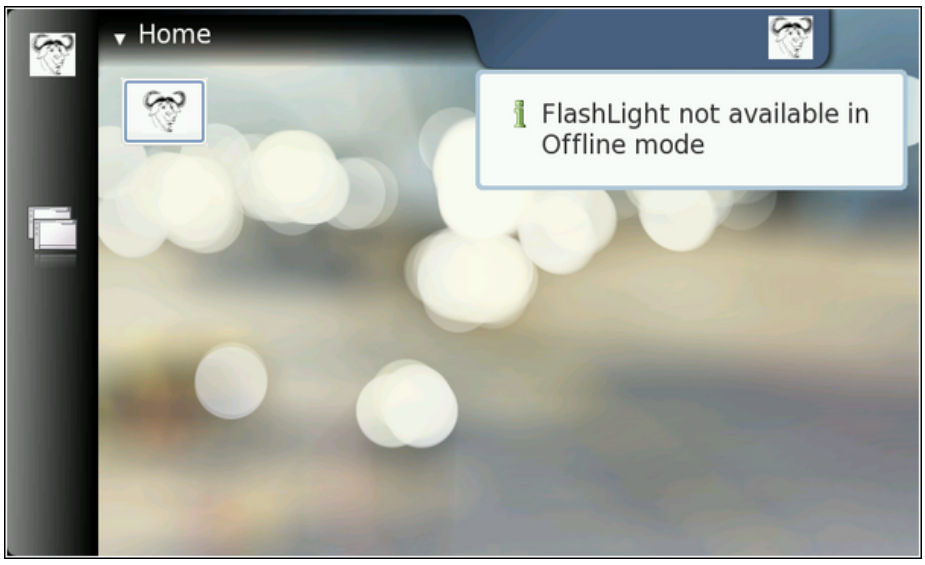



If you now start flashlight again, you will notice something peculiar. It will still see that the "device" is still in flight-mode. How convenient! This allows us to test the rest of the code paths remaining in our application (when it refuses to start if the device is already in flight mode).

```
[sbox-DIABLO_X86: ~/libosso-flashlight] > run-standalone.sh ./flashlight
flashlight:main Starting
flashlight:setupAppState starting
flashlight:setupAppState Initializing LibOSSO
flashlight:setupAppState Creating a GMainLoop object
flashlight:setupAddState Adding hw-state change callback.
flashlight:deviceStateChanged Starting
Mode: Flight, Shutdown: no, Save: no, MemLow: no, RedAct: no
flashlight:deviceStateChanged In/going into offline.
flashlight:setupAppState In offline, not continuing.
flashlight:displayExitMessage Displaying exit message
flashlight:main Setup failed, doing cleanup
flashlight:releaseAppState starting
flashlight:releaseAppState Releasing mainloop object.
flashlight:releaseAppState De-init LibOSSO.
flashlight:main Terminating with failure
```

Flashlight will refuse to start when device is in flight mode

In this case, the user is displayed the cause why flashlight cannot be started since not displaying any feedback to the user would be quite rude.




In order to return the "device" back to normal mode, you'll need to send the same signal as before (`sig_device_mode_ind`) but with the argument `normal`.

```
[sbox-DIABLO_X86: ~/libosso-flashlight] > run-standalone.sh  \
 dbus-send --system --type=signal /com/nokia/mce/signal  \
 com.nokia.mce.signal.sig_device_mode_ind string:'normal'
```

Restoring device mode back to normal

# Chapter 4

# Using the GLib wrappers for D-Bus

## 4.1 Introduction to GObjects

In order to support runtime binding of GTK+ widgets to interpreted languages, an somewhat complicated system for implementing object oriented machinery for C was developed. Depending on your particular viewpoint, this system is either called GObject or GType. GType is the low-level runtime type system which is used to implement GObjects and GObjects are the implementations of objects using the GType framework. For an short introduction about GObjects, please see the "Wikipedia entry on GType":http://en.wikipedia.org/wiki/GType. GObject/GType is part of GLib, but one might note that most of GLib is useable without the GType part of the library. In fact, the GObject/GType functionality is separated into its own library (`libgobject`).

We will be interested in the GType in order to implement a very simple object that will be published over the D-Bus. This also means that we can forego some of the inherent complexity involved in implementing a fully fledged GObject. For this reason, while our object will be usable over the D-Bus, it might not have all the support required to use it directly as a GObject (full dynamic type registration and properties are missing).

Our implementation will be a simple non-inheriting stand-alone class, implementing an interface to access and modify two private members: `value1` and `value2`, first of which is an 32-bit signed integer, and the second a `gdouble`.

We'll need to implement the per-class constructor as well as the per-object constructor. Both of these will be quite short for the first version of the implementation.

## 4.2 D-Bus interface definition using XML

Since our primary objective is to make the object available over D-Bus, we'll start by covering one of the easiest way of achieving this: the`dbus-bindings-tool`. The tool will generate a lot of the bindings code for both client and server side. As its input, it uses an XML file describing the interface for the service that

we're implementing.

We'll start by describing one method in XML. Each method is described with a separate `method` element, whose `name` attribute is the name of the method to be generated (this name will be copied into the generated stub code automatically by the tool). The first method is `setvalue1`, which will get one argument, `new_value`, which is an 32-bit signed integer:

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

Listing 4.1: A single D-Bus method definition (glib-dbus-sync/value-dbus-interface.xml)

Each argument needs to be defined explicitly with the `arg` element. The `type` attribute is required, since it will define the data type for the argument. Arguments are sometimes called parameters when used with D-Bus methods. Each argument needs to specify the "direction" of the argument. Parameters for method calls are "going into" the service, hence the correct content for the `direction` attribute is `in`. Return values from method calls are "coming out" of the service. Hence their direction will be `out`. If a method call doesn't return any value (returns `void`), then no argument with the direction `out` needs to be specified.

Also note that D-Bus by itself, does not limit the number of return arguments. C language supports only one return value from a function, but a lot of the higher level languages do not have this restriction.

The following argument types are supported for D-Bus methods (with respective closest types in GLib):

- `b`: boolean (`gboolean`)
- `y`: 8-bit unsigned integer (`guint8`)
- `q`/`n`: 16-bit unsigned/signed integer (`guint16`/`gint16`)
- `u`/`i`: 32-bit unsigned/signed integer (`guint32`/`gint32`)
- `t`/`x`: 64-bit unsigned/signed integer (`guint64`/`gint64`)
- `d`: IEEE 754 double precision floating point number (`gdouble`)
- `s`: UTF-8 encoded text string with NUL termination (only one NUL allowed) (`gchar*` with additional restrictions)
- `a`: Array of the following type specification (case-dependent)
- `o`/`g`/`r`/`(`/`)`/`v`/`e`/`{`/`}`: Complex types, please see the official D-Bus documentation on type signatures.

From the above list we see that `setvalue1` will accept one 32-bit signed integer argument (`new_value`). The name of the argument will affect the generated stub code prototypes (not the implementation) but is quite useful for documentation and also for D-Bus introspection (which will be covered later).

We next have the interface specification of another method: `getvalue1`, which will return the current integer value of the object. It has no method call parameters (no arguments with direction="`in`") and only returns one 32-bit signed integer:

```
    <!-- getvalue1(): returns the first value (int) -->
    <method name="getvalue1">
      <arg type="i" name="cur_value" direction="out"/>
    </method>
```

Listing 4.2: A method that returns data (glib-dbus-sync/value-dbus-interface.xml)

Naming of the return arguments is also supported in D-Bus (as above). This will not influence the generated stub code, but serves as additional documentation.

We'll need to bind the methods to a specific (D-Bus) interface, and this is achieved by placing the `method` elements within an `interface` element. The interface `name` -attribute is optional, but very much recommended (otherwise the interface becomes unnamed and provides less useful information on introspection).

You can implement multiple interfaces in the same object, and if this would be the case, you'd then list multiple `interface` elements within the `node` element. The `node` element is the "top-level" element in any case. In our case, we only implement one explicit interface (the binding tools will add the introspection interfaces automatically, so specifying them is not necessary in the XML). And so, we end up with the minimum required interface XML file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<node>
  <interface name="org.maemo.Value">
    <!-- getvalue1(): returns the first value (int) -->
    <method name="getvalue1">
      <arg type="i" 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>
```

Listing 4.3: Minimal valid interface specification for the two methods

The minimal interface specification is then extended by adding the correct reference to the proper DTD. This will allow validation tools to work automatically with the XML file. We also add methods to manipulate the second value. The full interface file will now contain comments, describing the purpose of the interface and the methods. This is highly recommended if you plan to publish your interface at some point, as the bare XML does not carry semantic information.

```
<?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 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>

  </interface>
</node>
```

Listing 4.4: The interface definition for the simplistic dual-value service (glib-dbus-sync/value-dbus-interface.xml)

When dealing with automatic code generation, it is quite useful to also automate testing of the "source files" (XML in this case). One important validation technique for XML is verifying for well-formedness (all XML files need to satisfy the rules in XML spec 1.0). Another is validating the structure of XML (elements are nested correctly, that only correct elements are present and that element attributes and data is legal). Structural validation rules are described

by a DTD (Document Type Definition) document for the XML format that your file is supposed to adhere to. The DTD is specified in the XML,within the `DOCTYPE` processing directive.

This is still not perfect, as DTD validation can only check for syntax and structure, but not meaning/semantics.

We'll add a target called `checkxml` to the Makefile so that it can be run whenever we want to check the validity of our interface XML.

```
# 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).

# ... Listing cut for brevity ...

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

# ... Listing cut for brevity ...

# 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
```

Listing 4.5: Integrating xmllint into makefiles (glib-dbus-sync/Makefile)

```
[sbox-DIABLO_X86: ~/glib-dbus-sync] > make checkxml
value-dbus-interface.xml checks out ok
```

Running the validation target on a valid interface specification

Just to demonstrate what kind of error messages to expect when there are problems in the XML, we modify the valid interface specification slightly by adding one invalid element (`invalidElement`) and by removing one starting tag (`method`).

```
[sbox-DIABLO_X86: ~/glib-dbus-sync] > make checkxml
value-dbus-interface.xml:36: element invalidElement: validity error :
  No declaration for element invalidElement
    </invalidElement>
             ^
value-dbus-interface.xml:53: parser error :
  Opening and ending tag mismatch: method line 39 and interface
  </interface>
             ^
value-dbus-interface.xml:54: parser error :
  Opening and ending tag mismatch: interface line 22 and node
</node>
      ^
value-dbus-interface.xml:55: parser error :
  Premature end of data in tag node line 21

^
make: *** [checkxml] Error 1
```

The first error (`validity error`) is detected because the file doesn't adhere to the DTD. The other errors (`parser errors`) are detected because the file is no longer a well-formed XML document.

If you'd have your makefile targets depend on `checkxml`, you could integrate the validation into the process of your build. However, as was noted before, it might not be always the best solution.

## 4.3 Generating automatic stub code

We can now proceed to generate the "glue" code that will implement the mapping from GLib into D-Bus. We will use the generated code later on, but it is instructive to see what the `dbus-binding-tool` program generates.

We'll expand the Makefile to invoke the tool whenever the interface XML changes and store the resulting glue code separately for both client and server.

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

# ... Listing cut for brevity ...

# 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 \
    $< > $@

# ... Listing cut for brevity ...

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

Listing 4.6: dbus-binding-tool support for make (glib-dbus-sync/Makefile)

We pass two parameters for the `dbus-binding-tool` program. The `--prefix` parameter is used to tell what text should be prefixed to all generated structure and function names. This will help avoid name-space collisions when we pull the generated glue files back into our programs. We'll use `value_object` since it seems like a logical prefix for our project. You will probably want to use a prefix that isn't used in your code (even in the object implementation in server). This way you don't risk reusing the same names that are generated with the tool.

The second parameter will select what kind of output the tool will generate. At the moment, the tool only supports generating GLib/D-Bus bindings, but this might change in the future. Also, we need to select which "side" of the D-Bus we're generating the bindings for. The `-client` side is for code that wishes to use GLib to access the Value object implementation over D-Bus. The `-server` side is respectively for the implementation of the Value object.

Running the tool will result in the two header files:

```
[sbox-DIABLO_X86: ~/glib-dbus-sync] > make value-server-stub.h value-client-stub.h
dbus-binding-tool --prefix=value_object --mode=glib-server \
  value-dbus-interface.xml > value-server-stub.h
dbus-binding-tool --prefix=value_object --mode=glib-client \
  value-dbus-interface.xml > value-client-stub.h
[sbox-DIABLO_X86: ~/glib-dbus-sync] > ls -la value*stub.h
-rw-rw-r-- 1 user user  5184 Nov 21 14:02 value-client-stub.h
-rw-rw-r-- 1 user user 10603 Nov 21 14:02 value-server-stub.h
```

Creating the glue code files

We'll start from the object implementation in a bit, but first let's see what the tool left us with, starting with the server stub file:

```
/* Generated by dbus-binding-tool; do not edit! */

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

#include <dbus/dbus-glib.h>
static const DBusGMethodInfo dbus_glib_value_object_methods[] = {
  { (GCallback) value_object_getvalue1,
    dbus_glib_marshal_value_object_BOOLEAN__POINTER_POINTER, 0 },
  { (GCallback) value_object_getvalue2,
    dbus_glib_marshal_value_object_BOOLEAN__POINTER_POINTER, 47 },
  { (GCallback) value_object_setvalue1,
    dbus_glib_marshal_value_object_BOOLEAN__INT_POINTER, 94 },
  { (GCallback) value_object_setvalue2,
    dbus_glib_marshal_value_object_BOOLEAN__DOUBLE_POINTER, 137 },
};

const DBusGObjectInfo dbus_glib_value_object_object_info = {
  0,
  dbus_glib_value_object_methods,
  4,
"org.maemo.Value\0getvalue1\0S\0cur_value\0O\0F\0N\0i\0\0"
"org.maemo.Value\0getvalue2\0S\0cur_value\0O\0F\0N\0d\0\0"
"org.maemo.Value\0setvalue1\0S\0new_value\0I\0i\0\0"
"org.maemo.Value\0setvalue2\0S\0new_value\0I\0d\0\0\0",
"\0",
"\0"
};
```

Listing 4.7: The method table and object introspection data (glib-dbus-sync/value-server-stub.h)

We're interested in the method table mainly because it lists the names of the functions that we need to implement: `value_object_getvalue1`, `value_object_getvalue2`, `value_object_setvalue1` and `value_object_setvalue2`. Each entry in the table consists of a function address, and the function to use to marshal data from/to GLib/D-Bus (the functions that start with `dbus_glib_marshal_*`). The marshaling functions are defined in the same file, but were omitted from the listing above.

Marshaling in its most generic form means the conversion of parameters or arguments from one format to another, in order to make two different parameter passing conventions compatible. It is a common feature found in almost all RPC mechanisms. Since GLib has its own type system (which we'll see shortly) and D-Bus its own, it would be very tedious to write the conversion code manually. This is where the binding generation tool really helps.

The other interesting feature of the above listing is the `_object_info` structure. It will be passed to the D-Bus daemon when we're ready with our object and wish to publish it on the bus (so that clients may invoke methods on it). The very long string (that contains binary zeroes) is the compact format of the interface specification. You will see similarities with the names in the string with the names of the interface, methods and arguments that we declared in the XML. It is also an important part of the D-Bus introspection mechanism which we'll cover at the end of this chapter.

As the snippet says at the very first line, it should never be edited manually. This holds true while you use the XML file as the source of your interface. It is also possible to use the XML only once, when you start your project, and then just start copy-pasting the generated glue code around while discarding the XML file and `dbus-binding-tool`. Needless to say, this makes maintenance of the interface much more difficult and isn't really recommended. We will not edit the generated stub code in this material.

We'll next continue with the server implementation for the functions that are called via the method table.

## 4.4 Creating a simple GObject for D-Bus

We start with the per-instance and per-class state structures for our object. The per-class structure contains only the bare minimum contents which is required from all classes in GObject. The per-instance structure also contains the required "parent object" state (`GObject`) but also includes the two internal values (`value1` and `value2`) with which we'll be working for the rest of this example:

```
/* This defines the per-instance state.

   Each GObject must start with the 'parent' definition so that common
   operations that all GObjects support can be called on it. */
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;

/* Per class state.

   For the first Value implementation we only have the bare minimum,
   that is, the common implementation for any GObject class. */
typedef struct {
  /* The parent class state. */
  GObjectClass parent;
} ValueObjectClass;
```

Listing 4.8: Per class and per instance state structures for the Value (glib-dbus-sync/server.c)

We then continue by defining convenience macros in a way expected for all GTypes. The `G_TYPE_`-macros are defined in GType and include the magic by which our object implementation doesn't need to know the internal specifics of

GType so much. The GType macros are described in the GObject API reference for GType at maemo.org.

We'll be using some of the macros internally in our implementation later on.

```
/* Forward declaration of the function that will return the GType of
   the Value implementation. Not used in this program since we only
   need to push this over the D-Bus. */
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))

/* Utility macro to define the value_object GType structure. */
G_DEFINE_TYPE(ValueObject, value_object, G_TYPE_OBJECT)
```

Listing 4.9: Macros for the Value object (and class) for convenience (glib-dbus-sync/server.c)

After the macros, we come to the instance initialisation and class initialisation functions, of which the class initialisation function contains the integration call into GLib/D-Bus:

```
/**
 * 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"

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

```c
/**
 * 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
 *
 * 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) {

  dbg("Called");

  g_assert(klass != NULL);

  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 */
}
```

Listing 4.10: Per-instance and per-class initialisation for Value (glib-dbus-sync/server.c)

The `dbus_g_object_type_install_info` will take a pointer to the structure describing the D-Bus integration (`dbus_glib_value_object_object_info`), which is generated by `dbus-bindings-tool`. This function will create all the necessary runtime information for our GType, so we don't need to worry about the details. It will also attach the introspection data to our GType so that D-Bus introspection may return information on the interface that the object will implement.

