Maemo Diablo Support Libraries
Training Material

February 9, 2009

Contents

Chapter 1

Support Libraries

1.1 Doing File I/O

When developing for maemo, there are two ways of doing file-related I/O operations: either the POSIX interface, which is light-weight and makes the kernel work for you, or using GnomeVFS (GNOME Virtual File System) with GLib. GnomeVFS is implemented via plug-in modules which are loaded on demand depending on what kind of path is used to open a "file". You might remember from the introduction that HTTP was mentioned, but there is a long list of plug-ins that come with GnomeVFS or can be added later on. Things like iterating through an compressed archive is possible as well as accessing such an archive over an ftp connection.

1.2 GnomeVFS

In maemo, GnomeVFS also provides some filename case insensitivity support so that end-users do not have to care about the UNIX filename conventions which are case-sensitive.

The GnomeVFS interface attempts to provide a POSIX-like interface, so that when one would use `open()` with POSIX, one will use `gnome_vfs_open` instead. Instead of `write()`, you have `gnome_vfs_write` and so on (for most functions). The GnomeVFS function names are sometimes a bit more verbose, but otherwise attempt to implement the basic API. Some POSIX functions like `mmap()` are impossible to implement in user-space, but normally this is not a big problem. Also some functions will fail to work properly over network connections and outside your local filesystem since they might not always make sense there.

We will present a simple example of using the GnomeVFS interface functions shortly.

In order to save and load data, you will at least need the following functions:

- `gnome_vfs_init()`: initialises the GnomeVFS library. Needs to be done once at an early stage at program startup.
- `gnome_vfs_shutdown()`: frees up resources inside the library and closes it down.

- `gnome_vfs_open()`: opens the given URI (explained below) and returns a file handle for that if successful.
- `gnome_vfs_get_file_info()`: get information about a file (similar to, but with broader scope than fstat).
- `gnome_vfs_read()`: read data from an opened file.
- `gnome_vfs_write()`: write data into an opened file.

In order to differentiate between different protocols, GnomeVFS uses Uniform Resource Location syntax when accessing resources. For example in `file:///tmp/somefile.txt`, the `file://` is the protocol to use, and the rest is the location within that protocol space for the resource or file to manipulate. Protocols can be stacked inside a single URI and the URI also supports username and password combinations (these are best demonstrated in the GnomeVFS API documentation).

We will be using local files for our simple demonstration.

We will extend our simple application in the following ways:

- Implement the "Open" command by using GnomeVFS with full error checking.
- We'll allocate and free our own memory when loading the contents of the file that the user has selected with `g_malloc0()` and `g_free()`.
- Data loaded through "Open" will replace the text in our GtkLabel that is in the center area of our `HildonWindow`. We will switch the label to support Pango simple text markup (maemo.org) which looks a lot like simple HTML.
- Notification about loading success and failures will be communicated to the user by using a widget called HildonBanner, which will float a small notification dialog (with an optional icon) in the top-right corner for a while without blocking our application.
- Note that saving into a file is not implemented in this code as it is a lab exercise (and it's simpler than opening).
- You can simulate file loading failures by attempting to load an empty file. Since we don't want empty files, our code will turn this into an error as well. If you don't have an empty file available, you can create one easily with the `touch`-command (under **MyDocs** so that the open dialog can find it). You can also attempt to load a file larger than 100 KiB, since the code limits the file size (artificially) and will refuse to load large files.
- The `goto`-statement should normally be avoided. You will have to check your team's coding guidelines whether this is an allowed practise. Note how it's used in this example to cut down the possibility of leaked resources (and typing). Another option for this would be using variable finalises but not many people know how to use them or even that they exist. They are gcc extensions into the C language and you can find more about them by reading gcc info pages (look for variable attributes).

- We're using the simple `GnomeVFS`-functions which are all synchronous, which means that if loading the file will take a long time, our application will remain unresponsive during that time. For small files residing in local storage this is a risk that we choose to take. Synchronous API should not be used when loading files over network since there are more uncertainties in those cases.
- I/O in most cases will be slightly slower than using a controlled approach with POSIX I/O API (controlled meaning that one should know what to use and how). This is a price we're willing to pay in order to easily switch to other protocols possibly later.

