Maemo Diablo Integration with the Application Framework

Training Material

February 9, 2009

Contents

Chapter 1

Integration with the Application Framework

1.1 Integrating into AF

This chapter covers the bare minimum in order to get the application integrated into the Task navigator and D-Bus.

To have your GUI application appear in the menu of Task navigator and be controllable by the TN, you will need to create two files. One in order for our application to be activateable via the D-Bus, the other integrating into the menu-structure.

In order for your application to survive the application killer on the device, you'll also need to use a function from `libosso`.

1.2 The desktop file

```
[Desktop Entry]
Encoding=UTF-8
Version=1.0
Type=Application
Name=HelloWorld X
Exec=@prefix@/bin/hhwX
X-Osso-Service=org.maemo.hhwX
Icon=qgn_list_gene_default_app
```

Listing 1.1: Contents of autotoolized-hildon-helloworld/hhwX.desktop.in

Above is the simplest desktop file one must make in order for the TN to be able to launch your application via D-Bus. Programs designed for maemo are normally started via the D-Bus activation mechanism, which explains the service name(`org.maemo.hwwX`) (D-Bus documentation refers to this name as the "well-known name").

You might notice that the file is an unprocessed version of the final one, and uses the `prefix`-configure variable. The correct value for the `prefix`-variable is `/usr`, otherwise the application and the files will be installed in the wrong place and won't be found by the TN.

The desktop file specifies the text to use in the menus, the icon for the menu entry and also the D-Bus service name, so that the application may be activated properly.

There needs to be one desktop file for each GUI application, and it needs to be copied into **/usr/share/applications/hildon**-directory. The name of the file should reflect the name of the application (`hwwX` means hildon_helloworld-10). The desktop file however is only one half of the equation.

1.3 The service file

So that the D-Bus daemon may activate the application on demand and make sure that only one instance of the application will ever be running, the application will need to have a service file installed.

A service file for our simple program is given below:

```
[D-BUS Service]
Name=org.maemo.hhwX
Exec=@prefix@/bin/hhwX
```

Listing 1.2: Contents of autotoolized-hildon-helloworld/org.maemo.hhwX.service.in

The service name needs to match the name in the desktop file as well as the name that we'll use in LibOSSO registration (below). The file name should also follow the service name if possible (although technically is not required to do so). The service file needs to be placed in **/usr/share/dbus-1/services** in order for the D-Bus daemon to find it. Placing the file in the directory is enough for the daemon to notice that it now has a new service that it can start.

The `Exec` member will need to point to the absolute path on the target where the D-Bus daemon will find the executable to start when the service will be required. If the executable in question doesn't register for the `Name` well-known name here, it will eventually be killed by the application killer. Using a `prefix` for the `Exec` is also recommended so that application install paths may be modified later using `./configure` (if necessary).

1.4 Application support

We need to register the application as a D-Bus service in order for it not to be automatically killed by the system. This is done easiest by using `osso_initialize()`. The first parameter needs to be the D-Bus service name (it must match the name that we use in the service file). The software version seems to be unused at the moment, but the idea is that one could have multiple different versions of the same program running, and they wouldn't collide in the D-Bus-namespace.

The other two parameters for normal GUI applications are always `TRUE` and `NULL`. See the `LibOSSO` documentation for their explanation, but normal GUI applications should use these two.