We next implement the get and set functions, which allows us to inspect the interface as well. Note that the names of the functions and their prototypes is ultimately dictated by `dbus-bindings-tool` generated stub header files. This means that if you change your interface XML sufficiently, your code will fail to build (since the generated stubs will yield different prototypes):

```
/**
 * 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);

  /* Change the value. */
  obj->value1 = valueIn;

  /* 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);

  obj->value2 = valueIn;

  return TRUE;
}

/**
 * Function that gets executed on "getvalue1".
 */
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;
}
```

Listing 4.11: Implementation for the get and set methods (glib-dbus-sync/server.c)

The GLib/D-Bus wrapper logic will implement all of the parameter conversion necessary from D-Bus into your functions, so you'll only need to handle the GLib corresponding types (`gint` and `gdouble`). The method implementations will always receive an object reference to the object as their first parameter and a pointer to a place where to store new GError objects if your method decides an error should be created. This error would then be propagated back to the caller of the D-Bus method. Our simple get/set examples will never set errors, so we're free to ignore the last parameter.

Note that returning values is not done via the conventional C way (by using `return someVal`), but instead return values are written via the given pointers. The return value of the method is always a `gboolean` signifying success or failure. If you decide to return failure (`FALSE`), you'll also need to create and setup an GError object and store its address to the `error` location.

## 4.5 Publishing a GType on the D-Bus

Once our implementation is complete, we'll need to publish an instance of the class on to the D-Bus. This will be done inside the `main` of the server example and involves doing a D-Bus method call on the bus.

So that we don't need to change both the server and client if we decide to change the object or well-known names later, we put them into a common header file that will be used by both:

```
/* 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"
```

Listing 4.12: Symbolic constants that both server and client will use for name space related parameters (glib-dbus-sync/common-defs.h)

The decision to use `/GlobalValue` as the object path is based on clarity only. Most of the time you'd use something like `/org/maemo/Value` instead.

Before using any of the GType functions, we'll need to initialise the runtime system by calling `g_type_init`. This will create the built in types and setup all the machinery necessary for creating custom types as well. If you're using GTK+, then the function is called for you automatically when you initialise GTK+. Since we're only using GLib, we need to call the function manually.

After initialising the GType system, we then proceed by opening a connection to the session bus, which we'll use for the remainder of the publishing sequence:

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

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

int main(int argc, char** argv) {

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

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

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

  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
        );
  }
```

Listing 4.13: Initialising GType and connecting to the session bus (glib-dbus-sync/server.c)

In order for prospective clients to find the object on the session bus, we'll need to attach the server to a well-known name. This is done with the `RequestName` method call on the D-Bus server (over D-Bus). In order to target the server, we'll need to create a GLib/D-Bus proxy object first:

```
  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. */
  }
```

Listing 4.14: Publishing one Value object onto the D-Bus (glib-dbus-sync/server.c)

The `dbus_g_proxy_call` function is used to do synchronous method calls in GLib/D-Bus wrappers, and in our case, we'll use it to run the two argument `RequestName` method call. The method returns one value (and `uint32`) which encodes the result of the well-known name registration.

One needs to be careful with the order and correctness of the parameters to the function call, as it is easy to get something wrong and the C compiler cannot really check for parameter type validity here.

After the successful name registration, we're finally now ready to create an instance of the ValueObject and publish it on the D-Bus:

```
  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 maemo 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");
```

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

}
```

Listing 4.15: Publishing one Value object onto the D-Bus (glib-dbus-sync/server.c)

And after this, `main` will enter into the main loop, and will serve client requests coming over the D-Bus until the server is terminated. Note that all the callback registration is done automatically by the GLib/D-Bus wrappers on object publication, so you don't need to worry about them.

Implementing the dependencies and rules for the server and the generated stub code will give this snippet:

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

# ... Listing cut for brevity ...

# 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) $(CFLAGS) -DPROGNAME=\"$(basename $@)\" -c $< -o $@
```

Listing 4.16: Publishing one Value object onto the D-Bus (glib-dbus-sync/server.c)

When implementing makefiles that separate compilation from linking, it's not possible to pass the target name (automatic variable `$@` directly as the `PROGNAME`-define (since that would expand into `server.o` and would look slightly silly when we prefix all our messages with the name). Instead, we use a GNU make function (`basename`) that will strip out any prefix and suffix out of the parameter. This way our `PROGNAME` will be set to `server`.

We then build the server and start it:

```
[sbox-DIABLO_X86: ~/glib-dbus-sync] > 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-sync] > run-standalone.sh ./server
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: 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)
```

We then use `dbus-send` to test out the implementation details from the server. This is done in the same session (for simplicity) by first suspending the server with `Ctrl+z` and then continuing running it with the `bg` shell built-in

command. This is done so that you can more easily see the reaction of the server to each `dbus-send` command.

We start by testing the `getvalue1` and `setvalue1` methods:

```
[Ctrl+z]
[1]+  Stopped                 run-standalone.sh ./server
[sbox-DIABLO_X86: ~/glib-dbus-sync] > bg
[1]+ run-standalone.sh ./server &
[sbox-DIABLO_X86: ~/glib-dbus-sync] > run-standalone.sh dbus-send \
 --type=method_call --print-reply --dest=org.maemo.Platdev_ex \
 /GlobalValue org.maemo.Value.getvalue1
server:value_object_getvalue1: Called (internal value1 is 0)
method return sender=:1.15 -> dest=:1.20
 int32 0
[sbox-DIABLO_X86: ~/glib-dbus-sync] > run-standalone.sh dbus-send \
 --type=method_call --print-reply --dest=org.maemo.Platdev_ex \
 /GlobalValue org.maemo.Value.setvalue1 int32:5
server:value_object_setvalue1: Called (valueIn=5)
method return sender=:1.15 -> dest=:1.21
[sbox-DIABLO_X86: ~/glib-dbus-sync] > run-standalone.sh dbus-send \
 --type=method_call --print-reply --dest=org.maemo.Platdev_ex \
 /GlobalValue org.maemo.Value.getvalue1
server:value_object_getvalue1: Called (internal value1 is 5)
method return sender=:1.15 -> dest=:1.22
 int32 5
```

Testing `value1` (32-bit integer)

And continue by testing the double state variable with `getvalue2` and `setvalue2` methods:

```
[sbox-DIABLO_X86: ~/glib-dbus-sync] > run-standalone.sh dbus-send \
 --type=method_call --print-reply --dest=org.maemo.Platdev_ex \
 /GlobalValue org.maemo.Value.getvalue2
server:value_object_getvalue2: Called (internal value2 is 0.000)
method return sender=:1.15 -> dest=:1.23
 double 0
[sbox-DIABLO_X86: ~/glib-dbus-sync] > run-standalone.sh dbus-send \
 --type=method_call --print-reply --dest=org.maemo.Platdev_ex \
 /GlobalValue org.maemo.Value.setvalue2 double:42.0
server:value_object_setvalue2: Called (valueIn=42.000)
method return sender=:1.15 -> dest=:1.24
[sbox-DIABLO_X86: ~/glib-dbus-sync] > run-standalone.sh dbus-send \
 --type=method_call --print-reply --dest=org.maemo.Platdev_ex \
 /GlobalValue org.maemo.Value.getvalue2
server:value_object_getvalue2: Called (internal value2 is 42.000)
method return sender=:1.15 -> dest=:1.25
 double 42
```

Testing `value2` (double)

We now have a fully functional D-Bus service implementation (albeit a very simple one).

We'll next proceed to utilise the service from a client.

## 4.6 Using the GLib/D-Bus wrapper from a client

By using the generated client stub file, it's now possible to write the client that will invoke the methods on the Value object. The D-Bus method calls could also be done "manually" (either with GLib/D-Bus functions, or even by using libdbus directly, but latter is discouraged).

The `dbus-bindings-tool` (when run with the `--mode=glib-client` parameter) will generate functions for each of the interface methods, and the functions will handle data marshaling operations internally.

Two generated stub functions are presented below (we'll be using them shortly):

```
/* Generated by dbus-binding-tool; do not edit! */

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

static
#ifdef G_HAVE_INLINE
inline
#endif
gboolean
org_maemo_Value_getvalue1 (DBusGProxy *proxy, gint* OUT_cur_value,
                                              GError **error)

{
  return dbus_g_proxy_call (proxy, "getvalue1", error, G_TYPE_INVALID,
                            G_TYPE_INT, OUT_cur_value, G_TYPE_INVALID);
}

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

static
#ifdef G_HAVE_INLINE
inline
#endif
gboolean
org_maemo_Value_setvalue1 (DBusGProxy *proxy, const gint IN_new_value,
                                              GError **error)

{
  return dbus_g_proxy_call (proxy, "setvalue1", error, G_TYPE_INT,
                            IN_new_value, G_TYPE_INVALID,
                            G_TYPE_INVALID);
}
```

Listing 4.17: Generated wrapper functions for setvalue1 and getvalue1 (glib-dbus-sync/value-client-stub.h)

The two functions presented above are both *blocking* which means that they will wait for the result to arrive over the D-Bus and only then return to the caller. The generated stub code also includes *asynchronous* functions (their names end with `_async`), but we'll cover using them later on.

For now, it's important to notice how the prototypes of the functions are named and what are the parameters that they expect one to pass to them.

The `org_maemo_Value` -prefix is taken from the interface XML file, from the `name` attribute of the `interface` element. All dots will be converted into underscores (since C reserves the dot character for other uses), but otherwise the name will be preserved (barring dashes in the name).

The rest of the function name will be the method name for each method defined in the interface XML file.

The first parameter for all the generated stub functions will always be a pointer to a `DBusProxy` object, which we'll need to use with the GLib/D-Bus wrapper functions. After the proxy, a list of method parameters is passed. The binding tool will prefix the parameter names with either `IN_` or `OUT_` depending on the "direction" of the parameter. Rest of the parameter name is taken from the `name` attributed of the `arg` element for the method, or if not given, will be

automatically generated as `arg0`, `arg1` and so forth. Input parameters will be passed as values (unless they're complex or strings, in which case they'll be passed as pointers). Output parameters are always passed as pointers.

The functions will always return a `gboolean`, indicating failure or success, and if they fail, they will also create and set the `error` pointer to an `GError` -object which can then be checked for the reason for the error (unless the caller passed a NULL pointer for `error`, in which case the error object won't be created).

```
#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"
```

Listing 4.18: The generated stub code is pulled into the client (glib-dbus-sync/client.c)

This will allow our client code to use the stub code directly as follows:

```
/**
 * 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.
 */
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;
}
```

Listing 4.19: Using the stubs in client code (glib-dbus-sync/client.c)

What is left is connecting to the correct D-Bus, creating a GProxy object which we'll do in our test program:

```
/**
 * 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) {
  /* 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);
  }

  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 4.20: Connecting to the bus creating a proxy and starting the mainloop (glib-dbus-sync/client.c)

Integrating the client into the Makefile is done the same way as we did for the server before:

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

# ... Listing cut for brevity ...

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

Listing 4.21: Integrating the client into the Makefile (glib-dbus-sync/Makefile)

After building the client, we then start it and let it execute in the same terminal session where we still have the server running:

```
[sbox-DIABLO_X86: ~/glib-dbus-sync] > make client
dbus-binding-tool --prefix=value_object --mode=glib-client \
 value-dbus-interface.xml > value-client-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=\"client\" \
 -c client.c -o client.o
cc client.o -o client -ldbus-glib-1 -ldbus-1 -lgobject-2.0 -lglib-2.0
[sbox-DIABLO_X86: ~/glib-dbus-sync] > run-standalone.sh ./client
client:main Connecting to Session D-Bus.
client:main Creating a GLib proxy object for Value.
client:main Starting main loop (first timer in 1s).
server:value_object_setvalue1: Called (valueIn=-80)
client:timerCallback Set value1 to -80
server:value_object_setvalue2: Called (valueIn=-120.000)
client:timerCallback Set value2 to -120.000
server:value_object_setvalue1: Called (valueIn=-70)
client:timerCallback Set value1 to -70
server:value_object_setvalue2: Called (valueIn=-110.000)
client:timerCallback Set value2 to -110.000
server:value_object_setvalue1: Called (valueIn=-60)
client:timerCallback Set value1 to -60
...
[Ctrl+c]
[sbox-DIABLO_X86: ~/glib-dbus-sync] > fg
run-standalone.sh ./server
[Ctrl+c]
```

Building and running the client. The client will print the status message after doing the RPC.

Since the client will normally run forever, we terminate it, and then move the server to the foreground so that it can also be terminated. This concludes our first GLib/D-Bus example, but for more information about the GLib D-Bus wrappers, please consult maemo.org.

## 4.7 D-Bus introspection

D-Bus supports a mechanism by which programs can interrogate the bus for existing well-known names, and then get the interfaces implemented by the objects available behind the well-known names. This mechanism is called _introspection_ in D-Bus terminology.

The main goal of supporting introspection in D-Bus is allowing dynamic bindings to be made with high-level programming languages. This way the language wrappers for D-Bus can be more intelligent automatically (assuming they utilise the introspection interface). The GLib-wrappers do not use the introspection interface.

Introspection is achieved with three D-Bus methods: `ListNames`, `GetNameOwner` and `Introspect`. The destination object must support the introspection interface in order to provide this information. If you use the `dbus-bindings-tool`, and register your GObject correctly, your service will automatically support introspection.

D-Bus (at this moment) does not come with introspection utilities, but some are available from other sources. One simple program is the "DBus Inspector", which is written in Python and uses the Python D-Bus bindings and GTK+. If you plan to write your own tool, you need to prepare to parse XML data, since that is the format of results that the `Introspect` method returns.

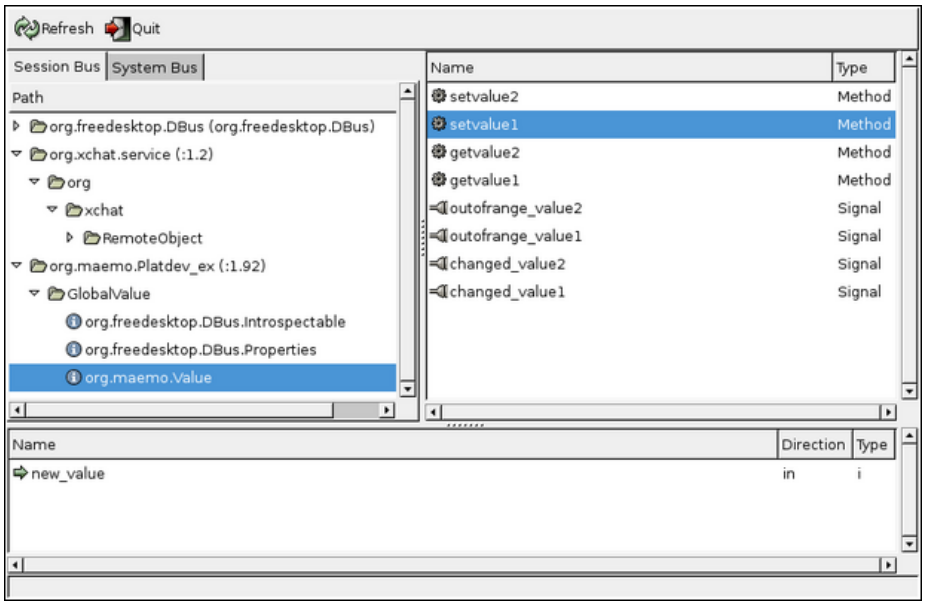



Figure 4.1: Using DBUS Inspector on GlobalValue on a desktop system. Note that the version of GlobalValue used here also implements signals, which we'll cover next.

Introspection can also be useful when trying to find out what are the different interfaces and methods available for use on a system. You'll just have to remember that not all D-Bus services actually implement the introspection interface. You can still get their well-known names, but their interface descriptions will come up empty when using `Introspect`.

# Chapter 5

# Implementing and using D-Bus signals

## 5.1 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`).

## 5.2 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 5.1: 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.

## 5.3 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 5.2: 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 5.3: 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 5.4: 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 5.5: 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:

```
/**
 * 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 5.6: 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`).

## 5.4 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 5.7: 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 5.8: 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.

## 5.5 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.

# Chapter 6

# Asynchronous GLib D-Bus

## 6.1 Asynchronicity in D-Bus clients

So far all the RPC method calls that we've implemented have been "fast" in that their execution does not depend on access to slow services or external resources. In real life however, it is quite likely that you won't be able to provide some service immediately, but will have to wait for some external service to complete before completing your own method call.

The GLib wrappers provide a version of doing method calls where the call will be launched (almost) immediately, and a callback will be executed when the method call will return (either with a return value, or an error).

Using the asynchronous wrappers is important when your program needs to update some kind of status, or be reactive to the user (via a GUI or other interface). Otherwise the program would block waiting for the RPC method to return, and won't be able to refresh the GUI or screen when required. An alternative solution would be to use separate threads which would run the synchronous methods, but synchronisation between threads will become an issue and debugging threaded programs is much harder than single threaded ones. Also, implementation of threads might be sub optimal in some environments. These are the reasons why we won't be covering the thread scenario here.

We will simulate slow running RPC methods by adding a delay into the server method implementations so that it will become clear why asynchronous RPC mechanisms are important. As signals by their nature are asynchronous as well, they don't add anything to our example this time. In order to simplify the code listings, we drop signal support from the asynchronous clients (the server still contains them and will emit them).

## 6.2 Slow test server

The only change on the server side is the addition of delays into each of the RPC methods (`setvalue1`, `setvalue2`, `getvalue1` and `getvalue2`). This delay is added to the start of each function as follows:

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

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

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) {
```

Listing 6.1: Slowing down setvalue1 (glib-dbus-async/server.c)

Building the server is done as before, but we'll notice the delay when we call an RPC method:

```
[sbox-DIABLO_X86: ~/glib-dbus-async] > run-standalone.sh ./server &
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-async] > time run-standalone.sh dbus-send \
 --type=method_call --print-reply --dest=org.maemo.Platdev_ex \
 /GlobalValue org.maemo.Value.getvalue1
server:value_object_getvalue1: Called (internal value1 is 0)
server:value_object_getvalue1: Delaying operation
method return sender=:1.54 -> dest=:1.56
 int32 0

real    0m5.066s
user    0m0.004s
sys     0m0.056s
```

Testing the delayed server version

Above we use the `time` shell built-in command. It will run the given command while measuring the wall clock time (aka real time) and time used while executing the code and system calls. In our case, we're only interested in the `real` time. The method call will delay for about 5 seconds as it should. The delay (even if given with microsecond resolution) is always approximate and longer than the requested amount. Exact delay will depend on many factors, most of which you cannot influence directly.