Note that since GnomeVFS is a separate library from GLib, you will have to add the flags and library options that it requires. The `pkg-config` package name for the library is `gnome-vfs-2.0`.

```
/**
 * hildon_helloworld-8.c
 *
 * 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.
 *
 * We add file loading support using GnomeVFS. Saving files using
 * GnomeVFS is left as an exercise. We also add a small notification
 * widget (HildonBanner).
 *
 * 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>
/* A small notification window widget (NEW). */
#include <hildon/hildon-banner.h>
/* Pull in the GnomeVFS headers (NEW). */
#include <libgnomevfs/gnome-vfs.h>

/* Declare the two slant styles. */
enum {
  STYLE_SLANT_NORMAL = 0,
  STYLE_SLANT_ITALIC
};

/**
 * The application state.
 */
typedef struct {
  gboolean styleUseUnderline;
  gboolean styleSlant;

  GdkColor currentColor;

  /* Pointer to the label so that we can modify its contents when a
     file is loaded by the user (NEW). */
  GtkWidget* textLabel;
```

```
  gboolean fullScreen;

  GtkWidget* findToolbar;
  GtkWidget* mainToolbar;
  gboolean findToolbarIsVisible;
  gboolean mainToolbarIsVisible;

  HildonProgram* program;
  HildonWindow* window;
} ApplicationState;

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

/**
 * Utility function to print a GnomeVFS I/O related error message to
 * standard error (not seen by the user in graphical mode) (NEW).
 */
static void dbgFileError(GnomeVFSResult errCode, const gchar* uri) {
  g_printerr("Error while accessing '%s': %s\n", uri,
             gnome_vfs_result_to_string(errCode));
}

/**
 * MODIFIED (A LOT)
 *
 * We read in the file selected by the user if possible and set the
 * contents of the file as the new Label content.
 *
 * If reading the file will fail, we leave the label unchanged.
 */
static void cbActionOpen(GtkWidget* widget, ApplicationState* app) {

  gchar* filename = NULL;
  /* We need to use URIs with GnomeVFS, so declare one here. */
  gchar* uri = NULL;

  g_assert(app != NULL);
  /* Bad things will happen if these two widgets don't exist. */
  g_assert(GTK_IS_LABEL(app->textLabel));
  g_assert(GTK_IS_WINDOW(app->window));

  g_print("cbActionOpen invoked\n");

  /* Ask the user to select a file to open. */
  filename = runFileChooser(app, GTK_FILE_CHOOSER_ACTION_OPEN);
  if (filename) {
    /* This will point to loaded data buffer. */
    gchar* buffer = NULL;
    /* Pointer to a structure describing an open GnomeVFS "file". */
    GnomeVFSHandle* fileHandle = NULL;
    /* Structure to hold information about a "file", initialized to
       zero. */
    GnomeVFSFileInfo fileInfo = {};
    /* Result code from the GnomeVFS operations. */
    GnomeVFSResult result;
    /* Size of the file (in bytes) that we'll read in. */
    GnomeVFSFileSize fileSize = 0;
    /* Number of bytes that were read in successfully. */
    GnomeVFSFileSize readCount = 0;

    g_print("  you chose to load file '%s'\n", filename);
```