Below is the tenth version of the Hello World, which highlights the changes necessary in order to integrate into the D-Bus:

```
/**
 * hhwX.c (hildon_helloworld-10)
 *
 * 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.
 *
 * Add LibOSSO support for the application.
 *
 * Look for lines with "NEW" or "MODIFIED" in them.
 */

#include <stdlib.h>
#include <hildon/hildon-program.h>
#include <hildon/hildon-color-button.h>
#include <hildon/hildon-find-toolbar.h>
#include <hildon/hildon-file-chooser-dialog.h>
#include <hildon/hildon-banner.h>
#include <libgnomevfs/gnome-vfs.h>
#include <gconf/gconf-client.h>
/* Pull in the LibOSSO library declarations (NEW). */
#include <libosso.h>

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

/* Build up the D-Bus name for this application (NEW). */
#define PACKAGE_DBUS_NAME "org.maemo." PACKAGE_NAME

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

/**
 * MODIFIED
 *
 * Create a LibOSSO context and shut it down when done with the
 * application.
 */
int main(int argc, char** argv) {

  ApplicationState aState = {};

  GtkWidget* label = NULL;
  GtkWidget* vbox = NULL;
  GtkWidget* mainToolbar = NULL;
  GtkWidget* findToolbar = NULL;

  /* Pointer to the LibOSSO context object/connection (NEW). */
  osso_context_t* ctx = NULL;

  if(!gnome_vfs_init()) {
    g_error("Failed to initialize GnomeVFS-libraries, exiting\n");
  }

  /* Initialize the GTK+ */
  gtk_init(&argc, &argv);

  /* Setup the HildonProgram, HildonWindow and application name. */
  aState.program = HILDON_PROGRAM(hildon_program_get_instance());
  g_set_application_name("Hello Hildon!");
  aState.window = HILDON_WINDOW(hildon_window_new());
  hildon_program_add_window(aState.program,
                            HILDON_WINDOW(aState.window));
```

```
/* Create a LibOSSO context (which will also attach this
   application to the D-Bus (NEW).
   NOTE:
     We use the name and version from the configure.ac.
     The D-Bus name is built by prefixing "org.maemo." to the
     package name. */
g_print("Initializing LibOSSO context (" PACKAGE_DBUS_NAME ", "
        PACKAGE_VERSION ")\n");
ctx = osso_initialize(PACKAGE_DBUS_NAME, PACKAGE_VERSION, TRUE,
                      NULL);
if (ctx == NULL) {
  g_print("Failed to init LibOSSO\n");
  return EXIT_FAILURE;
}
g_print("LibOSSO Init done\n");

label = gtk_label_new("<b>Hello</b> <i>Hildon</i> "
                      "(with LibOSSO!)");
gtk_label_set_line_wrap(GTK_LABEL(label), TRUE);
gtk_label_set_use_markup(GTK_LABEL(label), TRUE);
aState.textLabel = label;

buildMenu(&aState);

vbox = gtk_vbox_new(FALSE, 0);
gtk_container_add(GTK_CONTAINER(aState.window), vbox);
gtk_box_pack_end(GTK_BOX(vbox), label, TRUE, TRUE, 0);

mainToolbar = buildToolbar(&aState);
findToolbar = buildFindToolbar(&aState);

aState.mainToolbar = mainToolbar;
aState.findToolbar = findToolbar;

/* Connect the termination signals. */
g_signal_connect(G_OBJECT(aState.window), "delete-event",
                 G_CALLBACK(cbEventDelete), &aState);
g_signal_connect(G_OBJECT(aState.window), "destroy",
                 G_CALLBACK(cbActionTopDestroy), &aState);

/* Show all widgets that are contained by the Window. */
gtk_widget_show_all(GTK_WIDGET(aState.window));

/* Add the toolbars to the Hildon Window. */
hildon_window_add_toolbar(HILDON_WINDOW(aState.window),
                          GTK_TOOLBAR(mainToolbar));
hildon_window_add_toolbar(HILDON_WINDOW(aState.window),
                          GTK_TOOLBAR(findToolbar));

/* Register a callback to handle key presses. */
g_signal_connect(G_OBJECT(aState.window), "key_press_event",
                 G_CALLBACK(cbKeyPressed), &aState);

g_print("main: calling gtk_main\n");
gtk_main();

g_print("main: returned from gtk_main & de-initing LibOSSO\n");
/* De-initialize LibOSSO (detaches from the D-Bus) (NEW). */
osso_deinitialize(ctx);

g_print("main: exiting with success\n");
```

```
  return EXIT_SUCCESS;
}
```

Listing 1.3: Using LibOSSO for D-Bus registration (autotoolized-hildon-helloworld/hhwX.c)

You will need to add the `libosso` library support using `pkg-config`. This is done by modifying the `configure.ac` file that you should have for your project by now.

1.5 Autotools support for the service and desktop file

So that the service and desktop files will automatically be part of your autotoolized project, you'll need to add them as `_DATA` dependencies and also tell autoconf to run them through the variable substitution machinery. The former is required for the files to be distributed when you will run the `dist` target.

One way of adding the files into automake is used in the hhwX program:

```
# List of the filenames of binaries that this project will produce.
bin_PROGRAMS = hhwX

# For each binary file name, list the source files required for it to
# build. hhwX only consists of one program and that only requires one
# source code file. This is rather atypical.
hhwX_SOURCES = hhwX.c

# In order for the desktop and service to be copied into the correct
# places (and to support prefix-redirection), use the following
# configuration:
dbusdir=$(datadir)/dbus-1/services
dbus_DATA=org.maemo.hhwX.service
desktopdir=$(datadir)/applications/hildon
desktop_DATA=hhwX.desktop
# We described two directories and gave automake a list of files
# which are to be copied into these directories on install. Without
# these directives, the desktop and service files would never be
# installed even if they would be distributed (using EXTRA_DIST).
```

Listing 1.4: Integrating the service and interface files into the project (autotoolized-hildon-helloworld/Makefile.am)

Since the files need to be generated from `.in`-templates, we need to tell autoconf about that as well:

```
# Generate the service and desktop files based on the templates.
AC_OUTPUT(hhwX.desktop org.maemo.hhwX.service)
```

Listing 1.5: Part of configure.ac that will cause the templates to be converted into proper files

1.6 Testing

Before proceeding with Debian package creation, it is useful to test out whether your autotoolized project works:

1. If starting from scratch, prepare the files necessary for `automake` and `autoconf` by running **autogen.sh**.

2. Run `./configure --prefix=/usr` in order to test for necessary tools and to generate the final desktop and service files.

3. Build the software with `make`

4. Test it without installing with `run-standalone.sh ./hhwX`.

5. Assuming that the software started in the previous step, you can now install it (in the SDK) with `fakeroot make install`.

6. Verify that the Extras menu in Task navigator now contains your program, and start it. If the program will not start at this point, it means that either the D-Bus service name is incorrect (different from the name that you give LibOSSO on initialisation), the service file is in a wrong place, or your executable is in the wrong place. Maybe you forgot to use the correct `prefix`.

7. After you're done with the testing, uninstall the files with `fakeroot make uninstall` so that you don't leave unnecessary files around in the SDK.

At this point, you're ready to proceed with creating a Debian package out of your software.