We'll next experiment with a likely scenario where another method call comes along while the first one is still being executed. This is best tested by just repeating the sending command twice, but running the first one on the background (so that the shell doesn't wait it to complete first). The server is still running on the background from the previous test:

```
[sbox-DIABLO_X86: ~/glib-dbus-async] > time run-standalone.sh dbus-send \
 --type=method_call --print-reply --dest=org.maemo.Platdev_ex \
 /GlobalValue org.maemo.Value.getvalue1 &
[2] 17010
server:value_object_getvalue1: Called (internal value1 is 0)
server:value_object_getvalue1: Delaying operation
[sbox-DIABLO_X86: ~/glib-dbus-async] > time run-standalone.sh dbus-send \
 --type=method_call --print-reply --dest=org.maemo.Platdev_ex \
 /GlobalValue org.maemo.Value.getvalue1
method return sender=:1.54 -> dest=:1.57
 int32 0

real    0m5.176s
user    0m0.008s
sys     0m0.092s
server:value_object_getvalue1: Called (internal value1 is 0)
server:value_object_getvalue1: Delaying operation
method return sender=:1.54 -> dest=:1.58
 int32 0

real    0m9.852s
user    0m0.004s
sys     0m0.052s
```

Delays accumulating

What we can see from the above output is that the first client is delayed for about 5 seconds, while the second client (which was launched shortly after the first) is already delayed by much longer period. This is to be expected as the server can only process one request at a time and will delay each request by 5 seconds.

We'll cover some server concurrency issues later, but for now, we want our clients to be able to continue their "normal work" while they wait for the response from the server. Since we're dealing with example code, "normal work" for our clients will be just waiting for the response, while blocking on incoming events (converted into callbacks). However, if the example programs would be graphical, the asynchronous approach would make it possible for them to react to user input. D-Bus by itself does not support cancellation of method calls once processing has started on the server side, so adding cancellation support would require a separate method call to the server. Since the server only handles one operation at a time, the current server cannot support method call cancellations at all.

## 6.3 Asynchronous method calls using stubs

When you run the `glib-bindings-tool`, it will already generate the necessary wrapping stubs to support launching asynchronous method calls. What is then left to do is implementing the callback functions correctly, processing the return errors and launching the method call.

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

Listing 6.2: The client still pulls the generated stub code in like before (glib-dbus-async/client-stubs.c)

The client has been simplified so that it now only operates on `value1`. The callback that will be called from the stub code is presented next:

```c
/**
 * 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");
  }
}
```

Listing 6.3: Callback to use when the RPC method completes (glib-dbus-async/client-stubs.c)

Since the method call does not return any data, the parameters for the callback are at minimum (you'll always get those three). Handling errors must be done within the callback since errors could be delayed from the server and not visible immediately at launch time. Note that the callback will not terminate the program on errors. We do this on purpose in order to demonstrate common asynchronous problems below. The `timestamp` function is a small utility function to return a pointer to a string representing the number of seconds since the program started (useful to visualise the order of the different asynchronous events below).

```c
/**
 * 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;
}
```

Listing 6.4: The timer callback that will now use the async launching stub code (glib-dbus-async/client-stubs.c)

Using the stub code is rather simple. For each generated synchronous version of a method wrapper, there will also be a `_async` version of the call. The main difference with the parameters is the removal of the GError pointer (since errors will be handled in the callback) and addition of the callback function to use when the method will complete, time out or encounter an error.

The `main` function remains the same from previous client examples (a once per second timer will be created and run from the mainloop until the program is terminated).

## 6.4 Problems with asynchronicity

When the simple test program is built and run, we'll see that everything starts off quite well. But at some point problems start to appear:

```
[sbox-DIABLO_X86: ~/glib-dbus-async] > make client-stubs
dbus-binding-tool --prefix=value_object --mode=glib-client \
  value-dbus-interface.xml > value-client-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=\"client-stubs\" \
 -c client-stubs.c -o client-stubs.o
cc client-stubs.o -o client-stubs -ldbus-glib-1 -ldbus-1 -lgobject-2.0 -lglib-2.0
[sbox-DIABLO_X86: ~/glib-dbus-async] > run-standalone.sh ./client-stubs
client-stubs:main Connecting to Session D-Bus.
client-stubs:main Creating a GLib proxy object for Value.
client-stubs: 0.00:main Starting main loop (first timer in 1s).
client-stubs: 1.00:timerCallback launching setvalue1
client-stubs: 1.00:timerCallback setvalue1 launched
server:value_object_setvalue1: Called (valueIn=-80)
server:value_object_setvalue1: Delaying operation
client-stubs: 2.00:timerCallback launching setvalue1
client-stubs: 2.00:timerCallback setvalue1 launched
client-stubs: 3.01:timerCallback launching setvalue1
client-stubs: 3.01:timerCallback setvalue1 launched
client-stubs: 4.01:timerCallback launching setvalue1
client-stubs: 4.01:timerCallback setvalue1 launched
client-stubs: 5.02:timerCallback launching setvalue1
client-stubs: 5.02:timerCallback setvalue1 launched
server:value_object_emitSignal: Emitting signal id 0, with message 'value1'
server:value_object_setvalue1: Called (valueIn=-70)
server:value_object_setvalue1: Delaying operation
client-stubs: 6.01:setValue1Completed
client-stubs       SUCCESS
client-stubs: 6.02:timerCallback launching setvalue1
client-stubs: 6.02:timerCallback setvalue1 launched
client-stubs: 7.02:timerCallback launching setvalue1
client-stubs: 7.02:timerCallback setvalue1 launched
...
client-stubs:25.04:timerCallback launching setvalue1
client-stubs:25.04:timerCallback setvalue1 launched
server:value_object_emitSignal: Emitting signal id 0, with message 'value1'
server:value_object_setvalue1: Called (valueIn=-30)
server:value_object_setvalue1: Delaying operation
client-stubs:26.03:setValue1Completed
client-stubs       SUCCESS
...
client-stubs:30.05:timerCallback launching setvalue1
client-stubs:30.05:timerCallback setvalue1 launched
...
client-stubs:36.04:setValue1Completed
client-stubs       ERROR: Did not receive a reply. Possible causes include:
 the remote application did not send a reply, the message bus security policy
 blocked the reply, the reply timeout expired, or the network connection was
 broken.
server:value_object_emitSignal: Emitting signal id 0, with message 'value1'
server:value_object_setvalue1: Called (valueIn=-10)
server:value_object_setvalue1: Delaying operation
client-stubs:36.06:timerCallback launching setvalue1
client-stubs:36.06:timerCallback setvalue1 launched
client-stubs:37.04:setValue1Completed
client-stubs       ERROR: Did not receive a reply. Possible causes include:
 the remote application did not send a reply, the message bus security policy
 blocked the reply, the reply timeout expired, or the network connection was
 broken.
[Ctrl+c]
[sbox-DIABLO_X86: ~/glib-dbus-async] >
server:value_object_emitSignal: Emitting signal id 0, with message 'value1'
server:value_object_setvalue1: Called (valueIn=30)
server:value_object_setvalue1: Delaying operation
server:value_object_emitSignal: Emitting signal id 0, with message 'value1'
server:value_object_setvalue1: Called (valueIn=40)
server:value_object_setvalue1: Delaying operation
server:value_object_emitSignal: Emitting signal id 0, with message 'value1'
server:value_object_setvalue1: Called (valueIn=50)
server:value_object_setvalue1: Delaying operation
server:value_object_emitSignal: Emitting signal id 0, with message 'value1'
...
```

What happens above is rather subtle. The timer callback in the client will launch once per second and do the RPC method launch. The server however still has the 5 second delay for each method call in it. We see the successive launches going on without any responses for a while. The first response comes back at about 6 seconds since client has started. At this point the server already has 4 other outstanding method calls that it hasn't handled. Slowly the method calls are accumulating at the server end and it doesn't deal with them quickly enough to satisfy the client.

After about 30 seconds, we start seeing the `setValue1Completed` callback invoked, but with the method call failing. We have managed to trigger the method call timeout mechanism. After this point, all the method calls that have accumulated into the server (into a message queue) will fail in the client, since they all will now return late, even if the server actually does handle them.

Once we terminate the client, we'll see that the server is still happily continuing serving the requests, oblivious to the fact that there is no client to process the responses.

The above test demonstrates quite brutally that you need to design your services properly so that there is a clearly defined protocol what to do in case a method call is delayed. You also might want to design a notification protocol to tell clients that something has completed, instead of forcing them to time out. Using D-Bus signals is one way, but you need to take care not to generate signals when no one is listening for them. This can be done by only sending signals when an long operation will finish (assuming you have documented this as part of your service description).

One partial fix would be for the client to track and make sure that only one method call to one service it outstanding at any given time. So instead of just blindly launching the RPC methods, it should defer launching if it hasn't yet got a response from the server (and the call hasn't timed out).

This fix is not complete however, since the same problem will manifest itself once there are multiple clients running in parallel and requesting the same methods. The proper fix is to make the server capable of serving multiple requests in parallel. Some hints on how to do this are presented later on.

## 6.5 Asynchronous method calls using GLib wrappers

Sometimes the interface XML will be missing, so you cannot run the `dbus-bindings-tool` to generate the stub code. The GLib wrappers are generic enough for you to be able to build your own method calls when necessary.

It is often easiest to start with some known generated stub code to see which parts you could possibly reuse (with modifications). This is what we'll do last, in order to make a version of the asynchronous client that will work without the stub generator.

We start by taking a peek at the stub generated code for the `setvalue1` call (when used asynchronously):

```
typedef void (*org_maemo_Value_setvalue1_reply) (DBusGProxy *proxy,
                                                 GError *error,
                                                 gpointer userdata);

static void
org_maemo_Value_setvalue1_async_callback (DBusGProxy *proxy,
                                          DBusGProxyCall *call,
                                          void *user_data)
{
  DBusGAsyncData *data = user_data;
  GError *error = NULL;
  dbus_g_proxy_end_call (proxy, call, &error, G_TYPE_INVALID);
  (*(org_maemo_Value_setvalue1_reply)data->cb) (proxy, error,
                                                data->userdata);
  return;
}

static
#ifdef G_HAVE_INLINE
inline
#endif
DBusGProxyCall*
org_maemo_Value_setvalue1_async (DBusGProxy *proxy,
                                 const gint IN_new_value,
                              org_maemo_Value_setvalue1_reply callback,
                                 gpointer userdata)

{
  DBusGAsyncData *stuff;
  stuff = g_new (DBusGAsyncData, 1);
  stuff->cb = G_CALLBACK (callback);
  stuff->userdata = userdata;
  return dbus_g_proxy_begin_call (
    proxy, "setvalue1", org_maemo_Value_setvalue1_async_callback,
    stuff, g_free, G_TYPE_INT, IN_new_value, G_TYPE_INVALID);
}
```

Listing 6.5: Generated stub code for the asynchronous setvalue1 (glib-dbus-async/value-client-stub.h)

What is notable in the code snippet above is that the `_async` method will create a temporary small structure that will hold the pointer to the callback function, and a copy of the userdata pointer. This small structure will then be passed to `dbus_g_proxy_begin_call` along with the address of the generated callback wrapper function (`org_maemo_Value_setvalue1_async_callback`). The GLib async launcher will also take a function pointer to a function to use when the supplied "user-data" (in this case the small structure) will need to be disposed (after the call). Since it uses `g_new` to allocate the small structure, it passes `g_free` as the freeing function. Next comes the argument specification for the method call, which obeys the same rules as the LibOSSO ones before.

On RPC completion, the generated callback will be invoked, and it will get the real callback function pointer and the userdata as its "user-data" parameter. It will first collect the exit code for the call with `dbus_g_proxy_end_call` and unpacks the data and invokes the real callback. After returning, the GLib wrappers (which called the generated callback) will call `g_free` to release the small structure and the whole RPC launch will end.

We next re-implement pretty much the same logic, but also dispose of the

small structure, since we are going to implement our callback directly, not as a wrapper-callback (it also saves us from doing one memory allocation and one free).

We'll start with the RPC asynchronous launch code:

```
/**
 * 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;
}
```

Listing 6.6: Launching RPC methods using GLib functions without stubs (glib-dbus-async/client-glib.c)

And the callback that will be invoked on method call completion, timeouts or errors:

```
/**
 * 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");
  }
}
```

Listing 6.7: Handling the method response and ending the async RPC call (glib-dbus-async/client-glib.c)

We no longer need the generated stub code, so the dependency rules for the stubless GLib version will also be somewhat different:

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

Listing 6.8: Simpler dependencies for the stub code (glib-dbus-async/Makefile)

Since the example program logic hasn't changed from the previous version, testing `client-glib` is not presented here (you are of course free to test it yourself since the source code contains the fully working program). This version of the client will also launch the method calls without waiting for previous method calls to complete.

# Chapter 7

# Asynchronous GConf

## 7.1 Listening to changes in GConf

GConf as a central storage of configuration was presented before in the "maemo Application Development" material. We'll now see how to extend GConf to be more suitable in asynchronous work and especially when implementing services.

When you have simple configuration needs for your service, and want to support reacting to configuration changes in "real-time", you will want to use GConf. Also, people tend to use GConf when they're too lazy to write their own configuration file parsers (although there is a simple one in GLib) or too lazy to write the GUI part to change the settings. Our example program will simulate the first case and react to changes in a subset of GConf configuration name space when the changes happen.

Our application will be interested in two string values, one to set the device to use for communication (`connection`), and the other to set the communication parameters for the device (`connectionparams`). Since we'll be concentrating on just the change notifications, the program logic is simplified by omitting the proper setup code in the program. This means that you'll have to setup some values to the GConf keys prior to running the program. We'll be using the `gconftool-2` to do this, and have prepared a target in the Makefile just for this:

```
# Define a variable for this so that the GConf root may be changed
gconf_root := /apps/Maemo/platdev_ex

# ... Listing cut for brevity ...

# This will setup the keys into default values.
# It will first do a clear to remove any existing keys.
primekeys: clearkeys
  gconftool-2 --set --type string \
              $(gconf_root)/connection btcomm0
  gconftool-2 --set --type string \
              $(gconf_root)/connectionparams 9600,8,N,1

# Remove all application keys
clearkeys:
  @gconftool-2 --recursive-unset $(gconf_root)
```

```
# Dump all application keys
dumpkeys:
  @echo Keys under $(gconf_root):
  @gconftool-2 --recursive-list $(gconf_root)
```

Listing 7.1: Utility targets for setting up the GConf data for the application (gconf-listener/Makefile)

We prepare the keyspace next by running the `primekeys` target and verify that it succeeds by running the `dumpkeys` target:

```
[sbox-DIABLO_X86: ~/gconf-listener] > make primekeys
gconftool-2 --set --type string \
            /apps/Maemo/platdev_ex/connection btcomm0
gconftool-2 --set --type string \
            /apps/Maemo/platdev_ex/connectionparams 9600,8,N,1
[sbox-DIABLO_X86: ~/gconf-listener] > make dumpkeys
Keys under /apps/Maemo/platdev_ex:
 connectionparams = 9600,8,N,1
 connection = btcomm0
```

## 7.2 Implementing notifications on changes in GConf

We'll start with the necessary header information. The GConf namespace settings have been all implemented using `cpp` macros so that one can easily change the prefix of the name-space if required later on.

```
#include <glib.h>
#include <gconf/gconf-client.h>
#include <string.h> /* strcmp */

/* As per maemo Coding Style and Guidelines document, we use the
   /apps/Maemo/ -prefix.
   NOTE: There is no central registry (as of this moment) that you
         could check that your application name doesn't collide with
         other application names, so caution is advised! */
#define SERVICE_GCONF_ROOT "/apps/Maemo/platdev_ex"

/* We define the names of the keys symbolically so that we may change
   them later if necessary, and so that the GConf "root directory" for
   our application will be automatically prefixed to the paths. */
#define SERVICE_KEY_CONNECTION \
        SERVICE_GCONF_ROOT "/connection"
#define SERVICE_KEY_CONNECTIONPARAMS \
        SERVICE_GCONF_ROOT "/connectionparams"
```

Listing 7.2: The necessary headers and the name-space defines (gconf-listener/gconf-key-watch.c)

Our `main` starts innocently enough, by creating a GConf client object (that encapsulates the connection to the GConf daemon) and then displays the two values on output:

```
int main (int argc, char** argv) {
  /* Will hold reference to the GConfClient object. */
  GConfClient* client = NULL;
  /* Initialize this to NULL so that we'll know whether an error
     occurred or not (and we don't have an existing GError object
     anyway at this point). */
  GError* error = NULL;
```

```
  /* This will hold a reference to the mainloop object. */
  GMainLoop* mainloop = NULL;

  g_print(PROGNAME ":main Starting.\n");

  /* Must be called to initialize GType system. The API reference for
     gconf_client_get_default() insists.
     NOTE: Using gconf_init() is deprecated! */
  g_type_init();

  /* Create a new mainloop structure that we'll use. Use default
     context (NULL) and set the 'running' flag to FALSE. */
  mainloop = g_main_loop_new(NULL, FALSE);
  if (mainloop == NULL) {
    g_error(PROGNAME ": Failed to create mainloop!\n");
  }

  /* Create a new GConfClient object using the default settings. */
  client = gconf_client_get_default();
  if (client == NULL) {
    g_error(PROGNAME ": Failed to create GConfClient!\n");
  }

  g_print(PROGNAME ":main GType and GConfClient initialized.\n");

  /* Display the starting values for the two keys. */
  dispStringKey(client, SERVICE_KEY_CONNECTION);
  dispStringKey(client, SERVICE_KEY_CONNECTIONPARAMS);
```

Listing 7.3: Setting up the GConf connection (gconf-listener/gconf-key-watch.c)

The `dispStringKey` utility is rather simple as well, building on the GConf material that was covered in the "maemo Application Development" material:

```
/**
 * Utility to retrieve a string key and display it.
 * (Just as a small refresher on the API.)
 */
static void dispStringKey(GConfClient* client,
                          const gchar* keyname) {

  /* This will hold the string value of the key. It will be
     dynamically allocated for us, so we need to release it ourselves
     when done (before returning). */
  gchar* valueStr = NULL;

  /* We're not interested in the errors themselves (the last
     parameter), but the function will return NULL if there is one,
     so we just end in that case. */
  valueStr = gconf_client_get_string(client, keyname, NULL);

  if (valueStr == NULL) {
    g_error(PROGNAME ": No string value for %s. Quitting\n", keyname);
    /* Application terminates. */
  }

  g_print(PROGNAME ": Value for key '%s' is set to '%s'\n",
          keyname, valueStr);

  /* Normally one would want to use the value for something beyond
     just displaying it, but since this code doesn't, we release the
     allocated value string. */
```

```
  g_free(valueStr);
}
```