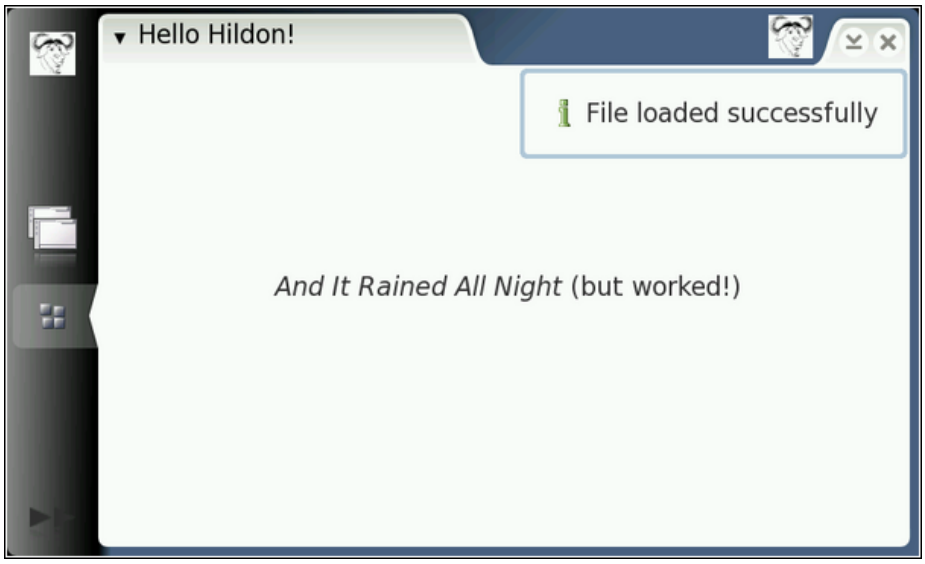
```
    /* Convert the filename into an GnomeVFS URI. */
    uri = gnome_vfs_get_uri_from_local_path(filename);
    /* We don't need the original filename anymore. */
    g_free(filename);
    filename = NULL;
    /* Should not happen since we got a filename before. */
    g_assert(uri != NULL);
    /* Attempt to get file size first. We need to get information
       about the file and aren't interested in other than the very
       basic information, so we'll use the INFO_DEFAULT setting. */
    result = gnome_vfs_get_file_info(uri, &fileInfo,
                                     GNOME_VFS_FILE_INFO_DEFAULT);
    if (result != GNOME_VFS_OK) {
      /* There was a failure. Print a debug error message and break
         out into error handling. */
      dbgFileError(result, uri);
      goto error;
    }

    /* We got the information (maybe). Let's check whether it
       contains the data that we need. */
    if (fileInfo.valid_fields & GNOME_VFS_FILE_INFO_FIELDS_SIZE) {
      /* Yes, we got the file size. */
      fileSize = fileInfo.size;
    } else {
      g_printerr("Couldn't get the size of file!\n");
      goto error;
    }

    /* By now we have the file size to read in. Check for some limits
       first. */
    if (fileSize > 1024*100) {
      g_printerr("Loading over 100KiB files is not supported!\n");
      goto error;
    }
    /* Refuse to load empty files. */
    if (fileSize == 0) {
      g_printerr("Refusing to load an empty file\n");
      goto error;
    }

    /* Allocate memory for the contents and fill it with zeroes.
       NOTE:
         We leave space for the terminating zero so that we can pass
         this buffer as gchar to string functions and it is
         guaranteed to be terminated, even if the file doesn't end
         with binary zero (odds of that happening are small). */
    buffer = g_malloc0(fileSize+1);
    if (buffer == NULL) {
      g_printerr("Failed to allocate %u bytes for buffer\n",
                 (guint)fileSize);
      goto error;
    }

    /* Open the file.

       Parameters:
       - A pointer to the location where to store the address of the
         new GnomeVFS file handle (created internally in open).
       - uri: What to open (needs to be GnomeVFS URI).
       - open-flags: Flags that tell what we plan to use the handle
         for. This will affect how permissions are checked by the
```

```
        Linux kernel. */

  result = gnome_vfs_open(&fileHandle, uri, GNOME_VFS_OPEN_READ);
  if (result != GNOME_VFS_OK) {
    dbgFileError(result, uri);
    goto error;
  }
  /* File opened succesfully, read its contents in. */
  result = gnome_vfs_read(fileHandle, buffer, fileSize,
                          &readCount);
  if (result != GNOME_VFS_OK) {
    dbgFileError(result, uri);
    goto error;
  }
  /* Verify that we got the amount of data that we requested.
     NOTE:
       With URIs it won't be an error to get less bytes than you
       requested. Getting zero bytes will however signify an
       End-of-File condition. */
  if (fileSize != readCount) {
    g_printerr("Failed to load the requested amount\n");
    /* We could also attempt to read the missing data until we have
       filled our buffer, but for simplicity, we'll flag this
       condition as an error. */
    goto error;
  }

  /* Whew, if we got this far, it means that we actually managed to
     load the file into memory. Let's set the buffer contents as
     the new label now. */
  gtk_label_set_markup(GTK_LABEL(app->textLabel), buffer);

  /* That's it! Display a message of great joy. For this we'll use
     a dialog (non-modal) designed for displaying short
     informational messages. It will linger around on the screen
     for a while and then disappear (in parallel to our program
     continuing). */
  hildon_banner_show_information(GTK_WIDGET(app->window),
    NULL, /* Use the default icon (info). */
    "File loaded successfully");

  /* Jump to the resource releasing phase. */
  goto release;

error:
  /* Display a failure message with a stock icon.
     Please see http://maemo.org/api_refs/4.0/gtk/gtk-Stock-Items.
         html
     for a full listing of stock items. */
  hildon_banner_show_information(GTK_WIDGET(app->window),
    GTK_STOCK_DIALOG_ERROR, /* Use the stock error icon. */
    "Failed to load the file");

release:
  /* Close and free all resources that were allocated. */
  if (fileHandle) gnome_vfs_close(fileHandle);
  if (filename) g_free(filename);
  if (uri) g_free(uri);
  if (buffer) g_free(buffer);
  /* Zero them all out to prevent stack-reuse-bugs. */
  fileHandle = NULL;
  filename = NULL;
```

