Maemo Diablo The Final Program
Training Material

February 9, 2009

Contents

Chapter 1

The Final Program

1.1 Appendix contents

This appendix presents the final hello world program and the necessary support files. This list does not include the files that result from Debianisation.

1.2 Autoconfigure driver

```
# The package name is hhwX (hildon helloworld 10).
#
# This will be lowercased when automake will create the distribution
# directories. The version number is currently set at 0.1, but new
# distributed versions should change this number (no other variables
# need be touched).
AC_INIT(hhwX, 0.1)

# Tell automake to prepare for real work.
AM_INIT_AUTOMAKE
# Check for the C compiler.
AC_PROG_CC
# Check that 'install' program is available (used by the automake
# generated Makefiles).
AC_PROG_INSTALL

# Check whether the necessary pkg-config packages are present. The
# PKG_CHECK_MODULES macro is supplied by pkg-config
# (/usr/share/aclocal/).
#
# The first parameter will be the variable name prefix that will be
# used to create two variables: one to hold the CFLAGS required by
# the packages, and one to hold the LDFLAGS (LIBS) required by the
# packages. The variable name prefix (HHW) can be chosen freely.
PKG_CHECK_MODULES(HHW, gtk+-2.0 hildon-1 hildon-fm-2 gnome-vfs-2.0 \
                       gconf-2.0 libosso)
# At this point HHW_CFLAGS will contain the necessary compiler flags
# and HHW_LIBS will contain the linker options necessary for all the
# packages listed above.
#
# Add the pkg-config supplied values to the ones that are used by
# Makefile or supplied by the user running ./configure.
CFLAGS="$HHW_CFLAGS $CFLAGS"
```

```
LIBS="$HHW_LIBS $LIBS"

# Generate the Makefile from Makefile.in
AC_OUTPUT(Makefile)

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

Listing 1.1: Contents of autotoolized-hildon-helloworld/configure.ac

1.3 Automake configuration

```
# 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 1.2: Contents of autotoolized-hildon-helloworld/Makefile.am

1.4 Desktop file template for AF

```
[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 1.3: Contents of autotoolized-hildon-helloworld/hhwX.desktop.in

1.5 Service file template for AF

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

Listing 1.4: Contents of autotoolized-hildon-helloworld/org.maemo.hhwX.service.in

1.6 Development bootstrap (autogen)

```
#!/bin/sh
#
# An utility script to setup the autoconf environment for the first
# time. Normally this script would be run when checking out a
# development version of the software from SVN/version control.
# Regular users expect to download .tar.gz/tar.bz2 source code
# instead, and those should come with with 'configure' script so that
# users do not require the autoconf/automake tools.
#

# Scan configure.ac and copy the necessary macros into aclocal.m4.
aclocal

# Generate Makefile.in from Makefile.am (and copy necessary support
# files, because of -ac).
automake -ac

# This step is not normally necessary, but documented here for your
# convenience. The files listed below need to be present to stop
# automake from complaining during various phases of operation.
#
# You also should consider maintaining these files separately once
# you release your project into the wild.
#
# touch NEWS README AUTHORS ChangeLog

# Run autoconf (will create the 'configure'-script).
autoconf

echo 'Ready to go (run configure)'
```

Listing 1.5: Contents of autotoolized-hildon-helloworld/autogen.sh

1.7 Development cleanup (antigen)

```
#!/bin/sh
#
# An utility script to remove all generated files.
#
# Running autogen.sh will be required after running this script since
# the 'configure' script will also be removed.
#
# This script is mainly useful when testing autoconf/automake changes
# and as a part of their development process.