Listing 7.4: Utility to retrieve and display a string key value (gconf-listener/gconf-key-watch.c)

We next tell the GConf client to attach itself to a specific name-space part that we'll want to operate with:

```
  /**
   * Register directory to watch for changes. This will then tell
   * GConf to watch for changes in this namespace, and cause the
   * "value-changed"-signal to be emitted. We won't be using that
   * mechanism, but will opt to a more modern (and potentially more
   * scalable solution). The directory needs to be added to the
   * watch list in either case.
   *
   * When adding directories, you can sometimes optimize your program
   * performance by asking GConfClient to preload some (or all) keys
   * under a specific directory. This is done via the preload_type
   * parameter (we use GCONF_CLIENT_PRELOAD_NONE below). Since our
   * program will only listen for changes, we don't want to use extra
   * memory to keep the keys cached.
   *
   * Parameters:
   * - client: GConf-client object
   * - SERVICEPATH: the name of the GConf namespace to follow
   * - GCONF_CLIENT_PRELOAD_NONE: do not preload any of contents
   * - error: where to store the pointer to allocated GError on
   *          errors.
   */
  gconf_client_add_dir(client,
                       SERVICE_GCONF_ROOT,
                       GCONF_CLIENT_PRELOAD_NONE,
                       &error);

  if (error != NULL) {
    g_error(PROGNAME ": Failed to add a watch to GCClient: %s\n",
            error->message);
    /* Normally we'd also release the allocated GError, but since
       this program will terminate on g_error, we won't do that.
       Hence the next line is commented. */
    /* g_error_free(error); */

    /* When you want to release the error if it has been allocated,
       or just continue if not, use g_clear_error(&error); */
  }

  g_print(PROGNAME ":main Added " SERVICE_GCONF_ROOT ".\n");
```

Listing 7.5: Registering the interest in a part of the GConf name space (gconf-listener/gconf-key-watch.c)

Proceeding with the callback function registration, we have:

```
  /* Register our interest (in the form of a callback function) for
     any changes under the namespace that we just added.

     Parameters:
     - client: GConfClient object.
     - SERVICEPATH: namespace under which we can get notified for
```

```
                    changes.
     - gconf_notify_func: callback that will be called on changes.
     - NULL: user-data pointer (not used here).
     - NULL: function to call on user-data when notify is removed or
             GConfClient destroyed. NULL for none (since we don't
             have user-data anyway).
     - error: return location for an allocated GError.

     Returns:
     guint: an ID for this notification so that we could remove it
            later with gconf_client_notify_remove(). We're not going
            to use it so we don't store it anywhere. */
  gconf_client_notify_add(client, SERVICE_GCONF_ROOT,
                          keyChangeCallback, NULL, NULL, &error);
  if (error != NULL) {
    g_error(PROGNAME ": Failed to add register the callback: %s\n",
            error->message);
    /* Program terminates. */
  }

  g_print(PROGNAME ":main CB registered & starting main loop\n");
```

Listing 7.6: Registering the interest in a part of the GConf name space (gconf-listener/gconf-key-watch.c)

When dealing with regular desktop software, you could use multiple callback functions; one for each key to track. However, this would require you to implement multiple callback functions and this runs a risk of enlarging the size of your code. For this reason, the example code uses one callback function, which will internally multiplex between the two keys (by using `strcmp`):

```
/**
 * Callback called when a key in watched directory changes.
 * Prototype for the callback must be compatible with
 * GConfClientNotifyFunc (for ref).
 *
 * It will find out which key changed (using strcmp, since the same
 * callback is used to track both keys) and the display the new value
 * of the key.
 *
 * The changed key/value pair will be communicated in the entry
 * parameter. userData will be NULL (can be set on notify_add [in
 * main]). Normally the application state would be carried within the
 * userData parameter, so that this callback could then modify the
 * view based on the change. Since this program does not have a state,
 * there is little that we can do within the function (it will abort
 * the program on errors though).
 */
static void keyChangeCallback(GConfClient* client,
                              guint        cnxn_id,
                              GConfEntry*  entry,
                              gpointer     userData) {

  /* This will hold the pointer to the value. */
  const GConfValue* value = NULL;
  /* This will hold a pointer to the name of the key that changed. */
  const gchar* keyname = NULL;
  /* This will hold a dynamically allocated human-readable
     representation of the changed value. */
  gchar* strValue = NULL;
```

```
  g_print(PROGNAME ": keyChangeCallback invoked.\n");

  /* Get a pointer to the key (this is not a copy). */
  keyname = gconf_entry_get_key(entry);

  /* It will be quite fatal if after change we cannot retrieve even
     the name for the gconf entry, so we error out here. */
  if (keyname == NULL) {
    g_error(PROGNAME ": Couldn't get the key name!\n");
    /* Application terminates. */
  }

  /* Get a pointer to the value from changed entry. */
  value = gconf_entry_get_value(entry);

  /* If we get a NULL as the value, it means that the value either has
     not been set, or is at default. As a precaution we assume that
     this cannot ever happen, and will abort if it does.
     NOTE: A real program should be more resilient in this case, but
           the problem is: what is the correct action in this case?
           This is not always simple to decide.
     NOTE: You can trip this assert with 'make primekeys', since that
           will first remove all the keys (which causes the CB to
           be invoked, and abort here). */
  g_assert(value != NULL);

  /* Check that it looks like a valid type for the value. */
  if (!GCONF_VALUE_TYPE_VALID(value->type)) {
    g_error(PROGNAME ": Invalid type for gconfvalue!\n");
  }

  /* Create a human readable representation of the value. Since this
     will be a new string created just for us, we'll need to be
     careful and free it later. */
  strValue = gconf_value_to_string(value);

  /* Print out a message (depending on which of the tracked keys
     change. */
  if (strcmp(keyname, SERVICE_KEY_CONNECTION) == 0) {
    g_print(PROGNAME ": Connection type setting changed: [%s]\n",
            strValue);
  } else if (strcmp(keyname, SERVICE_KEY_CONNECTIONPARAMS) == 0) {
    g_print(PROGNAME ": Connection params setting changed: [%s]\n",
            strValue);
  } else {
    g_print(PROGNAME ":Unknown key: %s (value: [%s])\n", keyname,
            strValue);
  }

  /* Free the string representation of the value. */
  g_free(strValue);

  g_print(PROGNAME ": keyChangeCallback done.\n");
}
```

Listing 7.7: Registering the interest in a part of the GConf name space (gconf-listener/gconf-key-watch.c)

The complications in the above code rise from the fact that GConf communicates values using a GValue structure, which can carry values of any simple type. Since we don't completely trust GConf (or the user for that matter) to return the correct type for the value, we need to be extra careful and not assume

that it will always be a string. GConf also supports "default" values, which are communicated to the application using NULLs, so we need to guard against that as well. Especially since our application doesn't provide a schema for the configuration space which would contain the default values.

We'll next build and test the program. We'll start the program on the background so that we can use `gconftool-2` to see how the program will react to changing parameters:

```
[sbox-DIABLO_X86: ~/gconf-listener] > make
cc -I/usr/include/glib-2.0 -I/usr/lib/glib-2.0/include -I/usr/include/gconf/2 \
   -I/usr/include/dbus-1.0 -I/usr/lib/dbus-1.0/include -Wall -g \
   -DPROGNAME=\"gconf-key-watch\" gconf-key-watch.c -o gconf-key-watch \
   -lgconf-2 -ldbus-glib-1 -ldbus-1 -lgobject-2.0 -lglib-2.0
[sbox-DIABLO_X86: ~/gconf-listener] > run-standalone.sh ./gconf-key-watch &
[2] 21385
gconf-key-watch:main Starting.
gconf-key-watch:main GType and GConfClient initialized.
gconf-key-watch: Value for key '/apps/Maemo/platdev_ex/connection'
 is set to 'btcomm0'
gconf-key-watch: Value for key '/apps/Maemo/platdev_ex/connectionparams'
 is set to '9600,8,N,1'
gconf-key-watch:main Added /apps/Maemo/platdev_ex.
gconf-key-watch:main CB registered & starting main loop
[sbox-DIABLO_X86: ~/gconf-listener] > gconftool-2 --set --type string \
  /apps/Maemo/platdev_ex/connection ttyS0
gconf-key-watch: keyChangeCallback invoked.
gconf-key-watch: Connection type setting changed: [ttyS0]
gconf-key-watch: keyChangeCallback done.
[sbox-DIABLO_X86: ~/gconf-listener] > gconftool-2 --set --type string \
  /apps/Maemo/platdev_ex/connectionparams ''
gconf-key-watch: keyChangeCallback invoked.
gconf-key-watch: Connection params setting changed: []
gconf-key-watch: keyChangeCallback done.
```

The latter change is somewhat problematic (which your code needs to deal with as well). You need to decide how to react to values whose type is correct, but their values are not sensible. GConf in itself does not provide syntax checking for the values, nor any semantic checking support. It's recommended that you will only react to configuration changes that will pass some internal (to the application) logic that will check their validity, both at syntax level and also at semantic level.

One option would also be resetting the value back to a valid value whenever your program will detect an invalid value set attempt. This will lead to a lot of problems if the value is set programmatically from another program that will obey the same rule, so do not do it. Quitting your program on invalid values is also not an option that you should use, since the restricted environment doesn't provide many ways to inform the user that your program has quit.

An additional possible problem is having multiple keys that are all "related" to a single setting or action. You'll need to decide how to deal with changes across multiple GConf keys that are related, yet changed separately. The two key example code demonstrates the inherent problem: should the server re-initialise the (theoretic) connection when the `connection`-key is changed, or when the `connectionparams`-key is changed? If the connection will be re-initialized when either of the keys will change, then the connection will be re-initialized twice when both are changed "simultaneously" (user presses "Apply" on a settings dialog, or `gconftool-2` is run and sets both keys). You can see how this might be an even larger problem if instead of two keys, you would have 5 per connection. GConf (and the `GConfClient` GObject wrapper that we've been using) does not support "configuration set transactions", that would

allow setting and processing multiple related keys using an atomic model. The example program ignores this issue completely.

We'll next test how the program (which is still running) will react to other problematic situations:

```
[sbox-DIABLO_X86: ~/gconf-listener] > gconftool-2 --set --type int \
  /apps/Maemo/platdev_ex/connectionparams 5
gconf-key-watch: keyChangeCallback invoked.
gconf-key-watch: Connection params setting changed: [5]
gconf-key-watch: keyChangeCallback done.
[sbox-DIABLO_X86: ~/gconf-listener] > gconftool-2 --set --type boolean \
  /apps/Maemo/platdev_ex/connectionparams true
gconf-key-watch: keyChangeCallback invoked.
gconf-key-watch: Connection params setting changed: [true]
gconf-key-watch: keyChangeCallback done.
```

Our application survives the wrong type of value

We'll next remove the configuration keys while the program is still running:

```
[sbox-DIABLO_X86: ~/gconf-listener] > make clearkeys
gconf-key-watch: keyChangeCallback invoked.
gconf-key-watch[21403]: GLIB ERROR ** default -
  file gconf-key-watch.c: line 129 (keyChangeCallback):
  assertion failed: (value != NULL)
aborting...
/usr/bin/run-standalone.sh: line 11: 21403 Aborted (core dumped) $@"
[1]+  Exit 134 run-standalone.sh ./gconf-key-watch
```

But doesn't survive the removal of the key

Since the code (in the callback function) contains an assert that checks for non-NULL values, it will abort when we remove the key and that causes the value to go to NULL. So the abortion in the above case is expected.

# Chapter 8

# D-Bus server design issues

## 8.1 Definition of a server

When speaking about software, a server is commonly understood to mean some kind of software component that provides some service to its clients. In Linux, servers are usually implemented as _daemons_, which is a technical term for a process that has detached from the terminal session, and done other preparatory actions so that it will stay running on the background until it terminates (or is terminated).

Sometimes you might hear people refer to servers as _engines_, but it is a more generic term and normally isn't related directly to the way how a service is implemented (as a separate process, or as part of some library, directly used from within a client process). Broadly defined, an engine is the part of application that implements the functionality, but not the interface, of an application. In Model-View-Controller, it would be the Model.

Our servers so far have been running without daemonisation, in order to display debugging messages on the terminal/screen more easily. Often a server can be started with a `"--stay-on-foreground"` option (or `-f` or something similar), which means that they will not daemonise. This is a useful feature to have since it will allow you to use simpler outputting primitives when testing your software.

By default, when a server will daemonise, its output and input files will be closed, so reading user input (from the terminal session, not GUI) will fail, as will each output write (including `printf` and `g_print`).

## 8.2 Daemonisation

The objective of turning a process into a daemon is to detach it from its parent process and create a separate session for it. This is necessary so that parent termination doesn't automatically cause the termination of the server as well. There is a library call that will do most of the daemonisation work, called `daemon` but it is also instructive to see what is necessary (and common) to do when implementing the functionality yourself:

- `fork` the process so that the original process can be terminated and this

will cause the child process to move under the system `init` process.

- Create a new session for the child process with `setsid`.
- Possibly switch working directory to root (/) so that the daemon won't keep filesystems from being unmounted.
- Setup up a restricted `umask` so that directories and files that will be created by the daemon (or its child processes) will not create publicly accessible objects in the filesystem. In Internet Tablets this doesn't really apply since the devices only have one user.
- Close all standard I/O file descriptors (and preferable files too) so that if the terminal device will close (user logs out), that won't cause `SIGPIPE` signals to the daemon when it next accesses the file descriptors (by mistake or intentionally because of `g_print`/`printf`). It is also possible to reopen the file descriptors so that they'll be connected to a device which will just ignore all operations (like `/dev/null` which is used with `daemon`).

The `daemon` function allows you to select whether you want to change the directory and to close the open file descriptors and we'll utilise that in our servers in the following way:

```
#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
```

Listing 8.1: Adding daemonisation support to server.c based on a define (glib-dbus-sync/server.c)

This define is then available to the user inside the Makefile:

```
# -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)
```

Listing 8.2: Daemonisation as a compile time option driver by make (glib-dbus-sync/Makefile)

Combining the options so that `CFLAGS` is appended to the Makefile provided defaults allows the user to override the define as well:

```
[sbox-DIABLO_X86: ~/glib-dbus-sync] > CFLAGS='-UNO_DAEMON' 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 -UNO_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
```

Undoing a define using an environmental variable, without modifying the Makefile

Since all `-D` and `-U` options will be processed from left to right by `gcc`, this allows the `-UNO_DAEMON` to undefine the symbol that is preset in the Makefile. If the user doesn't know this technique, it's also possible to edit the Makefile directly. Grouping all additional flags that the user might be interested to the top of the Makefile will make this simpler (for the user).

Running the server with daemonisation support is done as before, but this time we leave out the `&` (don't wait for child exit) token for the shell:

```
[sbox-DIABLO_X86: ~/glib-dbus-sync] > run-standalone.sh ./server
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:main Registering it on the D-Bus.
server:main Ready to serve requests (daemonizing).
[sbox-DIABLO_X86: ~/glib-dbus-sync] >
```

Server running as daemon

Since server messages will not be visible any more, we need some other mechanism to find out that the server is still running:

```
[sbox-DIABLO_X86: ~/glib-dbus-sync] > ps aux | grep "./server" | grep -v pts
user 8982  0.0  0.1 2780 664 ? Ss 00:14 0:00 ./server
```

The slightly convoluted way of using `grep` was necessary to only list those lines of the `ps` report which have `./server` in them and remove the lines which do not have `pts` in them (so that we see processes which have no controlling terminals).

We could've used the client to test whether the server responds as well, but the above technique is slightly more general. If you have the `pstree` tool available, you could run it with `-pu` options to see how the processes relate to each other, and that our demonised server is running directly as a child of `init` (which was the objective of the `fork`).

## 8.3 Event loops and power consumption

Most modern CPUs (even for desktops and servers) allow multiple levels of power savings to be selected. Each of the levels will be progressively more power-conservative, but there is always a price involved. The deeper the power saving level required, the more time it normally takes to achieve it, and the more time it also takes to come out of it. In some CPUs it also requires special sequences of instructions to run, hence taking extra power itself.

All this means that changing the power state of the CPU should be avoided when possible. Obviously this is in contrast in the requirement to conserve

battery life, so in effect, what is needed is to require the attention of the CPU as rarely as possible.

One way at looking the problem field is contrasting event-based and polling-based programming. A code which continuously checks for some status and only occasionally does useful work, is clearly keeping the CPU from powering down properly. This model should be avoided at all costs, or at least restricting its use to bare minimum if no other solution is possible.

In contrast, event-based programming is usually based on execution of callback functions when something happens, without requiring a separate polling loop. This then leaves the question on how to trigger the callbacks so that they will be issued when something happens. Using timer callbacks might seem like a simple solution, so that we continuously (once per second or more often) will check for some status and then possibly react to the change in status. This model is undesirable as well, since the CPU won't be able to enter into the deep sleep modes, but fluctuate between full power and high-power states.

Most operating system kernels provide a mechanism (or multiple mechanisms) by which a process can be woken up when data is available, and kept off the running queue of the scheduler otherwise. The most common mechanism in Linux is based around the `select/poll` system calls which are useful when waiting for change in status for a set of file descriptors. Since most of the interesting things in Linux can be represented as a "file" (an object supporting `read` and `write` system calls), using `select` and `poll` is quite common. However, when you're writing software that uses GLib (implicitly like in GTK+ or explicitly like in our non-GUI examples), you will use the GMainLoop structure instead. Internally it will use the event mechanism available on your platform (`select`/`poll`/others), but your program will need to register callbacks, start the main loop execution and then just execute the callbacks as they come.

If you then have some file descriptors (network sockets, open files, etc), you may integrate them into the GMainLoop using GIOChannels (please see the GLib API reference on this).

This still leaves the question of using timers and callbacks that are triggered by timers. You should avoid them when:

- You plan to use the timer at high frequencies (> 1 Hz) for long periods of time (> 5 sec).
- There is a mechanism that will trigger a callback when something happens, instead of forcing you to poll for the status "manually" or re-execute a timer callback that does the checking.