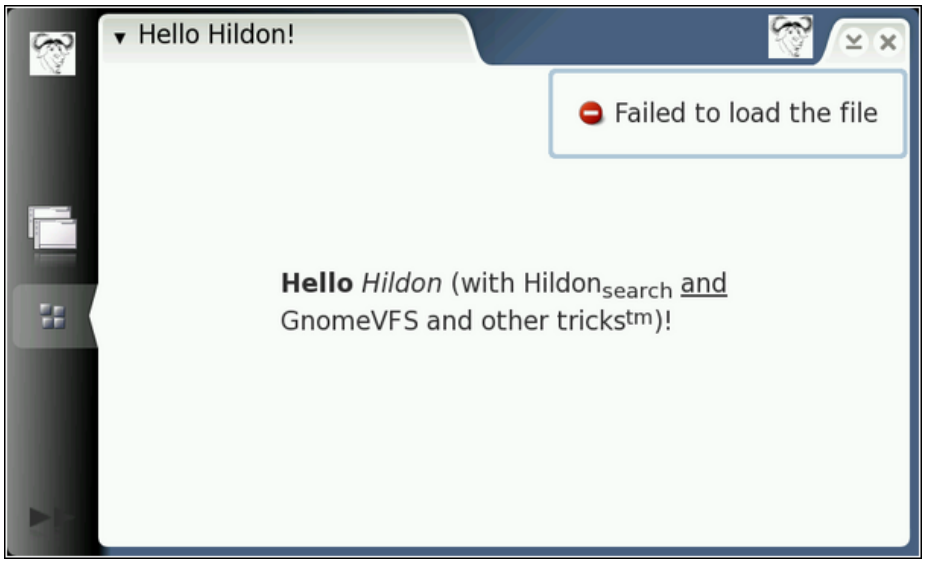
```
    uri = NULL;
    buffer = NULL;

    return;
  } else {
    g_print("  you didn't choose any file to open\n");
  }
}

/**
 * MODIFIED (kind of)
 *
 * Function to save the contents of the label (although it doesn't
 * actually save the contents, on purpose). Use gtk_label_get_label
 * to get a gchar pointer into the application label contents
 * (including current markup), then use gnome_vfs_create and
 * gnome_vfs_write to create the file (left as an exercise).
 */
static void cbActionSave(GtkWidget* widget, ApplicationState* app) {
  gchar* filename = NULL;

  g_assert(app != NULL);

  g_print("cbActionSave invoked\n");

  filename = runFileChooser(app, GTK_FILE_CHOOSER_ACTION_SAVE);
  if (filename) {
    g_print("  you chose to save into '%s'\n", filename);
    /* Process saving .. */

    g_free(filename);
    filename = NULL;
  } else {
    g_print("  you didn't choose a filename to save to\n");
  }
}

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

/**
 * MODIFIED
 *
 * Add support for GnomeVFS (it needs to be initialized before use)
 * and add support for the Pango markup feature of the GtkLabel
 * widget.
 */
int main(int argc, char** argv) {

  ApplicationState aState = {};

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

  /* Initialize the GnomeVFS (NEW). */
  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 the label widget, with Pango marked up content (NEW). */
  label = gtk_label_new("<b>Hello</b> <i>Hildon</i> (with Hildon"
                        "<sub>search</sub> <u>and</u> GnomeVFS "
                        "and other tricks<sup>tm</sup>)!");

  /* Allow lines to wrap (NEW). */
  gtk_label_set_line_wrap(GTK_LABEL(label), TRUE);

  /* Tell the GtkLabel widget to support the Pango markup (NEW). */
  gtk_label_set_use_markup(GTK_LABEL(label), TRUE);

  /* Store the widget pointer into the application state so that the
     contents can be replaced when a file will be loaded (NEW). */
  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 and exiting with success\n");

  return EXIT_SUCCESS;
}
```

Listing 1.1: Implementing file reading (hildon_helloworld-8.c)



Loading succeeded



Loading failed

In order to experiment with loading other content you can create a simple file containing Pango markup like this: **echo "Hello world" > MyDocs/hello.txt** , and then load hello.txt.

As you might imagine, we have only scratched the surface of GnomeVFS, which is quite a rich library and contains a broad API and a large amount of plug-ins. We completely avoided directory content iteration and the asyn-

chronous interface, callback signalling on directory content changes and so on. Please see maemo.org for more information. The API also contains some mini-tutorials on various GnomeVFS topics, so it's well worth the time spent reading. You will also note that GnomeVFS has been overloaded with functions which are not even file-operation related (like ZeroConf and creating TCP/IP connections and so on).

You don't need to use GTK+ in order to use `GnomeVFS`. One such example program is Midnight Commander (a Norton Commander clone, but better) which is a menu-based "text"-mode program. GnomeVFS uses GLib though, so if you decide you want to use GnomeVFS, you should think about using GLib as well, as it will be loaded anyway.

1.3 Storing user preferences

The traditional way to store user preferences on disk has been the resource configuration files (**.rc**). These are normally ASCII (not often UTF-8) formatted text files with common UNIX commenting supported. Clearly, using such files requires one to write a parser to read the file in, and some code to write the `rc-file` back. If the author is especially lazy, there won't be any interactive way of changing preferences anyhow, so only a parser is required. Of course in non-interactive programs storing preferences back doesn't make sense, so you won't find any code to that in those.

These resource files were emulated (in a way) in other operating systems as **.INI**-files, but were replaced with binary settings databases later on (also known as the "registry").