# If there's a Makefile, then run the 'distclean' target first (which
# will also remove the Makefile).
if test -f Makefile; then
  make distclean
```

```
fi

# Remove all tar-files (assuming there are some packages).
rm -f *.tar.* *.tgz

# Also remove the autotools cache directory.
rm -Rf autom4te.cache

# Remove rest of the generated files.
rm -f Makefile.in aclocal.m4 configure depcomp install-sh missing
```

Listing 1.6: Contents of autotoolized-hildon-helloworld/antigen.sh

1.8 Program listing

```
/**
 * 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.
 *
 * Source code of hhwX.c with comments.
 */

#include <stdlib.h>
#include <hildon/hildon-program.h>
#include <hildon/hildon-color-button.h>
/* Pull in the Hildon Find toolbar declarations. */
#include <hildon/hildon-find-toolbar.h>
/* We need HildonFileChooserDialog.
   NOTE:
     The include file is not in the same location as the other Hildon
     widgets, but instead is part of the hildon-fm-2 package. So
     in fact, the "hildon/"-prefix below points to a completely
     different directories than the ones above. */
#include <hildon/hildon-file-chooser-dialog.h>
/* A small notification window widget. */
#include <hildon/hildon-banner.h>
/* Pull in the GnomeVFS headers. */
#include <libgnomevfs/gnome-vfs.h>
/* Include the prototypes for GConf client functions. */
#include <gconf/gconf-client.h>
/* Pull in the LibOSSO library declarations. */
#include <libosso.h>

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

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

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

```
};

/**
 * This is all of the data that our application needs to run
 * properly. Rest of the data is not needed by our application so we
 * can leave that for GTK+ to handle (references and all).
 */
typedef struct {
  /* Underlining active for text? Either TRUE or FALSE. */
  gboolean styleUseUnderline;
  /* Currently selected slant for text. Either STYLE_SLANT_NORMAL or
     STYLE_SLANT_ITALIC. */
  gboolean styleSlant;

  /* The currently selected color. */
  GdkColor currentColor;

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

  /* Are we in fullscreen mode or not? */
  gboolean fullScreen;

  /* We need to keep pointers to these two widgets so that we can
     control their visibility from callbacks. */
  GtkWidget* findToolbar;
  GtkWidget* mainToolbar;
  /* We'll also keep visibility flags for both of the toolbars.

     You could also test widget-visibility like this:
     if (GTK_WIDGET_VISIBLE(widget)) { .. }; */
  gboolean findToolbarIsVisible;
  gboolean mainToolbarIsVisible;

  /* Pointer to our HildonProgram. */
  HildonProgram* program;
  /* Pointer to our main Window. */
  HildonWindow* window;
} ApplicationState;

/**
 * Turns the delete event from the top-level Window into a window
 * destruction signal (by returning FALSE).
 */
static gboolean cbEventDelete(GtkWidget* widget, GdkEvent* event,
                              ApplicationState* app) {
  return FALSE;
}

/**
 * Handles the 'destroy' signal by quitting the application.
 */
static void cbActionTopDestroy(GtkWidget* widget,
                               ApplicationState* app) {
  gtk_main_quit();
}

/**
 * Create a file chooser dialog and return a filename that user
 * selects.
 *
```

```
 * Parameters:
 * - application state: we need a pointer to the main application
 *   window and HildonProgram is extended from GtkWindow, so we'll
 *   use that. This is because we want to create a modal dialog
 *   (which normally would be a bad idea, but not always for
 *   applications designed for maemo).
 * - what kind of file chooser should be displayed:
 *   GTK_FILE_CHOOSER_ACTION_OPEN or _SAVE.
 *
 * Returns:
 * - A newly allocated string that we need to free ourselves or NULL
 *   if user will cancel the dialog.
 */
static gchar* runFileChooser(ApplicationState* app,
                             GtkFileChooserAction style) {

  GtkWidget* dialog = NULL;
  gchar* filename = NULL;

  g_assert(app != NULL);

  g_print("runFilechooser: invoked\n");

  /* Create the dialog (not shown yet). */
  dialog = hildon_file_chooser_dialog_new(GTK_WINDOW(app->window),
                                          style);

  /* Enable its visibility. */
  gtk_widget_show_all(GTK_WIDGET(dialog));

  /* Divert the GTK+ main loop to handle events from this dialog.
     We'll return into this function when the dialog will be exited
     by the user. The dialog resources will still be allocated at
     that point. */
  g_print(" running dialog\n");
  if (gtk_dialog_run(GTK_DIALOG(dialog)) == GTK_RESPONSE_OK) {
    /* We got something from the dialog at least. Copy a point to it
       into filename and we'll return that to caller shortly. */
    filename =
      gtk_file_chooser_get_filename(GTK_FILE_CHOOSER(dialog));
  }
  g_print(" dialog completed\n");

  /* Destroy all the resources of the dialog. */
  gtk_widget_destroy(dialog);

  if (filename != NULL) {
    g_print(" user selected filename '%s'\n", filename);
  } else {
    g_print(" user didn't select any filename\n");
  }

  return filename;
}

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