As an example, the LibOSSO program (`FlashLight`) that was covered before, will have to use timers in order to keep the backlight active. However, the timer is very slow (only once per 45 seconds), so this is not a big issue. Also, in `flashlight`'s defense, the backlight is on all the time, so having a slow timer won't hurt battery life very much anyway.

As an another example, consider a long lasting download operation which proceeds slowly, but steadily. You might want to consider whether updating a progress bar after each small bit of data is received makes sense (it doesn't normally). Instead you will want to keep track of when was the last time when you updated a progress bar, and if enough time has passed since the last time, update the GUI. In some cases this will allow the CPU to be left in somewhat

lower power state than full-power and will allow it to fall back to sleep more quickly.

Having multiple separate programs running each having their own timers presents another interesting problem. Since the timer callback is not precise, at some time the system will be waking at a very high frequency, handling each timer separately (the frequency and the number of timers executing in the system is something which you cannot control from a single program, but instead is a system wide issue).

If you're planning a GUI program, it is fairly easy to avoid contributing to this problem, since you can get a callback from LibOSSO which will tell you when your program is "on top" and when not visible. When not visible, you won't need to update the GUI, especially with timer based progress indicators and similar.

Since servers do not have a GUI (and their visibility is not controlled by the window manager), such mechanism does not exist. One possible solution in this case would be avoiding using timers (or any resources for that matter) when the server does not have any active clients. Only use resources when you get a client connection, or need to actually do something. As soon as it becomes likely that your server won't be used again, you will want to release the resources (remove the timer callbacks, and so on).

If you can, try to utilise the D-Bus signals available on the system bus (or even the hardware state structure via LibOSSO) to throttle down activity based on the conditions in the environment. Even if you're doing a non-GUI server, you will want to listen to the system shutdown signals, as they will tell your process to shutdown gracefully.

All in all, designing for a dynamic low powered environment is not always simple. Four simple rules will hold for most cases (all of them being important):

- Avoid doing extra work when possible.
- Do it as fast as possible (while trying to minimise resource usage).
- Do it as rarely as possible.
- Keep only those resources allocated that you need to get the work done.

For GUI programs one will have to take into account the "graphical side" of things as well. Making a GUI that is very conservative in its power usage will most of the time be very simple, provide little excitement to users and might even look quite ugly. Your priorities might lie in a different direction.

## 8.4 Supporting parallel requests

The value object server with delays has one major deficiency: it can only handle one request at a time, while blocking the progress of all the other requests. This will be a problem if multiple clients will use the same server at the same time.

Normally one would add support for parallel requests by using some kind of multiplexing mechanism right on top of the message delivery mechanism (libdbus in this case).

One can group the possible solutions around three models:

- Launching a separate thread to handle each request. This might seem like an easy way out of the problem, but coordinating access to shared resources (object states in this case) between multiple threads is prone to cause synchronisation problems and makes debugging much harder. Also, performance of such an approach would depend on efficient synchronisation primitives in the platform (which might not always be available) as well as light weight thread creation and tear-down capabilities of the platform.

- Using an event-driven model that supports multiple event sources simultaneously and "wakes up" only when there is an event on any of the event sources. The `select` and `poll` (and `epoll` on Linux) are very often used in these cases. Using them will normally require an application design that is driven by the requirements of the system calls (i.e., it is very difficult to retrofit them into existing "linear" designs). However, the event based approach normally outperforms the thread approach, since there is no need for synchronisation (when implemented correctly) and there will only be one context to switch from the kernel and back (there will be extra contexts with threads). GLib provides an high-level abstraction on top of the low level event programming model, in the form of GMainLoop. One would use GIOChannel objects to represent each event source and register callbacks that will be triggered on the events.

- Using `fork` to create a copy of the server process so that the new copy will just handle one request and then terminate (or return to the pool of "servers"). The problem here is the process creation overhead and lack of implicit sharing of resources between the processes. One would have to arrange a separate mechanism for synchronisation and data sharing between the processes (using shared memory and proper synchronisation primitives). In some cases resource sharing is not actually required, or happens at some lower level (accessing files), so this model shouldn't be automatically ruled out even if it seems quite heavy at first. Many static content web-servers use this model because of its simplicity (and they don't need to share data between themselves).

The problem for the slow server however lies elsewhere: the GLib/D-Bus wrappers do not support parallel requests directly. Even using the `fork`-model would be problematic as there would be multiple processes accessing the same D-Bus connection. Also, this problem is not specific to the slow server only. You will bump into the same issues when using other high-level frameworks (like GTK+) whenever cannot complete something immediately because not all data is present in your application. In the latter case it is normally sufficient to use the GMainLoop/GIOChannel approach in parallel with GTK+ (since it uses GMainLoop internally anyway), but with GLib/D-Bus there's no mechanism which you could use to integrate your own multiplexing code (no suitable API exists).

In this case, the solution would be picking one of the above models, and then using libdbus functions directly. In effect, this would require a complete rewrite of the server, forgetting about the GType implementation and possibly creating a light-weight wrapper for integrating libdbus functions into GLib GMainLoop mechanism (but dropping support for GType).

Dropping support for the GType and stub code will mean that you would have to implement the introspection support manually and be dependent on possible API changes in libdbus in the future.

Another possible solution would be to "fake" the completion of client method calls, so that the RPC method will complete immediately, but the server will continue (using GIOChannel integration) processing the request until it will really complete. The problem in this solution is that it is very difficult to know which client actually issued the original method call and how to communicate the final result (or errors) of the method call to the client once it does complete. One possible model here would be using signals to broadcast the end result of the method call, so that the client will get the result at some point (assuming the client is still attached to the message bus). Needless to say, this is quite inelegant and difficult to implement correctly, especially since sending signals will cause unnecessary load by waking up all the clients on the bus (even if they're not interested in that particular signal).

In short, there is no simple solution that works properly when GLib/D-Bus wrappers are used.

## 8.5 Debugging

The simplest way to debug your servers will be intelligent usage of print out of events in the code sections that are relevant. Tracing everything that goes on rarely makes sense, but having a reliable and working infrastructure (in code level) will help. One such mechanism is utilising various built in "magic" that gcc and cpp provide. In the server example, a macro called `dbg` is used, which will expand to `g_print` when the server is built as non-daemonizing version. If the server will become a daemon, the macro expands to "nothing", meaning that no code will be generated to format the parameters or to even access them. You will want to extend this idea to support multiple levels of debugging, and possibly use different "subsystem" identifiers so that you can switch one subsystem on or off depending on what you're debugging.

The `dbg` macro utilises the `__func__` symbol, which expands to the function name where the macro will be expanded, which is quite useful so that you don't have to add the function name explicitly:

```
/* 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
```

Listing 8.3: Adding debug message support to a server based on a define (glib-dbus-sync/server.c)

Using the macro is then quite simple, as it will look and act like a regular `printf`-formatting function (`g_print` included):

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

Listing 8.4: Utilising the dbg macro (glib-dbus-sync/server.c)

The only small difference being that you don't have to explicitly add the trailing newline (`\n`) into each call, since it will be automatically added.

Assuming `NO_DAEMON` is defined, the macro would expand to the following output when the server would be run:

```
server:value_object_getvalue2: Called (internal value2 is 42.000)
```

For larger projects, you will also want to combine `__file__` so that tracing multi file programs will become easier.

Coupled with proper test cases (which would be using the client code, and possibly also `dbus-send` in D-Bus related programs), this is a very powerful technique and often much easier than single stepping through your code with a debugger (`gdb`) or setting and evaluation breakpoints. You will also be interested in using Valgrind to help you detect memory leaks (and some other errors). More information on these topics and examples are available in maemo.org documentation.

# Appendix A

# Source code for the libdbus example

## A.1 libdbus-example/dbus-example.c

```
/**
 * A simple D-Bus RPC sending example.
 *
 * 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.
 *
 * The program will:
 * 1) Connect to the D-Bus Session bus
 * 2) Send one RPC method call (one that will cause a Note dialog to
 *    be popped up for user).
 * 3) Quit
 */

/**
 * Uses the low-level libdbus which shouldn't be used directly.
 * As the D-Bus API reference puts it "If you use this low-level API
 * directly, you're signing up for some pain".
 */

#include <dbus/dbus.h> /* Pull in all of D-Bus headers. */
#include <stdio.h>     /* printf, fprintf, stderr */
#include <stdlib.h>    /* EXIT_FAILURE, EXIT_SUCCESS */
#include <assert.h>    /* assert */

/* Symbolic defines for the D-Bus well-known name, interface, object
   path and method name that we're going to use. */

#define SYSNOTE_NAME  "org.freedesktop.Notifications"
#define SYSNOTE_OPATH "/org/freedesktop/Notifications"
#define SYSNOTE_IFACE "org.freedesktop.Notifications"
#define SYSNOTE_NOTE  "SystemNoteDialog"

/**
 * Utility to terminate if given DBusError is set.
 * Will print out the message and error before terminating.
```

```
 *
 * If error is not set, will do nothing.
 *
 * NOTE: In real applications you should spend a moment or two
 *       thinking about the exit-paths from your application and
 *       whether you need to close/unreference all resources that you
 *       have allocated. In this program, we rely on the kernel to do
 *       all necessary cleanup (closing sockets, releasing memory),
 *       but in real life you need to be more careful.
 *
 *       One possible solution model to this is implemented in
 *       "flashlight", a simple program that is presented later.
 */
static void terminateOnError(const char* msg,
                             const DBusError* error) {

  assert(msg != NULL);
  assert(error != NULL);

  if (dbus_error_is_set(error)) {
    fprintf(stderr, msg);
    fprintf(stderr, "DBusError.name: %s\n", error->name);
    fprintf(stderr, "DBusError.message: %s\n", error->message);
    /* If the program wouldn't exit because of the error, freeing the
       DBusError needs to be done (with dbus_error_free(error)).
       NOTE:
         dbus_error_free(error) would only free the error if it was
         set, so it is safe to use even when you're unsure. */
    exit(EXIT_FAILURE);
  }
}

/**
 * The main program that demonstrates a simple "fire & forget" RPC
 * method invocation.
 */
int main(int argc, char** argv) {

  /* Structure representing the connection to a bus. */
  DBusConnection* bus = NULL;
  /* The method call message. */
  DBusMessage* msg = NULL;

  /* D-Bus will report problems and exceptions using the DBusError
     structure. We'll allocate one in stack (so that we don't need to
     free it explicitly. */
  DBusError error;

  /* Message to display. */
  const char* dispMsg = "Hello World!";
  /* Text to use for the acknowledgement button. "" means default. */
  const char* buttonText = "";
  /* Type of icon to use in the dialog (1 = OSSO_GN_ERROR). We could
     have just used the symbolic version here as well, but that would
     have required pulling the LibOSSO-header files. And this example
     must work without LibOSSO, so this is why a number is used. */
  int iconType = 1;

  /* Clean the error state. */
  dbus_error_init(&error);

  printf("Connecting to Session D-Bus\n");
```

```
bus = dbus_bus_get(DBUS_BUS_SESSION, &error);
terminateOnError("Failed to open Session bus\n", &error);
assert(bus != NULL);

/* Normally one would just do the RPC call immediately without
   checking for name existence first. However, sometimes it's useful
   to check whether a specific name even exists on a platform on
   which you're planning to use D-Bus.

   In our case it acts as a reminder to run this program using the
   run-standalone.sh script when running in the SDK.

   The existence check is not necessary if the recipient is
   startable/activateable by D-Bus. In that case, if the recipient
   is not already running, the D-Bus daemon will start the
   recipient (a process that has been registered for that
   well-known name) and then passes the message to it. This
   automatic starting mechanism will avoid the race condition
   discussed below and also makes sure that only one instance of
   the service is running at any given time. */
printf("Checking whether the target name exists ("
       SYSNOTE_NAME ")\n");
if (!dbus_bus_name_has_owner(bus, SYSNOTE_NAME, &error)) {
  fprintf(stderr, "Name has no owner on the bus!\n");
  return EXIT_FAILURE;
}
terminateOnError("Failed to check for name ownership\n", &error);
/* Someone on the Session bus owns the name. So we can proceed in
   relative safety. There is a chance of a race. If the name owner
   decides to drop out from the bus just after we check that it is
   owned, our RPC call (below) will fail anyway. */

/* Construct a DBusMessage that represents a method call.
   Parameters will be added later. The internal type of the message
   will be DBUS_MESSAGE_TYPE_METHOD_CALL. */
printf("Creating a message object\n");
msg = dbus_message_new_method_call(SYSNOTE_NAME, /* destination */
                                   SYSNOTE_OPATH,  /* obj. path */
                                   SYSNOTE_IFACE,  /* interface */
                                   SYSNOTE_NOTE); /* method str */
if (msg == NULL) {
  fprintf(stderr, "Ran out of memory when creating a message\n");
  exit(EXIT_FAILURE);
}

/* Set the "no-reply-wanted" flag into the message. This also means
   that we cannot reliably know whether the message was delivered or
   not, but since we don't have reply message handling here, it
   doesn't matter. The "no-reply" is a potential flag for the remote
   end so that they know that they don't need to respond to us.

   If the no-reply flag is set, the D-Bus daemon makes sure that the
   possible reply is discarded and not sent to us. */
dbus_message_set_no_reply(msg, TRUE);

/* Add the arguments to the message. For the Note dialog, we need
   three arguments:
     arg0: (STRING) "message to display, in UTF-8"
     arg1: (UINT32) type of dialog to display. We will use 1.
                    (libosso.h/OSSO_GN_ERROR).
     arg2: (STRING) "text to use for the ack button". "" means
                    default text (OK in our case).
```

```
    When listing the arguments, the type needs to be specified first
    (by using the libdbus constants) and then a pointer to the
    argument content needs to be given.

    NOTE: It is always a pointer to the argument value, not the value
          itself!

    We terminate the list with DBUS_TYPE_INVALID. */
  printf("Appending arguments to the message\n");
  if (!dbus_message_append_args(msg,
                                DBUS_TYPE_STRING, &dispMsg,
                                DBUS_TYPE_UINT32, &iconType,
                                DBUS_TYPE_STRING, &buttonText,
                                DBUS_TYPE_INVALID)) {
    fprintf(stderr, "Ran out of memory while constructing args\n");
    exit(EXIT_FAILURE);
  }

  printf("Adding message to client's send-queue\n");
  /* We could also get a serial number (dbus_uint32_t) for the message
     so that we could correlate responses to sent messages later. In
     our case there won't be a response anyway, so we don't care about
     the serial, so we pass a NULL as the last parameter. */
  if (!dbus_connection_send(bus, msg, NULL)) {
    fprintf(stderr, "Ran out of memory while queueing message\n");
    exit(EXIT_FAILURE);
  }

  printf("Waiting for send-queue to be sent out\n");
  dbus_connection_flush(bus);

  printf("Queue is now empty\n");

  /* Now we could in theory wait for exceptions on the bus, but since
     this is only a simple D-Bus example, we'll skip that. */

  printf("Cleaning up\n");

  /* Free up the allocated message. Most D-Bus objects have internal
     reference count and sharing possibility, so _unref() functions
     are quite common. */
  dbus_message_unref(msg);
  msg = NULL;

  /* Free-up the connection. libdbus attempts to share existing
     connections for the same client, so instead of closing down a
     connection object, it is unreferenced. The D-Bus library will
     keep an internal reference to each shared connection, to
     prevent accidental closing of shared connections before the
     library is finalized. */
  dbus_connection_unref(bus);
  bus = NULL;

  printf("Quitting (success)\n");

  return EXIT_SUCCESS;
}
```

Listing A.1: libdbus-example/dbus-example.c

## A.2 libdbus-example/Makefile

```
#
# Simple Makefile for the libdbus example that will send the "Display
# Note dialog" RPC message. You need to run the example in the SDK or
# a compatible device.
#

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

PKG_CFLAGS  := $(shell pkg-config --cflags $(pkg_packages))
PKG_LDFLAGS := $(shell pkg-config --libs $(pkg_packages))
# Additional flags for the compiler:
#    -g : Add debugging symbols
# -Wall : Enable most gcc warnings
ADD_CFLAGS := -g -Wall

# Combine user supplied, additional, and pkg-config flags
CFLAGS  := $(PKG_CFLAGS) $(ADD_CFLAGS) $(CFLAGS)
LDFLAGS := $(PKG_LDFLAGS) $(LDFLAGS)

targets = dbus-example

.PHONY: all clean
all: $(targets)

dbus-example: dbus-example.c
  $(CC) $(CFLAGS) $< -o $@ $(LDFLAGS)

clean:
  $(RM) $(targets)
```