1.4 GConf basics

GConf is a system for GNOME applications to store settings into a database system in a centralised manner. The aim of the GConf library is to provide applications a consistent view of the database access functions as well as to provide tools for system administrators to enable them to distribute software settings in a centralised manner (across multiple computers).

The GConf "database" may consist of multiple databases (configured by the system administrator), but normally you will find at least one database engine that uses XML to store settings. This keeps the database still in a human understandable form (as opposed to binary) and allows some consistency checks via schema verifications.

The interface for the client (program that uses GConf to store its settings) is always the same irrespective of the database back-end (the client doesn't see this).

What makes GConf interesting, is its capability of notifying running clients that their settings have been changed by some other process than themselves. This allows for the clients to react soon (not quite real-time) and this leads to a situation where a user will change the GNOME HTTP-proxy settings (for example) and clients that are interested in that setting will get a notification (via a callback function) that the setting has changed. The clients then read

the new setting and modify their data structures to take into account the new setting.

1.5 Using GConf

The GConf model consists of two parts: the GConf client library (which we'll be using) and the GConf server daemon that is the guardian and reader/writer of the back-end databases. In regular GNOME environment, the client communicates with the server either by using the Bonobo-library (light-weight object IPC-mechanism) or D-Bus.

As Bonobo is not used in maemo (it is quite heavy, even if light-weight), the client will communicate with the server using D-Bus. This also allows the daemon to be started on demand when there is at least one client wanting to use that service (this is a feature of D-Bus). The communication mechanism is encapsulated by the GConf client library, and as such, will be transparent to us.

In order to read or write the preference database, you will need to decide on the key to use to access your application values. The database namespace is hierarchical, and uses the '/'-character to implement this hierarchy, starting from a root location similar to UNIX filesystem namespace.

Each application will use its own "directory" under **/apps/Maemo/app_name/**. Note that even when you see the word "directory" in connection to GConf, you'll have to be careful to distinguish **real directories** from **preference namespaces** inside the GConf namespace. The **/apps/Maemo/app_name/** above is in the GConf namespace, so you won't actually find a physical directory called **/apps/** on your system.

The keys should be named according to the platform guidelines. The current guideline is that each application should store its configuration keys under `/apps/Maemo/appname/` where `appname` is the name of your application. There is no central registry on the names in use currently, so be careful when selecting your name. Key-names should all be lowercase with underscore used to separate multiple words. Also, use ASCII since GConf does not support localisation for key names (it does for key values but that is not covered in this material).

GConf values are typed, which means that you will have to select the type for the data that you want your key to hold.

The following types are supported for values in GConf:

- `gint` (32-bit signed)
- `gboolean`
- `gchar` (ASCII/ISO 8859-1/UTF-8 C string)
- `gfloat` (with the limitation that the resolution is not guaranteed nor specified by GConf because of portability issues)
- a list of values of one type
- a pair of values, each having their own type (useful for storing "mapping" data)

What is missing from the above list is storing binary data (for good reasons). The type system is also fairly limited. This is on purpose so that complex configurations (like the Apache HTTP-daemon uses, or Samba) are not attempted using GConf.

There is a diagnostic and administration tool called `gconftool-2` which is also available in the SDK. You can set and unset keys with it as well as display their current contents.

Some examples of using `gconftool-2` (on the SDK):

```
[sbox-DIABLO_X86: ~] > run-standalone.sh gconftool-2 -R /apps
 /apps/osso:
  /apps/osso/inputmethod:
   launch_finger_kb_on_select = true
   input_method_plugin = himExample_vkb
   available_languages = [en_GB]
   use_finger_kb = true
   /apps/osso/inputmethod/hildon-im-languages:
    language-0 = en_GB
    current = 0
    language-1 =
    list = []
  /apps/osso/fontconfig:
   font_scaling_factor = Schema (type: 'float' list_type:
    '*invalid*' car_type: '*invalid*' cdr_type: '*invalid*'
    locale: 'C')
  /apps/osso/apps:
   /apps/osso/apps/controlpanel:
    groups = [copa_ia_general,copa_ia_connectivity,
              copa_ia_personalisation]
    icon_size = false
    group_ids = [general,connectivity,personalisation]
  /apps/osso/osso:
   /apps/osso/osso/thumbnailers:
    /apps/osso/osso/thumbnailers/audio@x-mp3:
     command = /usr/bin/hildon-thumb-libid3
    /apps/osso/osso/thumbnailers/audio@x-m4a:
     command = /usr/bin/hildon-thumb-libid3
    /apps/osso/osso/thumbnailers/audio@mp3:
     command = /usr/bin/hildon-thumb-libid3
    /apps/osso/osso/thumbnailers/audio@x-mp2:
     command = /usr/bin/hildon-thumb-libid3
```

Displaying the contents of all keys stored under **/apps/** (listing cut for brevity).

```
[sbox-DIABLO_X86: ~] > run-standalone.sh gconftool-2 \
 --set /apps/Maemo/testing/testkey --type=int 5
```