```
}

/**
 * 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");
  }
}

/**
 * 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");
  }
}

/**
 * Terminate the program since user wishes to do so.
 */
static void cbActionQuit(GtkWidget* widget, ApplicationState* app) {

  g_print("cbActionQuit invoked. Terminating using gtk_main_quit\n");
  gtk_main_quit();
}

/**
 * Update the underlining status based on a signal.
 */
static void cbActionUnderlineToggled(GtkCheckMenuItem* item,
                                     ApplicationState* app) {

  /* Verify that 'app' is not NULL by using a GLib function which
     checks whether the given C statement evaluates to 0(false) or
     non-zero(true). Will terminate the program on FALSE assertions
     with an error message (using abort()). */
  g_assert(app != NULL);
  /* Normally we'd also need to check that the 'item' is of the
     correct type with GTK_CHECK_MENU_ITEM and put that inside an if-
     statement which would create a protective block around us. We'll
     implement this in another function below. */
  g_print("cbActionUnderlineToggled invoked\n");
  app->styleUseUnderline = gtk_check_menu_item_get_active(item);
  g_print(" underlining is now %s\n",
          app->styleUseUnderline?"on":"off");
}

/**
 * The 'Normal' item has been toggled in the Style-menu.
 */
static void cbActionStyleNormalToggled(GtkCheckMenuItem* item,
                                       ApplicationState* app) {

  g_assert(app != NULL);

  g_print("cbActionStyleNormalToggled invoked\n");
  /* We will switch slant if and only if the item is active. */
  if (gtk_check_menu_item_get_active(item)) {
    app->styleSlant = STYLE_SLANT_NORMAL;
    g_print(" selected slanting for text is now Normal\n");
  }
}

/**
 * The 'Italic' item has been toggled in the Style-menu.
 */
```

```
static void cbActionStyleItalicToggled(GtkCheckMenuItem* item,
                                       ApplicationState* app) {

  g_assert(app != NULL);

  g_print("cbActionStyleItalicToggled invoked\n");
  if (gtk_check_menu_item_get_active(item)) {
    app->styleSlant = STYLE_SLANT_ITALIC;
    g_print(" selected slanting for text is now Italic\n");
  }
}

/**
 * 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;
}

/**
 * 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;
}

/**
 * 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;
}

static void cbActionColorChanged(HildonColorButton* colorButton,
                                 ApplicationState* app) {

  /* Local variables that we'll need to handle the change. */
  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. */
  hildon_color_button_get_color(colorButton, &newColor);
  /* Just an alias to save some typing (could also use
     app->currentColor). */
  curColor = &app->currentColor;

  /* Check whether the color really changed. */
  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. */
  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;
}

/**
 * Toggle the visibility of the Find toolbar.
 */
static void cbActionFindToolbarToggle(GtkWidget* widget,
                                      ApplicationState* app) {

  /* Local flag to detect whether the find toolbar should be shown
     or hidden (modified below). */
  gboolean newVisibilityState = FALSE;
```

```
  g_assert(app != NULL);
  /* See below for the explanation (next function). */
  g_assert(GTK_IS_TOOLBAR(app->findToolbar));

  g_print("cbActionFindToolbarToggle invoked\n");