Listing A.2: libdbus-example/Makefile

# Appendix B

# Source code for the LibOSSO RPC examples

## B.1 libosso-example-sync/libosso-rpc-sync.c

```
/**
 * A program that will issue a "system_note_dialog" RPC method call.
 *
 * 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 function does the same as dbus-example.c, but this time
 * utilizing LibOSSO _rpc-functions (synchronously).
 *
 * It would be unfair to compare this program directly against the
 * dbus-example.c, since this program contains additional
 * functionality. In general using the LibOSSO _rpc functions will
 * lead to more compact code, but there are cases when you need to go
 * beyond what LibOSSO provides (and using the GLib/D-Bus interface is
 * the proper way to go in that case).
 *
 * Please run with run-standalone.sh (LibOSSO initialization will fail
 * otherwise).
 */

#include <glib.h>

#include <libosso.h>

#include <stdlib.h> /* EXIT_SUCCESS */

/* Symbolic defines for the D-Bus well-known name, interface, object
   path and method name that we're going to use. Same as before.
   These are defined in osso-internal.h but we cannot use it here. */

#define SYSNOTE_NAME  "org.freedesktop.Notifications"
#define SYSNOTE_OPATH "/org/freedesktop/Notifications"
#define SYSNOTE_IFACE "org.freedesktop.Notifications"
#define SYSNOTE_NOTE  "SystemNoteDialog"
```

```
/**
 * Utility to return a pointer to a statically allocated string giving
 * the textural representation of LibOSSO errors. Has no internal
 * state (safe to use from threads).
 *
 * LibOSSO does not come with a function for this, so we define one
 * ourselves.
 */
static const gchar* ossoErrorStr(osso_return_t errCode) {

  switch (errCode) {
    case OSSO_OK:
      return "No error (OSSO_OK)";
    case OSSO_ERROR:
      return "Some kind of error occurred (OSSO_ERROR)";
    case OSSO_INVALID:
      return "At least one parameter is invalid (OSSO_INVALID)";
    case OSSO_RPC_ERROR:
      return "Osso RPC method returned an error (OSSO_RPC_ERROR)";
    case OSSO_ERROR_NAME:
      return "(undocumented error) (OSSO_ERROR_NAME)";
    case OSSO_ERROR_NO_STATE:
      return "No state file found to read (OSSO_ERROR_NO_STATE)";
    case OSSO_ERROR_STATE_SIZE:
      return "Size of state file unexpected (OSSO_ERROR_STATE_SIZE)";
    default:
      return "Unknown/Undefined";
  }
}

/**
 * Utility to print out the type and content of given osso_rpc_t.
 * It also demonstrates the types available when using LibOSSO for
 * the RPC. Most simple types are available, but arrays are not
 * (unfortunately).
 */
static void printOssoValue(const osso_rpc_t* val) {

  g_assert(val != NULL);

  switch (val->type) {
    case DBUS_TYPE_BOOLEAN:
      g_print("boolean:%s", (val->value.b == TRUE)?"TRUE":"FALSE");
      break;
    case DBUS_TYPE_DOUBLE:
      g_print("double:%.3f", val->value.d);
      break;
    case DBUS_TYPE_INT32:
      g_print("int32:%d", val->value.i);
      break;
    case DBUS_TYPE_UINT32:
      g_print("uint32:%u", val->value.u);
      break;
    case DBUS_TYPE_STRING:
      g_print("string:'%s'", val->value.s);
      break;
    case DBUS_TYPE_INVALID:
      g_print("invalid/void");
      break;
    default:
      g_print("unknown(type=%d)", val->type);
      break;
```

```
  }
}

/**
 * Do the RPC call.
 *
 * Note that this function will block until the method call either
 * succeeds, or fails. If the method call would take a long time to
 * run, this would block the GUI of the program (which we don't have).
 *
 * Needs the LibOSSO state to do the launch.
 */
static void runRPC(osso_context_t* ctx) {

  /* Message to display. */
  const char* dispMsg = "Hello SystemNote!\nVia LibOSSO/sync.";
  /* Icon type to use. */
  gint iconType = OSSO_GN_ERROR;
  /* Button label text to use, "" means leaving the defaults. */
  const char* labelText = "";

  /* Will hold the result from the RPC invocation function. */
  osso_return_t result;
  /* Will hold the result of the method call (or error). */
  osso_rpc_t methodResult = {};

  g_print("runRPC called\n");

  g_assert(ctx != NULL);

  /* Compared to the libdbus functions, LibOSSO provides conveniently
     a function that will do the dispatch and also allows us to pass
     the arguments all with one call.

     The arguments for the "SystemNoteDialog" are the same as in
     dbus-example.c (since it is the same service). You might also
     notice that even if LibOSSO provides some convenience, it does
     not completely isolate us from libdbus. We still supply the
     argument types using D-Bus constants.

     NOTE Do not pass the argument values by pointers as with libdbus,
          instead pass them by value (as below). */
  result = osso_rpc_run(ctx,
                        SYSNOTE_NAME,      /* well-known name */
                        SYSNOTE_OPATH,         /* object path */
                        SYSNOTE_IFACE,           /* interface */
                        SYSNOTE_NOTE,          /* method name */
                        &methodResult, /* method return value */
                        /* The arguments for the RPC. The types
                           are unchanged, but instead of passing
                           them via pointers, they're passed by
                           "value" instead. */
                        DBUS_TYPE_STRING, dispMsg,
                        DBUS_TYPE_UINT32, iconType,
                        DBUS_TYPE_STRING, labelText,
                        DBUS_TYPE_INVALID);
  /* Check whether launching the RPC succeeded. */
  if (result != OSSO_OK) {
    g_error("Error launching the RPC (%s)\n",
            ossoErrorStr(result));
    /* We also terminate right away since there's nothing to do. */
  }
```

```
  g_print("RPC launched successfully\n");

  /* Now decode the return data from the method call.
     NOTE: If there is an error during RPC delivery, the return value
           will be a string. It is not possible to differentiate that
           condition from an RPC call that returns a string.

     If a method returns "void", the type-field in the methodResult
     will be set to DBUS_TYPE_INVALID. This is not an error. */
  g_print("Method returns: ");
  printOssoValue(&methodResult);
  g_print("\n");

  g_print("runRPC ending\n");
}

int main(int argc, char** argv) {

  /* The LibOSSO context that we need to do RPC. */
  osso_context_t* ossoContext = NULL;

  g_print("Initializing LibOSSO\n");
  /* The program name for registration is communicated from the
     Makefile via a -D preprocessor directive. Since it doesn't
     contain any dots in it, a prefix of "com.nokia." will be added
     to it internally within osso_initialize(). */
  ossoContext = osso_initialize(ProgName, "1.0", FALSE, NULL);
  if (ossoContext == NULL) {
    g_error("Failed to initialize LibOSSO\n");
  }

  g_print("Invoking the method call\n");
  runRPC(ossoContext);

  g_print("Shutting down LibOSSO\n");
  /* Deinitialize LibOSSO. The function doesn't return status code so
     we cannot know whether it succeeded or failed. We assume that it
     always succeeds. */
  osso_deinitialize(ossoContext);
  ossoContext = NULL;

  g_print("Quitting\n");
  return EXIT_SUCCESS;
}
```

Listing B.1: libosso-example-sync/libosso-rpc-sync.c

## B.2 libosso-example-sync/Makefile

```
#
# Simple Makefile to build the LibOSSO/Async-RPC example
#

# define a list of pkg-config packages we want to use
pkg_packages := glib-2.0 libosso

PKG_CFLAGS  := $(shell pkg-config --cflags $(pkg_packages))
PKG_LDFLAGS := $(shell pkg-config --libs $(pkg_packages))
# add warnings and debugging info CFLAGS-variable
```

```
ADD_CFLAGS += -g -Wall

# Combine user supplied, additional and pkg-config flags
CFLAGS  := $(PKG_CFLAGS) $(ADD_CFLAGS) $(CFLAGS)
LDFLAGS := $(PKG_LDFLAGS) $(LDFLAGS)

targets = libosso-rpc-sync

all: $(targets)

libosso-rpc-sync: libosso-rpc-sync.c
  $(CC) $(CFLAGS) -DProgName=\"LibOSSOExample\" \
    $< -o $@ $(LDFLAGS)

.PHONY: clean all
clean:
  $(RM) $(targets)
```

Listing B.2: libosso-example-sync/Makefile

## B.3 libosso-example-async/libosso-rpc-async.c

```
/**
 * A program that will issue a "system_note_dialog" RPC method call.
 *
 * 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 function does the same as dbus-example.c, but this time
 * utilizing LibOSSO _rpc-functions and demonstrating how to integrate
 * asynchronous calls with LibOSSO.
 *
 * It would be unfair to compare this program directly against the
 * dbus-example.c, since this program contains additional
 * functionality. In general using the LibOSSO _rpc functions will
 * lead to more compact code, but there are cases when you need to go
 * beyond what LibOSSO provides (and using the GLib/D-Bus interface is
 * the proper way to go in that case).
 *
 * Please run with run-standalone.sh (LibOSSO initialization will fail
 * otherwise).
 */

#include <glib.h>
#include <libosso.h>
#include <stdlib.h> /* EXIT_SUCCESS */

/* Symbolic defines for the D-Bus well-known name, interface, object
   path and method name that we're going to use. Same as before.
   These are defined in osso-internal.h but we cannot use it here. */

#define SYSNOTE_NAME  "org.freedesktop.Notifications"
#define SYSNOTE_OPATH "/org/freedesktop/Notifications"
#define SYSNOTE_IFACE "org.freedesktop.Notifications"
#define SYSNOTE_NOTE  "SystemNoteDialog"

/**
```

```
 * Small application state so that we can pass both LibOSSO context
 * and the mainloop around to the callbacks.
 */
typedef struct {
  /* A mainloop object that will "drive" our example. */
  GMainLoop* mainloop;
  /* The LibOSSO context which we use to do RPC. */
  osso_context_t* ossoContext;
} ApplicationState;

/**
 * Utility to return a pointer to a statically allocated string giving
 * the textural representation of LibOSSO errors. Has no internal
 * state (safe to use from threads).
 *
 * LibOSSO does not come with a function for this, so we define one
 * ourselves.
 */
static const gchar* ossoErrorStr(osso_return_t errCode) {

  switch (errCode) {
    case OSSO_OK:
      return "No error (OSSO_OK)";
    case OSSO_ERROR:
      return "Some kind of error occurred (OSSO_ERROR)";
    case OSSO_INVALID:
      return "At least one parameter is invalid (OSSO_INVALID)";
    case OSSO_RPC_ERROR:
      return "Osso RPC method returned an error (OSSO_RPC_ERROR)";
    case OSSO_ERROR_NAME:
      return "(undocumented error) (OSSO_ERROR_NAME)";
    case OSSO_ERROR_NO_STATE:
      return "No state file found to read (OSSO_ERROR_NO_STATE)";
    case OSSO_ERROR_STATE_SIZE:
      return "Size of state file unexpected (OSSO_ERROR_STATE_SIZE)";
    default:
      return "Unknown/Undefined";
  }
}

/**
 * Utility to print out the type and content of given osso_rpc_t.
 * It also demonstrates the types available when using LibOSSO for
 * the RPC. Most simple types are available, but arrays are not
 * (unfortunately).
 */
static void printOssoValue(const osso_rpc_t* val) {

  g_assert(val != NULL);

  switch (val->type) {
    case DBUS_TYPE_BOOLEAN:
      g_print("boolean:%s", (val->value.b == TRUE)?"TRUE":"FALSE");
      break;
    case DBUS_TYPE_DOUBLE:
      g_print("double:%.3f", val->value.d);
      break;
    case DBUS_TYPE_INT32:
      g_print("int32:%d", val->value.i);
      break;
    case DBUS_TYPE_UINT32:
      g_print("uint32:%u", val->value.u);
```

```
      break;
    case DBUS_TYPE_STRING:
      g_print("string:'%s'", val->value.s);
      break;
    case DBUS_TYPE_INVALID:
      g_print("invalid/void");
      break;
    default:
      g_print("unknown(type=%d)", val->type);
      break;
  }
}

/**
 * Will be called from LibOSSO when the RPC return data is available.
 * Will print out the result, and return. Note that it must not free
 * the value, since it does not own it.
 *
 * The prototype (for reference) must be osso_rpc_async_f().
 *
 * The parameters for the callback are the D-Bus interface and method
 * names (note that object path and well-known name are NOT
 * communicated). The idea is that you can then reuse the same
 * callback to process completions from multiple simple RPC calls.
 */
static void rpcCompletedCallback(const gchar* interface,
                                 const gchar* method,
                                 osso_rpc_t* retVal,
                                 gpointer userData) {

  ApplicationState* state = (ApplicationState*)userData;

  g_print("rpcCompletedCallback called\n");

  g_assert(interface != NULL);
  g_assert(method != NULL);
  g_assert(retVal != NULL);
  g_assert(state != NULL);

  g_print(" interface: %s\n", interface);
  g_print(" method: %s\n", method);
  /* NOTE If there is an error in the RPC delivery, the return value
          will be a string. This is unfortunate if your RPC call is
          supposed to return a string as well, since it is not
          possible to differentiate between the two cases.

          If a method returns "void", the type-field in the retVal
          will be set to DBUS_TYPE_INVALID (it's not an error). */
  g_print(" result: ");
  printOssoValue(retVal);
  g_print("\n");

  /* Tell the main loop to terminate. */
  g_main_loop_quit(state->mainloop);

  g_print("rpcCompletedCallback done\n");
}

/**
 * We launch the RPC call from within a timer callback in order to
 * make sure that a mainloop object will be running when the RPC will
 * return (to avoid a nasty race condition).
```

```
 *
 * So, in essence this is a one-shot timer callback.
 *
 * In order to launch the RPC, it will need to get a valid LibOSSO
 * context (which is carried via the userData/application state
 * parameter).
 */
static gboolean launchRPC(gpointer userData) {

  ApplicationState* state = (ApplicationState*)userData;
  /* Message to display. */
  const char* dispMsg = "Hello SystemNote!\nVia LibOSSO/async.";
  /* Icon type to use. */
  gint iconType = OSSO_GN_ERROR;
  /* Button label text to use. */
  const char* labelText = "Execute!";

  /* Will hold the result from the RPC launch call. */
  osso_return_t result;

  g_print("launchRPC called\n");

  g_assert(state != NULL);

  /* Compared to the libdbus functions, LibOSSO provides conveniently
     a function that will do the dispatch, registration of callbacks
     and allow argument supplying in one call.

     The arguments for the "system_note_dialog" are the same as in
     dbus-example.c (since it is the same service). You might also
     notice that even if LibOSSO provides some convenience, it does
     not completely isolate us from libdbus. We still supply the
     argument types using D-Bus constants.

     NOTE Do not pass the argument values by pointers as with libdbus,
          instead pass them by value (as below). */

  /* The only difference compared to the synchronous version is the
     addition of the callback function parameter, and the user-data
     parameter for data that will be passed to the callback. */
  result = osso_rpc_async_run(state->ossoContext,
                              SYSNOTE_NAME,     /* well-known name */
                              SYSNOTE_OPATH,        /* object path */
                              SYSNOTE_IFACE,          /* interface */
                              SYSNOTE_NOTE,         /* method name */
                              rpcCompletedCallback,    /* async cb */
                              state,           /* user-data for cb */
                              /* The arguments for the RPC. */
                              DBUS_TYPE_STRING, dispMsg,
                              DBUS_TYPE_UINT32, iconType,
                              DBUS_TYPE_STRING, labelText,
                              DBUS_TYPE_INVALID);
  /* Check whether launching the RPC succeeded (we don't know the
     result from the RPC itself). */
  if (result != OSSO_OK) {
    g_error("Error launching the RPC (%s)\n",
            ossoErrorStr(result));
    /* We also terminate right away since there's nothing to do. */
  }
  g_print("RPC launched successfully\n");

  g_print("launchRPC ending\n");
```

```
  /* We only want to be called once, so ask the caller to remove this
     callback from the timer launch list by returning FALSE. */
  return FALSE;
}

int main(int argc, char** argv) {

  /* Keep the application state in main's stack. */
  ApplicationState state = {};
  /* Keeps the results from LibOSSO functions for decoding. */
  osso_return_t result;
  /* Default timeout for RPC calls in LibOSSO. */
  gint rpcTimeout;

  g_print("Initializing LibOSSO\n");
  state.ossoContext = osso_initialize(ProgName, "1.0", FALSE, NULL);
  if (state.ossoContext == NULL) {
    g_error("Failed to initialize LibOSSO\n");
  }

  /* Print out the default timeout value (which we don't change, but
     could, with osso_rpc_set_timeout()). */
  result = osso_rpc_get_timeout(state.ossoContext, &rpcTimeout);
  if (result != OSSO_OK) {
    g_error("Error getting default RPC timeout (%s)\n",
            ossoErrorStr(result));
  }
  /* Interestingly the timeout seems to be -1, but is something else
     (by default). -1 probably then means that "no timeout has been
     set". */
  g_print("Default RPC timeout is %d (units)\n", rpcTimeout);

  g_print("Creating a mainloop object\n");
  /* Create a GMainLoop with default context and initial condition of
     not running (FALSE). */
  state.mainloop = g_main_loop_new(NULL, FALSE);
  if (state.mainloop == NULL) {
    g_error("Failed to create a GMainLoop\n");
  }

  g_print("Adding timeout to launch the RPC in one second\n");
  /* This could be replaced by g_idle_add(cb, &state), in order to
     guarantee that the RPC would be launched only after the mainloop
     has started. We opt for a timeout here (for no particular
     reason). */
  g_timeout_add(1000, (GSourceFunc)launchRPC, &state);

  g_print("Starting mainloop processing\n");
  g_main_loop_run(state.mainloop);

  g_print("Out of mainloop, shutting down LibOSSO\n");
  /* Deinitialize LibOSSO. */
  osso_deinitialize(state.ossoContext);
  state.ossoContext = NULL;

  /* Free GMainLoop as well. */
  g_main_loop_unref(state.mainloop);
  state.mainloop = NULL;

  g_print("Quitting\n");
  return EXIT_SUCCESS;
```

```
}
```

Listing B.3: libosso-example-async/libosso-rpc-async.c

## B.4 libosso-example-async/Makefile

```
#
# Simple Makefile to build the LibOSSO/Async-RPC example
#

# Define a list of pkg-config packages we want to use
pkg_packages := glib-2.0 libosso

PKG_CFLAGS  := $(shell pkg-config --cflags $(pkg_packages))
PKG_LDFLAGS := $(shell pkg-config --libs $(pkg_packages))
# Add warnings and debugging info
ADD_CFLAGS += -g -Wall

# Combine user supplied, additional, and pkg-config flags
CFLAGS  := $(PKG_CFLAGS) $(ADD_CFLAGS) $(CFLAGS)
LDFLAGS := $(PKG_LDFLAGS) $(LDFLAGS)

targets = libosso-rpc-async

all: $(targets)

libosso-rpc-async: libosso-rpc-async.c
  $(CC) $(CFLAGS) -DProgName=\"LibOSSOExample\" \
    $< -o $@ $(LDFLAGS)

.PHONY: clean all
clean:
  $(RM) $(targets)
```

