Maemo Diablo D-Bus, The Message Bus System

Training Material

February 9, 2009

Contents

Chapter 1

D-Bus, The Message Bus System

1.1 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.

1.2 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.

1.3 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 1.1: 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.

1.4 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").

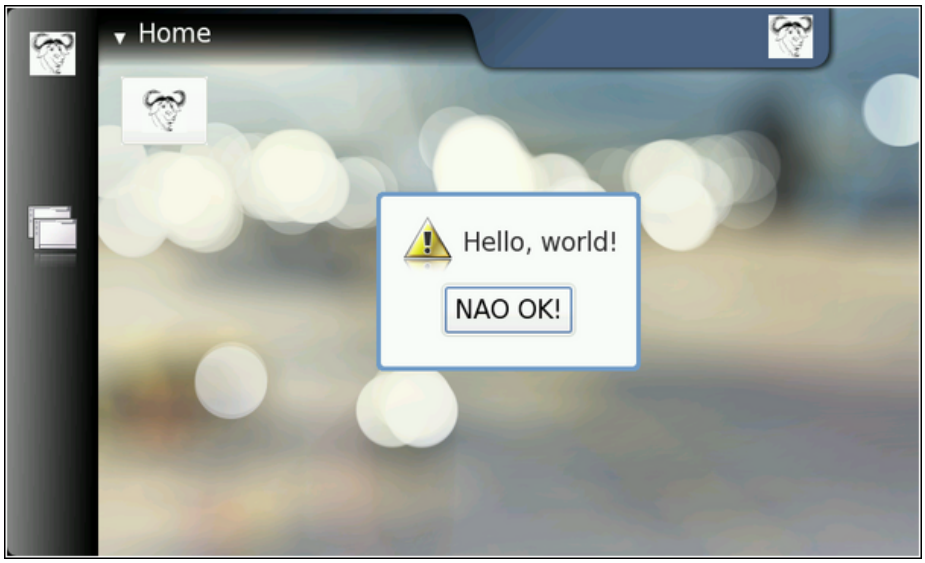
Arguments are specified by giving the argument type and its contents separated with a colon as follows:

```
[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).

1.5 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 1.2: 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 1.3: 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 1.4: 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 1.5: 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 1.6: 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 1.7: 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 1.8: 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 1.9: 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 1.10: 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)
```

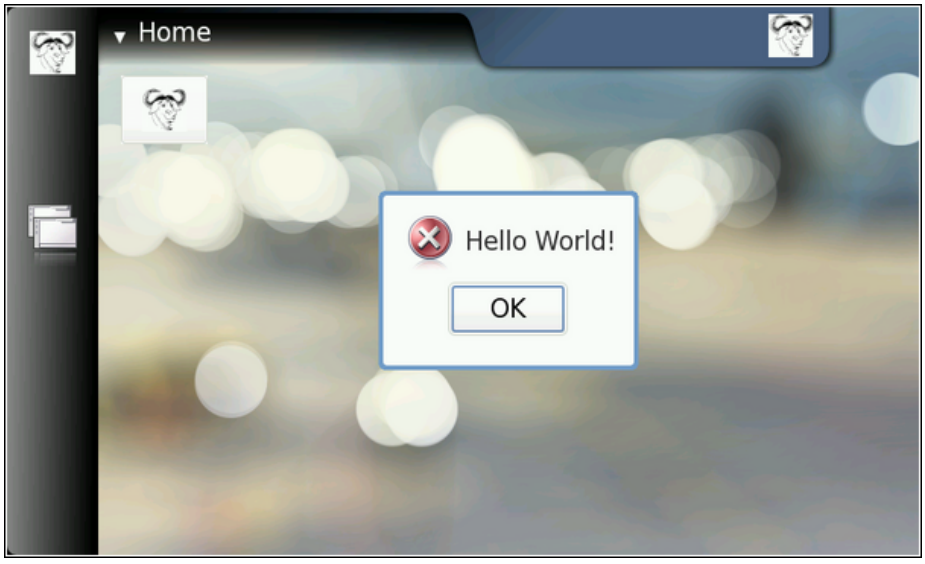
Running the example, but without the proper environmental variables

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 1.11: 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.