  /* Toggle visibility variable first.
     NOTE:
       With the NOT-operator TRUE becomes FALSE and FALSE becomes
       TRUE. */
  newVisibilityState = ~app->findToolbarIsVisible;

  if (newVisibilityState) {
    g_print(" showing find-toolbar\n");
    /* We could also toggle visibility of all child widgets but this
       is unnecessary since they will only be seen if all of their
       parents are seen. */
    gtk_widget_show(app->findToolbar);
  } else {
    g_print(" hiding find-toolbar\n");
    gtk_widget_hide(app->findToolbar);
  }
  /* Store the new state of the visibility flag back into the
     application state. */
  app->findToolbarIsVisible = newVisibilityState;
}

/**
 * Toggle the visibility of the main toolbar, based on check menu
 * item.
 */
static void cbActionMainToolbarToggle(GtkCheckMenuItem* item,
                                      ApplicationState* app) {

  gboolean newVisibilityState = FALSE;

  g_assert(app != NULL);
  /* Since someone might initialize the application state
     incorrectly, check that we'll find a GTK+ widget that is a
     GtkToolbar (or any widget that is a subclass of GtkToolbar.

     We can stop the program in many ways:
     - We could use a standard assert(stmt). It would terminate the
       program.
     - We could use code like this:
       g_assert(GTK_IS_TOOLBAR(app->findToolbar))
       This will also abort the program (and dump core if your ulimit
       has been setup to allow this).
     - A somewhat more restrained approach would be:
       g_return_if_fail(GTK_IS_TOOLBAR(app->findToolbar));
       This is used quite a lot inside GTK+ code. The message is not
       an "ERROR", but instead "CRITICAL".
       The function will cause your function to return after
       displaying the error message (but does not terminate the
       program).
     - It would be useful for the user to know the reason for a
       problem (not just the assertion message, although that will
       contain the source file name and line number where it fails).
       This is what we'll do and also terminate the program.

    Note that this is the only place where we add such niceties.
    Normally we'll be only using g_assert to terminate the program in
```

```
    critical sections. */
  if (!GTK_IS_TOOLBAR(app->mainToolbar)) {
    /* Print a warning. */
    g_warning(G_STRLOC ": You need to have a GtkToolbar in "
                       "application state first!");
    /* Then terminate. Not very elegant, but this is an example. */
    g_assert(GTK_IS_TOOLBAR(app->mainToolbar));
  }

  /* One could argue that this should be displayed on function entry.
     However, if the asserts will fail, user/debugger will see what
     was the filename and source code file line number where the
     problem was located. This is just extra (for tracing). */
  g_print("cbActionMainToolbarToggle invoked\n");

  newVisibilityState = gtk_check_menu_item_get_active(item);

  /* If the visibility state has changed, act on it. */
  if (app->mainToolbarIsVisible != newVisibilityState) {
    if (newVisibilityState) {
      g_print(" showing main toolbar\n");
      gtk_widget_show(app->mainToolbar);
    } else {
      g_print(" hiding main toolbar\n");
      gtk_widget_hide(app->mainToolbar);
    }
    app->mainToolbarIsVisible = newVisibilityState;
  }
}

/**
 * Handles the search function from Hildon Find toolbar.
 */
static void cbActionFindToolbarSearch(HildonFindToolbar* fToolbar,
                                      ApplicationState* app) {

  gchar* findText = NULL;

  g_assert(app != NULL);

  g_print("cbActionFindToolbarSearch invoked\n");

  /* This is one of the oddities in Hildon widgets. There is no
     accessor function for this at all (not in the headers at
     least). */
  g_object_get(G_OBJECT(fToolbar), "prefix", &findText, NULL);
  if (findText != NULL) {
    /* The above test should never fail. An empty search text should
       return a string with zero characters (a character buffer
       consisting only of binary zero). */
    g_print(" would search for '%s' if would know how to\n",
            findText);
  }
}

/**
 * This will be called when the user closes the Find toolbar. We'll
 * hide it and store the new visibility state.
 */
static void cbActionFindToolbarClosed(HildonFindToolbar* fToolbar,
                                      ApplicationState* app) {
```

```
  g_assert(app != NULL);

  g_print("cbActionFindToolbarClosed invoked\n");
  g_print(" hiding search toolbar\n");