Listing B.4: libosso-example-async/Makefile

# Appendix C

# Source code for flashlight

## C.1 libosso-flashlight/flashlight.c

```
/**
 * A simple LibOSSO non-GUI program.
 *
 * 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.
 *
 * Demonstrates how to utilize LibOSSO for trivial requests as well as
 * how to receive state change notifications from the system.
 *
 * The program will use LibOSSO to enable the backlight on the screen
 * of a device, and then periodically extend the backlight timeout
 * mechanism in the device, in order to keep the backlight on. As
 * such, it can be used a "flashlight"-replacement, but really is just
 * a demonstration program about using LibOSSO.
 *
 * The program will either refuse to run, or stop, when the device is
 * placed into "Flight"-mode. This is achieved by utilizing the
 * LibOSSO device state change callbacks.
 *
 * Other demonstrations include displaying Note dialogs to user from
 * a non-GUI program, utilizing the infoprint component and LibOSSO
 * error and state decoding.
 *
 * If you really want a "flashlight" application, you should consider
 * writing a small Hildon application that will display a fully white
 * GtkCanvas and switch it to run in full screen mode. You might also
 * want to maximize the backlight for the duration of the program. The
 * brightness is controlled via a GConf key (an integer from 1 to 9)
 * /system/osso/dsm/display/display_brightness and changing that key
 * will affect brightness automagically (assuming the backlight
 * dimming timer has not expired, but that is what this program stops
 * from happening).
 *
 * NOTE: Keeping the screen and backlight on without a good reason is
 *       not a good idea as this drains the battery on the device.
 *
 * NOTE: When running this on the SDK, you will need to use the
 *       run-standalone.sh script, otherwise LibOSSO initialization
```

```
 *       will fail. The RPC method calls will all succeed in the
 *       SDK (assuming you have AF running), but since there is no
 *       real "screen blank/backlight dimming" functionality in the
 *       SDK, you won't see any changes. Also, it is not possible to
 *       simulate state change events in the SDK (so you will have to
 *       terminate the program using Ctrl+c).
 */

#include <glib.h>
#include <libosso.h>
#include <stdlib.h> /* EXIT_SUCCESS */
#include <string.h> /* memset */

/* Application state.

   Contains the necessary state to control the application lifetime
   and use LibOSSO. */
typedef struct {
  /* The GMainLoop that will run our code. */
  GMainLoop* mainloop;
  /* LibOSSO context that is necessary to use LibOSSO functions. */
  osso_context_t* ossoContext;
  /* Flag to tell the timer that it should stop running. Also utilized
     to tell the main program that the device is already in Flight-
     mode and the program shouldn't continue startup. */
  gboolean running;
} ApplicationState;

/**
 * Utility to return a pointer to a statically allocated string giving
 * the textural representation of LibOSSO errors. Has no internal
 * state (safe to use from threads).
 *
 * LibOSSO does not come with a function for this, so we define one
 * ourselves.
 */
static const gchar* ossoErrorStr(osso_return_t errCode) {

  switch (errCode) {
    case OSSO_OK:
      return "No error (OSSO_OK)";
    case OSSO_ERROR:
      return "Some kind of error occurred (OSSO_ERROR)";
    case OSSO_INVALID:
      return "At least one parameter is invalid (OSSO_INVALID)";
    case OSSO_RPC_ERROR:
      return "Osso RPC method returned an error (OSSO_RPC_ERROR)";
    case OSSO_ERROR_NAME:
      return "(undocumented error) (OSSO_ERROR_NAME)";
    case OSSO_ERROR_NO_STATE:
      return "No state file found to read (OSSO_ERROR_NO_STATE)";
    case OSSO_ERROR_STATE_SIZE:
      return "Size of state file unexpected (OSSO_ERROR_STATE_SIZE)";
    default:
      return "Unknown/Undefined";
  }
}

/**
 * Utility to ask the device to pause the screen blanking timeout.
 * Does not return success/status.
 */
```

```
static void delayDisplayBlanking(ApplicationState* state) {

  osso_return_t result;

  g_assert(state != NULL);

  result = osso_display_blanking_pause(state->ossoContext);
  if (result != OSSO_OK) {
    g_printerr(PROGNAME ":delayDisplayBlanking. Failed (%s)\n",
               ossoErrorStr(result));
    /* But continue anyway. */
  } else {
    g_print(PROGNAME ":delayDisplayBlanking RPC succeeded\n");
  }
}

/**
 * Timer callback that will be called within 45 seconds after
 * installing the callback and will ask the platform to defer any
 * display blanking for another 60 seconds.
 *
 * This will also prevent the device from going into suspend
 * (according to LibOSSO documentation).
 *
 * It will continue doing this until the program is terminated.
 *
 * NOTE: Normally timers shouldn't be abused like this since they
 *       bring the CPU out from power saving state. Because the screen
 *       backlight dimming might be activated again after 60 seconds
 *       of receiving the "pause" message, we need to keep sending the
 *       "pause" messages more often than every 60 seconds. 45 seconds
 *       seems like a safe choice.
 */
static gboolean delayBlankingCallback(gpointer data) {
  ApplicationState* state = (ApplicationState*)data;

  g_print(PROGNAME ":delayBlankingCallback Starting\n");

  g_assert(state != NULL);
  /* If we're not supposed to be running anymore, return immediately
     and ask caller to remove the timeout. */
  if (state->running == FALSE) {
    g_print(PROGNAME ":delayBlankingCallback Removing\n");
    return FALSE;
  }

  /* Delay the blanking for a while still (60 seconds). */
  delayDisplayBlanking(state);

  g_print(PROGNAME ":delayBlankingCallback Done\n");

  /* We want the same callback to be invoked from the timer
     launch again, so we return TRUE. */
  return TRUE;
}

/* Small macro to return "YES" or "no" based on given parameter.
   Used in printDeviceState below. YES is in capital letters in order
   for it to "stand out" in the program output (since it's much
   rarer). */
#define BOOLSTR(p) ((p)?"YES":"no")
```

```
/**
 * Utility to decode the hwstate structure and print it out in human
 * readable format. Mainly useful for debugging on the device.
 *
 * The mode constants unfortunately are not documented in LibOSSO.
 */
static void printDeviceState(osso_hw_state_t* hwState) {

  gchar* modeStr = "Unknown";

  g_assert(hwState != NULL);

  switch(hwState->sig_device_mode_ind) {
    case OSSO_DEVMODE_NORMAL:
      /* Non-flight-mode. */
      modeStr = "Normal";
      break;
    case OSSO_DEVMODE_FLIGHT:
      /* Power button -> "Offline mode". */
      modeStr = "Flight";
      break;
    case OSSO_DEVMODE_OFFLINE:
      /* Unknown. Even if all connections are severed, this mode will
         not be triggered. */
      modeStr = "Offline";
      break;
    case OSSO_DEVMODE_INVALID:
      /* Unknown. */
      modeStr = "Invalid(?)";
      break;
    default:
      /* Leave at "Unknown". */
      break;
  }
  g_print(
    "Mode: %s, Shutdown: %s, Save: %s, MemLow: %s, RedAct: %s\n",
    modeStr,
    /* Set to TRUE if the device is shutting down. */
    BOOLSTR(hwState->shutdown_ind),
    /* Set to TRUE if our program has registered for autosave and
       now is the moment to save user data. */
    BOOLSTR(hwState->save_unsaved_data_ind),
    /* Set to TRUE when device is running low on memory. If possible,
       memory should be freed by our program. */
    BOOLSTR(hwState->memory_low_ind),
    /* Set to TRUE when system wants us to be less active. */
    BOOLSTR(hwState->system_inactivity_ind));
}

/**
 * This callback will be called by LibOSSO whenever the device state
 * changes. It will also be called right after registration to inform
 * the current device state.
 *
 * The device state structure contains flags telling about conditions
 * that might affect applications, as well as the mode of the device
 * (for example telling whether the device is in "in-flight"-mode).
 */
static void deviceStateChanged(osso_hw_state_t* hwState,
                               gpointer data) {
  ApplicationState* state = (ApplicationState*)data;
```

```
  g_print(PROGNAME ":deviceStateChanged Starting\n");

  printDeviceState(hwState);

  /* If device is in/going into "flight-mode" (called "Offline" on
     some devices), we stop our operation automatically. Obviously
     this makes flashlight useless (as an application) if someone gets
     stuck in a dark cargo bay of a plane with snakes.. But we still
     need a way to shut down the application and react to device
     changes, and this is the easiest state to test with.

     Note that since offline mode will terminate network connections,
     you will need to test this on the device itself, not over ssh. */
  if (hwState->sig_device_mode_ind == OSSO_DEVMODE_FLIGHT) {
    g_print(PROGNAME ":deviceStateChanged In/going into offline.\n");

    /* Terminate the mainloop.
       NOTE: Since this callback is executed immediately on
             registration, the mainloop object is not yet "running",
             hence calling quit on it will be ineffective! _quit only
             works when the mainloop is running. */
    g_main_loop_quit(state->mainloop);
    /* We also set the running to correct state to fix the above
       problem. */
    state->running = FALSE;
  }
}

/**
 * Utility to display an "exiting" message to user using infoprint.
 */
static void displayExitMessage(ApplicationState* state,
                               const gchar* msg) {

  osso_return_t result;

  g_assert(state != NULL);

  g_print(PROGNAME ":displayExitMessage Displaying exit message\n");
  result = osso_system_note_infoprint(state->ossoContext, msg, NULL);
  if (result != OSSO_OK) {
    /* This is rather harsh, since we terminate the whole program if
       the infoprint RPC fails. It is used to display messages at
       program exit anyway, so this isn't a critical issue. */
    g_error(PROGNAME ": Error doing infoprint (%s)\n",
            ossoErrorStr(result));
  }
}

/**
 * Utility to setup the application state.
 *
 * 1) Initialize LibOSSO (this will connect to D-Bus)
 * 2) Create a mainloop object
 * 3) Register the device state change callback
 *    The callback will be called once immediately on registration.
 *    The callback will reset the state->running to FALSE when the
 *    program needs to terminate so we'll know whether the program
 *    should run at all. If not, display an error dialog.
 *    (This is the case if the device will be in "Flight"-mode when
 *    the program starts.)
 * 4) Register the timer callback (which will keep the screen from
```

```
 *    blanking).
 * 5) Un-blank the screen.
 * 6) Display a dialog to the user (on the background) warning about
 *    battery drain.
 * 7) Send the first "delay backlight dimming" command.
 *
 * Returns TRUE when everything went ok, FALSE when caller should call
 * releaseAppState and terminate. The code below will print out the
 * errors if necessary.
 */
static gboolean setupAppState(ApplicationState* state) {

  osso_return_t result;

  g_assert(state != NULL);

  g_print(PROGNAME ":setupAppState starting\n");

  /* Zero out the state. Has the benefit of setting all pointers to
     NULLs and all gbooleans to FALSE. This is useful when we'll need
     to determine what to release later. */
  memset(state, 0, sizeof(ApplicationState));

  g_print(PROGNAME ":setupAppState Initializing LibOSSO\n");
  /* Parameters for osso_initialize:
     gchar* : Application "name". This is not the name that will be
              visible to user, but rather the name that other programs
              can use to communicate with this program (eventually
              over D-Bus). Note that if your name does _not_ include
              dot, 'com.nokia.' will be prepended to the name
              automatically.
     gchar* : Application version.
     gboolean : Unused / no effect.
     GMainContext* : Context under which LibOSSO will integrate into.
              Leave to NULL in order to use the default context (which
              will be true for programs that use one GMainLoop, from
              the default context). */
  state->ossoContext = osso_initialize(ProgName, "1.0", FALSE, NULL);
  if (state->ossoContext == NULL) {
    g_printerr(PROGNAME ": Failed to initialize LibOSSO\n");
    return FALSE;
  }

  g_print(PROGNAME ":setupAppState Creating a GMainLoop object\n");
  /* Create a new GMainLoop object, with default context (NULL) and
     initial "running"-state set to FALSE. */
  state->mainloop = g_main_loop_new(NULL, FALSE);
  if (state->mainloop == NULL) {
    g_printerr(PROGNAME ": Failed to create a GMainLoop\n");
    return FALSE;
  }

  g_print(PROGNAME
          ":setupAddState Adding hw-state change callback.\n");
  /* The callback will be called immediately with the state, so we
     need to know whether we're in offline mode to start with. If so,
     the callback will set the running-member to FALSE (and we'll
     check it below). */
  state->running = TRUE;
  /* In order to receive information about device state and changes
     in it, we register our callback here.
```

```
    Parameters for the osso_hw_set_event_cb():
      osso_context_t* :  LibOSSO context object to use.
      osso_hw_state_t* : Pointer to a device state type that we're
                         interested in. NULL for "all states".
      osso_hw_cb_f* :    Function to call on state changes.
      gpointer :         User-data passed to callback. */
result = osso_hw_set_event_cb(state->ossoContext,
                              NULL, /* We're interested in all. */
                              deviceStateChanged,
                              state);
if (result != OSSO_OK) {
  g_printerr(PROGNAME
             ":setupAppState Failed to get state change CB\n");
  /* Since we cannot reliably know when to terminate later on
     without state information, we will refuse to run because of the
     error. */
  return FALSE;
}
/* We're in "Flight" mode? */
if (state->running == FALSE) {
  g_print(PROGNAME ":setupAppState In offline, not continuing.\n");
  displayExitMessage(state, ProgName " not available in Offline mode"
      );
  return FALSE;
}

g_print(PROGNAME ":setupAppState Adding blanking delay timer.\n");
if (g_timeout_add(45000,
                  (GSourceFunc)delayBlankingCallback,
                  state) == 0) {
  /* If g_timeout_add returns 0, it signifies an invalid event
     source id. This means that adding the timer failed. */
  g_printerr(PROGNAME ": Failed to create a new timer callback\n");
  return FALSE;
}

/* Un-blank the display (will always succeed in the SDK). */
g_print(PROGNAME ":setupAppState Unblanking the display\n");
result = osso_display_state_on(state->ossoContext);
if (result != OSSO_OK) {
  g_printerr(PROGNAME ": Failed in osso_display_state_on (%s)\n",
             ossoErrorStr(result));
  /* If the RPC call fails, odds are that nothing else will work
     either, so we decide to quit instead. */
  return FALSE;
}

/* Display a "Note"-dialog with a WARNING icon.
   The Dialog is MODAL, so user cannot do anything with the stylus
   until the Ok is selected, or the Back-key is pressed. */

/* In most cases, you should avoid displaying modal dialogs,
   especially when running without a normal GUI (like this program
   is). Instead, you might want to use osso_system_note_infoprint()
   instead.

   In our case, we really really want the user to realize that
   the program will cause extra battery usage. Hence the warning,
   and a modal note dialog. If you run this on a device, you might
   also hear a special alerting sound. */

/* Other icons available:
```

```
     OSSO_GN_NOTICE: For general notices.
    OSSO_GN_WARNING: For warning messages.
      OSSO_GN_ERROR: For error messages.
       OSSO_GN_WAIT: For messages about "delaying" for something (an
                     hourglass icon is displayed).
                  5: Animated progress indicator. */

  /* The text must be UTF-8 formatted, and may contain newlines, but
     other markup is not supported (and you should limit the amount of
     text that you will put into the dialog).

     If we'd be interested in the return data from the RPC method,
     we could pass a pointer to a osso_rpc_t where the result would
     be then stored.

     Unfortunately for us, the Note dialog returns "void" and returns
     it immediately. This means that the dialog will be waiting for
     the user to press "Ok", but we won't get notified about that.
     In fact, we'll continue running immediately after asking for the
     dialog. This is common to most of the convenience wrappers in
     LibOSSO.

     So, we pass NULL to tell LibOSSO not to bother with the return
     value. */

  g_print(PROGNAME ":setupAppState Displaying Note dialog\n");
  result = osso_system_note_dialog(state->ossoContext,
                                   /* UTF-8 text into the dialog */
                                   "Started " ProgName ".\n"
                        "Please remember to stop it when you're done, "
                                "in order to conserve battery power.",
                                   /* Icon to use */
                                   OSSO_GN_WARNING,
                                   /* We're not interested in the RPC
                                      return value. */
                                   NULL);
  if (result != OSSO_OK) {
    g_error(PROGNAME ": Error displaying Note dialog (%s)\n",
            ossoErrorStr(result));
  }
  g_print(PROGNAME ":setupAppState Requested for the dialog\n");

  /* Then delay the blanking timeout so that our timer callback has a
     chance to run before that. */
  delayDisplayBlanking(state);

  g_print(PROGNAME ":setupAppState Completed\n");

  /* State set up. */
  return TRUE;
}

/**
 * Release all resources allocated by setupAppState in reverse order.
 */
static void releaseAppState(ApplicationState* state) {

  g_print(PROGNAME ":releaseAppState starting\n");

  g_assert(state != NULL);

  /* First set the running state to FALSE so that if the timer will
```

```
    (for some reason) be launched, it will remove itself from the
    timer call list. This shouldn't be possible since we are running
    only with one thread. */
  state->running = FALSE;

  /* Normally we would also release the timer, but since the only way
    to do that is from the timer callback itself, there's not much we
    can do about it here. */

  /* Remove the device state change callback. It is possible that we
    run this even if the callback was never installed, but it is not
    harmful. */
  if (state->ossoContext != NULL) {
    osso_hw_unset_event_cb(state->ossoContext, NULL);
  }

  /* Release the mainloop object. */
  if (state->mainloop != NULL) {
    g_print(PROGNAME ":releaseAppState Releasing mainloop object.\n");
    g_main_loop_unref(state->mainloop);
    state->mainloop = NULL;
  }

  /* Lastly, free up the LibOSSO context. */
  if (state->ossoContext != NULL) {
    g_print(PROGNAME ":releaseAppState De-init LibOSSO.\n");
    osso_deinitialize(state->ossoContext);
    state->ossoContext = NULL;
  }
  /* All resources released. */
}

/**
 * Main program:
 *
 * 1) Setup application state
 * 2) Start mainloop
 * 3) Release application state & terminate
 */
int main(int argc, char** argv) {

  /* We'll keep one application state in our program and allocate
    space for it from the stack. */
  ApplicationState state = {};

  g_print(PROGNAME ":main Starting\n");
  /* Attempt to setup the application state and if something goes
    wrong, do cleanup here and exit. */
  if (setupAppState(&state) == FALSE) {
    g_print(PROGNAME ":main Setup failed, doing cleanup\n");
    releaseAppState(&state);
    g_print(PROGNAME ":main Terminating with failure\n");
    return EXIT_FAILURE;
  }

  g_print(PROGNAME ":main Starting mainloop processing\n");
  g_main_loop_run(state.mainloop);
  g_print(PROGNAME ":main Out of main loop (shutting down)\n");
  /* We come here when the application state has the running flag set
    to FALSE and the device state changed callback has decided to
    terminate the program. Display a message to the user about
    termination next. */
```

```
  displayExitMessage(&state, ProgName " exiting");

  /* Release the state and exit with success. */
  releaseAppState(&state);

  g_print(PROGNAME ":main Quitting\n");
  return EXIT_SUCCESS;
}
```

Listing C.1: libosso-flashlight/flashlight.c

## C.2 libosso-flashlight/Makefile

```
#
# Makefile for the simple LibOSSO flashlight program
#

# define a list of pkg-config packages we want to use
pkg_packages := glib-2.0 libosso

PKG_CFLAGS  := $(shell pkg-config --cflags $(pkg_packages))
PKG_LDFLAGS := $(shell pkg-config --libs $(pkg_packages))
# Add warnings and debugging info
ADD_CFLAGS := -g -Wall

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

targets = flashlight

all: $(targets)

# For simple one-file programs, combining both compiling and linking
# makes sense.
#
# PROGNAME = name of program to prefix to all printout in program
# ProgName = name of program to use for LibOSSO registration and user
#            visible dialogs/infoprints.
flashlight: flashlight.c
  $(CC) $(CFLAGS) -DPROGNAME=\"$@\" -DProgName=\"FlashLight\" \
    $< -o $@ $(LDFLAGS)

.PHONY: clean all
clean:
  $(RM) $(targets)
```

Listing C.2: libosso-flashlight/Makefile

# Appendix D

# Source code for the GLib D-Bus synchronous example

## D.1 glib-dbus-sync/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 D.1: glib-dbus-sync/common-defs.h

## D.2 glib-dbus-sync/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 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>

  </interface>
</node>
```

Listing D.2: glib-dbus-sync/value-dbus-interface.xml

## D.3 glib-dbus-sync/server.c

```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 are 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"

/* This defines the per-instance state.