Creating and setting the value to a new key.

```
[sbox-DIABLO_X86: ~] > run-standalone.sh gconftool-2 \
 -R /apps/Maemo/testing
 testkey = 5
```

Listing all keys under the namespace **/apps/Maemo/testing**.

```
[sbox-DIABLO_X86: ~] > run-standalone.sh gconftool-2 \
 --unset /apps/Maemo/testing/testkey
[sbox-DIABLO_X86: ~] > run-standalone.sh gconftool-2 \
 -R /apps/Maemo/testing
```

Removing the last key will also remove the key directory.

```
[sbox-DIABLO_X86: ~] > run-standalone.sh gconftool-2 \
 --recursive-unset /apps/Maemo/testing
```

Removing whole key hierarchies is also possible.

For more detailed information, please see the GConf API documentation at maemo.org.

1.6 Using GConf to read and write preferences

We'll now implement a simple GConf client logic into our application.

We want to:

- Store the color that the user selects when the color button (in the toolbar) is used.
- Load the color preference on application startup.

You'll see that even if GConf concepts seem to be logical, using GConf will require you to learn some new things (the GError-object for example). Since the GConf client code is in its own library, you will again need to add the relevant compiler flags and library options. The pkg-config package name is `gconf-2.0`.

```
/**
 * hildon_helloworld-9.c
 *
 * 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.
 *
 * We'll store the color that the user selects into a GConf
 * preference. In fact, we'll have three settings, one for each
 * channel of the color (red, green and blue).
 *
 * 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 the prototypes for GConf client functions (NEW). */
#include <gconf/gconf-client.h>

/* The application name -part of the GConf namespace (NEW). */
#define APP_NAME "hildon_hello"
/* This will be the root "directory" for our preferences (NEW). */
#define GC_ROOT  "/apps/Maemo/" APP_NAME "/"

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

/**
 * NEW
 *
 * Utility function to store the given color into our application
 * preferences. We could use a list of integers as well, but we'll
 * settle for three separate properties; one for each of RGB
```

```
 * channels.
 *
 * The config keys that will be used are 'red', 'green' and 'blue'.
 *
 * NOTE:
 *   We're doing things very non-optimally. If our application would
 *   have multiple preference settings, and we would like to know
 *   when someone will change them (external program, another
 *   instance of our program, etc), we'd have to keep a reference to
 *   the GConf client connection. Listening for changes in
 *   preferences would also require a callback registration, but this
 *   is covered in the "maemo Platform Development" material.
 */
static void confStoreColor(const GdkColor* color) {

  /* We'll store the pointer to the GConf connection here. */
  GConfClient* gcClient = NULL;

  /* Make sure that no NULLs are passed for the color. GdkColor is
     not a proper GObject, so there is no GDK_IS_COLOR macro. */
  g_assert(color);

  g_print("confStoreColor: invoked\n");

  /* Open a connection to gconfd-2 (via D-Bus in maemo). The GConf
     API doesn't say whether this function can ever return NULL or
     how it will behave in error conditions. */
  gcClient = gconf_client_get_default();
  /* We make sure that it's a valid GConf-client object. */
  g_assert(GCONF_IS_CLIENT(gcClient));

  /* Store the values. */
  if (!gconf_client_set_int(gcClient, GC_ROOT "red", color->red,
                            NULL)) {
    g_warning(" failed to set %s/red to %d\n", GC_ROOT, color->red);
  }
  if (!gconf_client_set_int(gcClient, GC_ROOT "green", color->green,
                            NULL)) {
    g_warning(" failed to set %s/green to %d\n", GC_ROOT,
              color->green);
  }
  if (!gconf_client_set_int(gcClient, GC_ROOT "blue", color->blue,
                            NULL)) {
    g_warning(" failed to set %s/blue to %d\n", GC_ROOT,
              color->blue);
  }

  /* Release the GConf client object (with GObject-unref). */
  g_object_unref(gcClient);
  gcClient = NULL;
}

/**
 * NEW
 *
 * An utility function to get an integer but also return the status
 * whether the requested key existed or not.
 *
 * NOTE:
 *   It's also possible to use gconf_client_get_int(), but it's not
 *   possible to then know whether they key existed or not, because
 *   the function will return 0 if the key doesn't exist (and if the
```

```
 *   value is 0, how could you tell these two conditions apart?).
 *
 * Parameters:
 * - GConfClient: the client object to use
 * - const gchar*: the key
 * - gint*: the address to store the integer to if the key exists
 *
 * Returns:
 * - TRUE: if integer has been updated with a value from GConf.
 *   FALSE: there was no such key or it wasn't an integer.
 */
static gboolean confGetInt(GConfClient* gcClient, const gchar* key,
                           gint* number) {

  /* This will hold the type/value pair at some point. */
  GConfValue* val = NULL;
  /* Return flag (tells the caller whether this function wrote behind
     the 'number' pointer or not). */
  gboolean hasChanged = FALSE;

  /* Try to get the type/value from the GConf DB.
     NOTE:
       We're using a version of the getter that will not return any
       defaults (if a schema would specify one). Instead, it will
       return the value if one has been set (or NULL).

     We're not really interested in errors as this will return a NULL
     in case of missing keys or errors and that is quite enough for
     us. */
  val = gconf_client_get_without_default(gcClient, key, NULL);
  if (val == NULL) {
    /* Key wasn't found, no need to touch anything. */
    g_warning("confGetInt: key %s not found\n", key);
    return FALSE;
  }

  /* Check whether the value stored behind the key is an integer. If
     it is not, we issue a warning, but return normally. */