  /* It's enough to hide the toolbar and set it's visibility status.
     It is not necessary to use hide_all (the find toolbar will be
     faster to restore back to visibility). */
  gtk_widget_hide(GTK_WIDGET(fToolbar));
  app->findToolbarIsVisible = FALSE;
}

/**
 * Switch the application into fullscreen mode. Called from a menu
 * item "fullscreen-toggle". While in fullscreen, the application
 * menu will not be shown. It would be probably a good idea to
 * implement a toolbar button that can toggle fullscreen mode as
 * well (it would connect the "clicked" signal to this callback
 * function as well).
 */
static void cbActionGoFullscreen(GtkMenuItem* mi,
                                 ApplicationState* app) {

  g_assert(app != NULL);

  g_print("cbActionGoFullscreen invoked. Going fullscreen.\n");
  gtk_window_fullscreen(GTK_WINDOW(app->window));
  /* Also set the flag in application state. */
  app->fullScreen = TRUE;
}

/**
 * Handle hardware key presses. Currently will switch into and out of
 * fullscreen mode if the "fullscreen" button is pressed (F6 in the
 * SDK).
 *
 * As the keypresses come from outside GTK+ (and even GDK), this
 * needs to be an event handler.
 */
static gboolean cbKeyPressed(GtkWidget* widget, GdkEventKey* ev,
                             ApplicationState* app) {

  g_assert(app != NULL);

  g_print("cbKeyPress invoked\n");

  /* We use a switch statement here is so that you can extend this
     code easily to handle other key presses. Please see the maemo
     tutorial for a list of defines that map to the Internet Tablet
     hardware keys. */
  switch(ev->keyval) {
    case HILDON_HARDKEY_FULLSCREEN:
      g_print(" Fullscreen hw-button pressed (or F6 in SDK)\n");

      /* Toggle fullscreen mode. */
      if (app->fullScreen) {
        gtk_window_unfullscreen(GTK_WINDOW(app->window));
        app->fullScreen = FALSE;
      } else {
        gtk_window_fullscreen(GTK_WINDOW(app->window));
        app->fullScreen = TRUE;
      }
```

```
      /* We want to handle only the keys that we recognize. For this
         reason we return TRUE at this point and return FALSE for any
         other key. This will signal GTK+ that the event wasn't
         processed and it can decide what to do with it. */
      return TRUE;
    default:
      g_print(" not Fullscreen-key/F6 (something else)\n");
  }
  /* We didn't process the event. */
  return FALSE;
}

/**
 * Utility function that will create the toolbar for us.
 *
 * Parameters:
 * - ApplicationState: used to do signal connection and to set the
 *                     initial visibility status.
 *
 * Returns:
 * - New toolbar suitable to be added to a container.
 */
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. */
  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);

  toolbar = GTK_TOOLBAR(gtk_toolbar_new());

  gtk_toolbar_insert(toolbar, tbOpen, -1);
  gtk_toolbar_insert(toolbar, tbSave, -1);
```

```
  gtk_toolbar_insert(toolbar, tbSep,  -1);
  gtk_toolbar_insert(toolbar, tbFind, -1);
  gtk_toolbar_insert(toolbar, tbColorButton, -1);

  /* Setup visibility according to application state.

     We first "show" everything, then hide the top level if it's
     supposed to be hidden. This won't cause any problems, since
     GTK+ will not update the screen until we leave this callback
     function (we're not forcing a screen update here). */
  gtk_widget_show_all(GTK_WIDGET(toolbar));
  if (!app->mainToolbarIsVisible) {
    /* Hide the top level since toolbar is supposed to be invisible
       if the above test succeeds. */
    gtk_widget_hide(GTK_WIDGET(toolbar));
  }

  g_signal_connect(G_OBJECT(tbOpen), "clicked",
                   G_CALLBACK(cbActionOpen), app);
  g_signal_connect(G_OBJECT(tbSave), "clicked",
                   G_CALLBACK(cbActionSave), app);
  g_signal_connect(G_OBJECT(tbFind), "clicked",
                   G_CALLBACK(cbActionFindToolbarToggle), app);
  g_signal_connect(G_OBJECT(colorButton), "clicked",
                   G_CALLBACK(cbActionColorChanged), app);