   Each GObject must start with the 'parent' definition so that common
   operations that all GObjects support can be called on it. */
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;

/* Per class state.

   For the first Value implementation we only have the bare minimum,
   that is, the common implementation for any GObject class. */
typedef struct {
  /* The parent class state. */
  GObjectClass parent;
} ValueObjectClass;
```

```
/* Forward declaration of the function that will return the GType of
   the Value implementation. Not used in this program since we only
   need to push this over the D-Bus. */
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 are 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))

/* Utility macro to define the value_object GType structure. */
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 does not
   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
 *
 * 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) {

  dbg("Called");

  g_assert(klass != NULL);

  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 */
}

/**
 * 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 will 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);

  /* Change the value. */
  obj->value1 = valueIn;
```

```
  /* Return success to GLib/D-Bus wrappers. In this case we do not 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);

  obj->value2 = valueIn;

  return TRUE;
}

/**
 * Function that gets executed on "getvalue1".
 */
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
     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("Could not 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("Could not 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 are interested in 1 (we are now the primary owner of the name)
   or 4 (we were already the owner of the name, however in this
   application it would not 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 are 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 will 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 will not 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
     does not 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 maemo 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 not 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 D.3: glib-dbus-sync/server.c

## D.4 glib-dbus-sync/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).
 *
 */

#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 */
```

```c
#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);
  }
}

/**
 * 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.
 *
 * 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) Start a timer that will launch timerCallback once per second.
 * 6) 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);
  }

  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 D.4: glib-dbus-sync/client.c

## D.5 glib-dbus-sync/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) $(CFLAGS) -DPROGNAME=\"$(basename $@)\" -c $< -o $@

client.o: client.c common-defs.h value-client-stub.h
  $(CC) $(CFLAGS) -DPROGNAME=\"$(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 would 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 additinal 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 D.5: glib-dbus-sync/Makefile

# Appendix E

# Source code for the GLib D-Bus signal example

## E.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 E.1: glib-dbus-signals/common-defs.h

## E.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>
```

```
    <!-- 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 E.2: glib-dbus-signals/value-dbus-interface.xml

## E.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. */
    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 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
     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 E.3: glib-dbus-signals/server.c

## E.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 E.4: glib-dbus-signals/client.c

## E.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) $(CFLAGS) -DPROGNAME=\"$(basename $@)\" -c $< -o $@

client.o: client.c common-defs.h value-client-stub.h
  $(CC) $(CFLAGS) -DPROGNAME=\"$(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 additinal 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 E.5: glib-dbus-signals/Makefile

# Appendix F

# Source code for the GLib D-Bus asynchronous examples

## F.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 F.1: glib-dbus-async/common-defs.h

## F.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 F.2: glib-dbus-async/value-dbus-interface.xml

## F.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 F.3: glib-dbus-async/server.c

## F.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 F.4: glib-dbus-async/client-stubs.c

## F.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 F.5: glib-dbus-async/client-glib.c

## F.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 F.6: glib-dbus-async/Makefile

# Appendix G

# Source code for the asynchronous GConf example

## G.1 gconf-listener/gconf-key-watch.c

```
/**
 * A simple CLI program that uses the GObject:ified GConf library to
 * wait for GConf key changes (from within GMainLoop). Will run
 * until terminated manually.
 *
 * 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.
 *
 * Depends on glib and gconf libraries (when building).
 *
 * One thing to note about this example as that even if it is used to
 * track GConf key/value changes, it might not be the optimal in all
 * cases. When you are following many keys, using a single
 * notification callback will force you to use strcmp() to find out
 * exactly which key changed and act accordingly.
 *
 * The strcmp-model is used in this example, as we only have two
 * keys to track. The other option would be to have separate
 * callbacks for _each_ key that is tracked, and that way you could
 * forego the strcmp within each callback.
 *
 * The drawback in the latter approach is that it might increase the
 * size of the generated code (since each callback would have to be
 * a separate function). The latter approach however is the one
 * that is more commonly used in real programs, since they have
 * multiple key/values to track.
 *
 * Please use gconftool-2 to change the values so that you
 * will see some output from this program (see below for examples).
 *
 * Also, you will need to create the entries before-hand, as the
 * program is very simplistic, and will abort if the values are not
 * present when it's run. A proper application would populate the
 * missing keys with sane defaults. A demonstration of this can be
 * found in the GConf example in "maemo Application Development".
```

```
 *
 * To set the values from CLI, you can use the following commands:
 * $ gconftool-2 -s --type string /apps/Maemo/platdev_ex/connection \
 *               btcomm0
 * $ gconftool-2 -s --type string \
 *               /apps/Maemo/platdev_ex/connectionparams 9600,8,N,1
 *
 * You can use the provided Makefile with target 'primekeys' to
 * achieve the same effect as above (there are also other targets
 * useful for testing in the Makefile).
 *
 * You can also use the same commands to change the values (while
 * running this program).
 *
 * To check the existing values from CLI:
 * $ gconftool-2 -R /apps/Maemo/platdev_ex
 */

#include <glib.h>
#include <gconf/gconf-client.h>
#include <string.h> /* strcmp */

/* As per maemo Coding Style and Guidelines document, we use the
   /apps/Maemo/ -prefix.
   NOTE: There is no central registry (as of this moment) that you
         could check that your application name doesn't collide with
         other application names, so caution is advised! */
#define SERVICE_GCONF_ROOT "/apps/Maemo/platdev_ex"

/* We define the names of the keys symbolically so that we may change
   them later if necessary, and so that the GConf "root directory" for
   our application will be automatically prefixed to the paths. */
#define SERVICE_KEY_CONNECTION \
        SERVICE_GCONF_ROOT "/connection"
#define SERVICE_KEY_CONNECTIONPARAMS \
        SERVICE_GCONF_ROOT "/connectionparams"

/**
 * Callback called when a key in watched directory changes.
 * Prototype for the callback must be compatible with
 * GConfClientNotifyFunc (for ref).
 *
 * It will find out which key changed (using strcmp, since the same
 * callback is used to track both keys) and the display the new value
 * of the key.
 *
 * The changed key/value pair will be communicated in the entry
 * parameter. userData will be NULL (can be set on notify_add [in
 * main]). Normally the application state would be carried within the
 * userData parameter, so that this callback could then modify the
 * view based on the change. Since this program does not have a state,
 * there is little that we can do within the function (it will abort
 * the program on errors though).
 */
static void keyChangeCallback(GConfClient* client,
                              guint        cnxn_id,
                              GConfEntry*  entry,
                              gpointer     userData) {

  /* This will hold the pointer to the value. */
  const GConfValue* value = NULL;
  /* This will hold a pointer to the name of the key that changed. */
```

```
  const gchar* keyname = NULL;
  /* This will hold a dynamically allocated human-readable
     representation of the changed value. */
  gchar* strValue = NULL;

  g_print(PROGNAME ": keyChangeCallback invoked.\n");

  /* Get a pointer to the key (this is not a copy). */
  keyname = gconf_entry_get_key(entry);

  /* It will be quite fatal if after change we cannot retrieve even
     the name for the gconf entry, so we error out here. */
  if (keyname == NULL) {
    g_error(PROGNAME ": Couldn't get the key name!\n");
    /* Application terminates. */
  }

  /* Get a pointer to the value from changed entry. */
  value = gconf_entry_get_value(entry);

  /* If we get a NULL as the value, it means that the value either has
     not been set, or is at default. As a precaution we assume that
     this cannot ever happen, and will abort if it does.
     NOTE: A real program should be more resilient in this case, but
           the problem is: what is the correct action in this case?
           This is not always simple to decide.
     NOTE: You can trip this assert with 'make primekeys', since that
           will first remove all the keys (which causes the CB to
           be invoked, and abort here). */
  g_assert(value != NULL);

  /* Check that it looks like a valid type for the value. */
  if (!GCONF_VALUE_TYPE_VALID(value->type)) {
    g_error(PROGNAME ": Invalid type for gconfvalue!\n");
  }

  /* Create a human readable representation of the value. Since this
     will be a new string created just for us, we'll need to be
     careful and free it later. */
  strValue = gconf_value_to_string(value);

  /* Print out a message (depending on which of the tracked keys
     change. */
  if (strcmp(keyname, SERVICE_KEY_CONNECTION) == 0) {
    g_print(PROGNAME ": Connection type setting changed: [%s]\n",
            strValue);
  } else if (strcmp(keyname, SERVICE_KEY_CONNECTIONPARAMS) == 0) {
    g_print(PROGNAME ": Connection params setting changed: [%s]\n",
            strValue);
  } else {
    g_print(PROGNAME ":Unknown key: %s (value: [%s])\n", keyname,
            strValue);
  }

  /* Free the string representation of the value. */
  g_free(strValue);

  g_print(PROGNAME ": keyChangeCallback done.\n");
}

/**
 * Utility to retrieve a string key and display it.
```

```
 * (Just as a small refresher on the API.)
 */
static void dispStringKey(GConfClient* client,
                          const gchar* keyname) {

  /* This will hold the string value of the key. It will be
     dynamically allocated for us, so we need to release it ourselves
     when done (before returning). */
  gchar* valueStr = NULL;

  /* We're not interested in the errors themselves (the last
     parameter), but the function will return NULL if there is one,
     so we just end in that case. */
  valueStr = gconf_client_get_string(client, keyname, NULL);

  if (valueStr == NULL) {
    g_error(PROGNAME ": No string value for %s. Quitting\n", keyname);
    /* Application terminates. */
  }

  g_print(PROGNAME ": Value for key '%s' is set to '%s'\n",
          keyname, valueStr);

  /* Normally one would want to use the value for something beyond
     just displaying it, but since this code doesn't, we release the
     allocated value string. */
  g_free(valueStr);
}

/**
 * The main application.
 *
 * 1) Initialize the GType/GObject system.
 * 2) Attach to GConf system (gconfd)
 * 3) Retrieve the two keys (just to print them out)
 * 4) Register the application configuration directory for possible
 *    notifications.
 * 5) Register the callback to be used on changes.
 * 6) Start mainloop (and stay there forever).
 *
 * Note that this program does not terminate by itself!
 *
 * You could add g_main_loop_quit to do that from the callback,
 * but since the callback does not have access to the mainloop object,
 * you'd need to either:
 * a) Pass the mainloop object as user-specified data (trivial, but
 *    somewhat wrong by design).
 * b) Create an application state, and put the reference to the
 *    mainloop object there. Then pass the application state as the
 *    user specified data to the callback (better design, but out of
 *    scope for this simple program).
 */
int main (int argc, char** argv) {
  /* Will hold reference to the GConfClient object. */
  GConfClient* client = NULL;
  /* Initialize this to NULL so that we'll know whether an error
     occurred or not (and we don't have an existing GError object
     anyway at this point). */
  GError* error = NULL;
  /* This will hold a reference to the mainloop object. */
  GMainLoop* mainloop = NULL;
```

```
g_print(PROGNAME ":main Starting.\n");

/* Must be called to initialize GType system. The API reference for
   gconf_client_get_default() insists.
   NOTE: Using gconf_init() is deprecated! */
g_type_init();

/* Create a new mainloop structure that we'll use. Use default
   context (NULL) and set the 'running' flag to FALSE. */
mainloop = g_main_loop_new(NULL, FALSE);
if (mainloop == NULL) {
  g_error(PROGNAME ": Failed to create mainloop!\n");
}

/* Create a new GConfClient object using the default settings. */
client = gconf_client_get_default();
if (client == NULL) {
  g_error(PROGNAME ": Failed to create GConfClient!\n");
}

g_print(PROGNAME ":main GType and GConfClient initialized.\n");

/* Display the starting values for the two keys. */
dispStringKey(client, SERVICE_KEY_CONNECTION);
dispStringKey(client, SERVICE_KEY_CONNECTIONPARAMS);

/**
 * Register directory to watch for changes. This will then tell
 * GConf to watch for changes in this namespace, and cause the
 * "value-changed"-signal to be emitted. We won't be using that
 * mechanism, but will opt to a more modern (and potentially more
 * scalable solution). The directory needs to be added to the
 * watch list in either case.
 *
 * When adding directories, you can sometimes optimize your program
 * performance by asking GConfClient to preload some (or all) keys
 * under a specific directory. This is done via the preload_type
 * parameter (we use GCONF_CLIENT_PRELOAD_NONE below). Since our
 * program will only listen for changes, we don't want to use extra
 * memory to keep the keys cached.
 *
 * Parameters:
 * - client: GConf-client object
 * - SERVICEPATH: the name of the GConf namespace to follow
 * - GCONF_CLIENT_PRELOAD_NONE: do not preload any of contents
 * - error: where to store the pointer to allocated GError on
 *          errors.
 */
gconf_client_add_dir(client,
                     SERVICE_GCONF_ROOT,
                     GCONF_CLIENT_PRELOAD_NONE,
                     &error);

if (error != NULL) {
  g_error(PROGNAME ": Failed to add a watch to GCClient: %s\n",
          error->message);
  /* Normally we'd also release the allocated GError, but since
     this program will terminate on g_error, we won't do that.
     Hence the next line is commented. */
  /* g_error_free(error); */

  /* When you want to release the error if it has been allocated,
```

```
      or just continue if not, use g_clear_error(&error); */
  }

  g_print(PROGNAME ":main Added " SERVICE_GCONF_ROOT ".\n");

  /* Register our interest (in the form of a callback function) for
     any changes under the namespace that we just added.

     Parameters:
     - client: GConfClient object.
     - SERVICEPATH: namespace under which we can get notified for
                    changes.
     - gconf_notify_func: callback that will be called on changes.
     - NULL: user-data pointer (not used here).
     - NULL: function to call on user-data when notify is removed or
             GConfClient destroyed. NULL for none (since we don't
             have user-data anyway).
     - error: return location for an allocated GError.

     Returns:
     guint: an ID for this notification so that we could remove it
            later with gconf_client_notify_remove(). We're not going
            to use it so we don't store it anywhere. */
  gconf_client_notify_add(client, SERVICE_GCONF_ROOT,
                          keyChangeCallback, NULL, NULL, &error);
  if (error != NULL) {
    g_error(PROGNAME ": Failed to add register the callback: %s\n",
            error->message);
    /* Program terminates. */
  }

  g_print(PROGNAME ":main CB registered & starting main loop\n");

  /* Start the main loop. */
  g_main_loop_run(mainloop);
  /* Main loop finished.
     NOTE: Without modifications, this program will never actually
           reach this point, since the main loop is never
           terminated. */

  g_print(PROGNAME ":main Out of main loop (shouldn't happen)\n");

  /* Release the mainloop object. */
  g_main_loop_unref(mainloop);

  /* Release the gconfclient since we're done now. This would also run
     the freeing function that we could have passed on notify_add-
     registration above, but since we didn't have a need for one,
     no free'ers will be run here (for us at least). */
  g_object_unref(client);

  g_print(PROGNAME ":main Ending\n");

  return 0;
}
```

Listing G.1: gconf-listener/gconf-key-watch.c

## G.2 gconf-listener/Makefile

```
# This is the Makefile for the GConf key watch example.
# The default target will build the application.
#
# There are also two special targets:
# primekeys : populate the GConf parameters with default values
# clearkeys : remote all of the application GConf keys (recursively)
# dumpkeys  : list all keys for this application (recursively)
#
# In order to use the targets, the GConf daemon will need to be
# running in your system (af-sb-init.sh). The daemon is running
# at all times on the device.

# Define a variable for this so that the GConf root may be changed
gconf_root := /apps/Maemo/platdev_ex

# pkg-config packages that we'll need
pkg_packages := glib-2.0 gconf-2.0

PKG_CFLAGS  := $(shell pkg-config --cflags $(pkg_packages))
PKG_LDFLAGS := $(shell pkg-config --libs $(pkg_packages))
# Additional flags for the compiler:
#    -g : Add debugging symbols
# -Wall : Enable most gcc warnings
ADD_CFLAGS := -g -Wall

# Combine user supplied, additional, and pkg-config flags
CFLAGS  := $(PKG_CFLAGS) $(ADD_CFLAGS) $(CFLAGS)
LDFLAGS := $(PKG_LDFLAGS) $(LDFLAGS)

# Default targets
targets := gconf-key-watch

.PHONY: all phony primekeys clearkeys dumpkeys
all: $(targets)

# We define a define (PROGNAME) so that we can (or rename the program
# later if necessary).
gconf-key-watch: gconf-key-watch.c
  $(CC) $(CFLAGS) -DPROGNAME=\"$@\" $^ -o $@ $(LDFLAGS)

# This will setup the keys into default values.
# It will first do a clear to remove any existing keys.
primekeys: clearkeys
  gconftool-2 --set --type string \
              $(gconf_root)/connection btcomm0
  gconftool-2 --set --type string \
              $(gconf_root)/connectionparams 9600,8,N,1

# Remove all application keys
clearkeys:
  @gconftool-2 --recursive-unset $(gconf_root)

# Dump all application keys
dumpkeys:
  @echo Keys under $(gconf_root):
  @gconftool-2 --recursive-list $(gconf_root)

clean:
  $(RM) $(targets)
```

Listing G.2: gconf-listener/Makefile