  if (val->type == GCONF_VALUE_INT) {
    /* It's an integer, get it and store. */
    *number = gconf_value_get_int(val);
    /* Mark that we've changed the integer behind 'number'. */
    hasChanged = TRUE;
  } else {
    g_warning("confGetInt: key %s is not an integer\n", key);
  }

  /* Free the type/value-pair. */
  gconf_value_free(val);
  val = NULL;

  return hasChanged;
}

/**
 * NEW
 *
 * Utility function to change the given color into the one that is
 * specified in application preferences.
 *
 * If some key is missing, that channel is left untouched. The
 * function also checks for proper values for the channels so that
```

```
 * invalid values are not accepted (guint16 range of GdkColor).
 *
 * Parameters:
 * - GdkColor*: the color structure to modify if changed from prefs.
 *
 * Returns:
 * - TRUE if the color was been changed by this routine.
 *   FALSE if the color wasn't changed (there was an error or the
 *   color was already exactly the same as in the preferences).
 */
static gboolean confLoadCurrentColor(GdkColor* color) {

  GConfClient* gcClient = NULL;
  /* Temporary holders for the pref values. */
  gint red = -1;
  gint green = -1;
  gint blue = -1;
  /* Temp variable to hold whether the color has changed. */
  gboolean hasChanged = FALSE;

  g_assert(color);

  g_print("confLoadCurrentColor: invoked\n");

  /* Open a connection to gconfd-2 (via d-bus). */
  gcClient = gconf_client_get_default();
  /* Make sure that it's a valid GConf-client object. */
  g_assert(GCONF_IS_CLIENT(gcClient));

  if (confGetInt(gcClient, GC_ROOT "red", &red)) {
    /* We got the value successfully, now clamp it. */
    g_print(" got red = %d, ", red);
    /* We got a value, so let's limit it between 0 and 65535 (the
       legal range for guint16). We use the CLAMP macro from GLib for
       this. */
    red = CLAMP(red, 0, G_MAXUINT16);
    g_print("after clamping = %d\n", red);
    /* Update & mark that at least this component changed. */
    color->red = (guint16)red;
    hasChanged = TRUE;
  }
  /* Repeat the same logic for the green component. */
  if (confGetInt(gcClient, GC_ROOT "green", &green)) {
    g_print(" got green = %d, ", green);
    green = CLAMP(green, 0, G_MAXUINT16);
    g_print("after clamping = %d\n", green);
    color->green = (guint16)green;
    hasChanged = TRUE;
  }
  /* Repeat the same logic for the last component (blue). */
  if (confGetInt(gcClient, GC_ROOT "blue", &blue)) {
    g_print(" got blue = %d, ", blue);
    blue = CLAMP(blue, 0, G_MAXUINT16);
    g_print("after clamping = %d\n", blue);
    color->blue = (guint16)blue;
    hasChanged = TRUE;
  }

  /* Release the client object (with GObject-unref). */
  g_object_unref(gcClient);
  gcClient = NULL;
```

```
  /* Return status if the color was been changed by this routine. */
  return hasChanged;
}

/**
 * MODIFIED
 *
 * Invoked when the user selects a color (or will cancel the dialog).
 *
 * Will also write the color to preferences (GConf) each time the
 * color changes. We'll compare whether it has really changed (to
 * avoid writing to GConf is nothing really changed).
 */
static void cbActionColorChanged(HildonColorButton* colorButton,
                                 ApplicationState* app) {

  /* Local variables that we'll need to handle the change (NEW). */
  gboolean hasChanged = FALSE;
  GdkColor newColor = {};
  GdkColor* curColor = NULL;

  g_assert(app != NULL);

  g_print("cbActionColorChanged invoked\n");
  /* Retrieve the new color from the color button (NEW). */
  hildon_color_button_get_color(colorButton, &newColor);
  /* Just an alias to save some typing (could also use
     app->currentColor) (NEW). */
  curColor = &app->currentColor;

  /* Check whether the color really changed (NEW). */
  if ((newColor.red   != curColor->red) ||
      (newColor.green != curColor->green) ||
      (newColor.blue  != curColor->blue)) {
    hasChanged = TRUE;
  }
  if (!hasChanged) {
    g_print(" color not really changed\n");
    return;
  }
  /* Color really changed, store to preferences (NEW). */
  g_print(" color changed, storing into preferences.. \n");
  confStoreColor(&newColor);
  g_print(" done.\n");

  /* Update the changed color into the application state. */
  app->currentColor = newColor;
}

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

/**
 * MODIFIED
 *
 * The color of the color button will be loaded from the application
 * preferences (or keep the default if preferences have no setting).
 */
static GtkWidget* buildToolbar(ApplicationState* app) {

  GtkToolbar*  toolbar = NULL;
  GtkToolItem* tbOpen = NULL;
  GtkToolItem* tbSave = NULL;
```

```
  GtkToolItem* tbSep = NULL;
  GtkToolItem* tbFind = NULL;
  GtkToolItem* tbColorButton = NULL;
  GtkWidget*   colorButton = NULL;

  g_assert(app != NULL);

  tbOpen = gtk_tool_button_new_from_stock(GTK_STOCK_OPEN);
  tbSave = gtk_tool_button_new_from_stock(GTK_STOCK_SAVE);
  tbSep  = gtk_separator_tool_item_new();
  tbFind = gtk_tool_button_new_from_stock(GTK_STOCK_FIND);

  tbColorButton = gtk_tool_item_new();
  colorButton = hildon_color_button_new();
  /* Copy the color from the color button into the application state.
     This is done to detect whether the color in preferences matches
     the default color or not (NEW). */
  hildon_color_button_get_color(HILDON_COLOR_BUTTON(colorButton),
                                &app->currentColor);
  /* Load preferences and change the color if necessary. */
  g_print("buildToolbar: loading color pref.\n");
  if (confLoadCurrentColor(&app->currentColor)) {
    g_print(" color not same as default one\n");
    hildon_color_button_set_color(HILDON_COLOR_BUTTON(colorButton),
                                  &app->currentColor);
  } else {
    g_print(" loaded color same as default\n");
  }
  gtk_container_add(GTK_CONTAINER(tbColorButton), colorButton);

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

}
```

Listing 1.2: Implementing preference storage (hildon_helloworld-9.c)

Since the graphical appearance of the program doesn't change (except that the ColorButton will display the correct initial color), we'll look at the stdout display of the program.

```
[sbox-DIABLO_X86: ~/appdev] > run-standalone.sh ./hildon_helloworld-9
buildToolbar: loading color pref.
confLoadCurrentColor: invoked
hildon_helloworld-9[19840]: GLIB WARNING \*\* default -
 confGetInt: key /apps/Maemo/hildon_hello/red not found
hildon_helloworld-9[19840]: GLIB WARNING \*\* default -
 confGetInt: key /apps/Maemo/hildon_hello/green not found
hildon_helloworld-9[19840]: GLIB WARNING \*\* default -
 confGetInt: key /apps/Maemo/hildon_hello/blue not found
 loaded color same as default
main: calling gtk_main
cbActionMainToolbarToggle invoked
cbActionColorChanged invoked
 color changed, storing into preferences..
confStoreColor: invoked
 done.
main: returned from gtk_main and exiting with success
```

Running the program, selecting a color from color button.

When running the program for the first time, we expect the warnings about the missing keys (since the values weren't present in GConf).

Run the program again and exit:

```
[sbox-DIABLO_X86: ~/appdev] > run-standalone.sh ./hildon_helloworld-9
buildToolbar: loading color pref.
confLoadCurrentColor: invoked
 got red = 65535, after clamping = 65535
 got green = 65535, after clamping = 65535
 got blue = 0, after clamping = 0
 color not same as default one
main: calling gtk_main
main: returned from gtk_main and exiting with success
```

Running the program second time.

We now remove one key (**red**) and run the program again (this is to test and verify that our logic works):

```
[sbox-DIABLO_X86: ~/appdev] > run-standalone.sh gconftool-2 \
 --unset /apps/Maemo/hildon_hello/red
[sbox-DIABLO_X86: ~/appdev] > run-standalone.sh gconftool-2 \
 -R /apps/Maemo/hildon_hello
 green = 65535
 blue = 0
[sbox-DIABLO_X86: ~/appdev] > run-standalone.sh ./hildon_helloworld-9
buildToolbar: loading color pref.
confLoadCurrentColor: invoked
hildon_helloworld-9[19924]: GLIB WARNING \*\* default -
 confGetInt: key /apps/Maemo/hildon_hello/red not found
 got green = 65535, after clamping = 65535
 got blue = 0, after clamping = 0
 color not same as default one
main: calling gtk_main
main: returned from gtk_main and exiting with success
```

Removing one of the preference keys and testing resilience of the program.