  /* Return the toolbar as a GtkWidget*. */
  return GTK_WIDGET(toolbar);
}

/**
 * Utility to create the Find toolbar (connects the signals).
 *
 * Parameters:
 * - ApplicationState: used to connect signals and set up initial
 *                     visibility.
 *
 * Returns:
 * - New FindToolbar which can be used immediately (returned as
 *   GtkWidget*).
 */
static GtkWidget* buildFindToolbar(ApplicationState* app) {

  GtkWidget* findToolbar = NULL;

  g_assert(app != NULL);

  /* The text parameter will be displayed before the search
     text input box (Label for the search field). */
  findToolbar = hildon_find_toolbar_new("Find ");

  /* Connect the two signals that the Find toolbar can emit. */
  g_signal_connect(G_OBJECT(findToolbar), "search",
                   G_CALLBACK(cbActionFindToolbarSearch), app);
  g_signal_connect(G_OBJECT(findToolbar), "close",
                   G_CALLBACK(cbActionFindToolbarClosed), app);

  /* Setup the visibility according to the current application state.
     Uses the same logic as for the main toolbar (above). */
  gtk_widget_show_all(findToolbar);
  if (!app->findToolbarIsVisible) {
    gtk_widget_hide(findToolbar);
```

```
  }

  return findToolbar;
}

/**
 * Create the submenu for style selection.
 *
 * Parameters:
 * - ApplicationState: used to do signal connection and to set the
 *                     initial state of radio/check items.
 *
 * Returns:
 * - New submenu ready to use.
 */
static GtkWidget* buildSubMenu(ApplicationState* app) {

  GtkWidget* subMenu = NULL;
  GtkWidget* mciUnderline = NULL;
  GtkWidget* miSep = NULL;
  GtkWidget* mriNormal = NULL;
  GtkWidget* mriItalic = NULL;

  g_assert(app != NULL);

  mciUnderline = gtk_check_menu_item_new_with_label("Underline");
  gtk_check_menu_item_set_active(GTK_CHECK_MENU_ITEM(mciUnderline),
                                 app->styleUseUnderline);

  {
    GSList* group = NULL;

    mriItalic = gtk_radio_menu_item_new_with_label(NULL, "Italic");
    group = gtk_radio_menu_item_get_group(
                                   GTK_RADIO_MENU_ITEM(mriItalic));
    mriNormal = gtk_radio_menu_item_new_with_label(group, "Normal");
  }

  if (app->styleSlant == STYLE_SLANT_NORMAL) {
    gtk_check_menu_item_set_active(GTK_CHECK_MENU_ITEM(mriNormal),
                                   TRUE);
  } else {
    gtk_check_menu_item_set_active(GTK_CHECK_MENU_ITEM(mriItalic),
                                   TRUE);
  }

  miSep = gtk_separator_menu_item_new();

  subMenu = gtk_menu_new();

  gtk_menu_shell_append(GTK_MENU_SHELL(subMenu), mciUnderline);
  gtk_menu_shell_append(GTK_MENU_SHELL(subMenu), miSep);
  gtk_menu_shell_append(GTK_MENU_SHELL(subMenu), mriNormal);
  gtk_menu_shell_append(GTK_MENU_SHELL(subMenu), mriItalic);

  g_signal_connect(G_OBJECT(mciUnderline), "toggled",
    G_CALLBACK(cbActionUnderlineToggled), app);
  g_signal_connect(G_OBJECT(mriNormal), "toggled",
    G_CALLBACK(cbActionStyleNormalToggled), app);
  g_signal_connect(G_OBJECT(mriItalic), "toggled",
    G_CALLBACK(cbActionStyleItalicToggled), app);
```

```
  return subMenu;
}

/**
 * Create the menus (top-level and one sub-menu) and attach to the
 * HildonProgram.
 *
 * Parameters:
 * - ApplicationState: bound as signal parameter and also used to
 *   determine initial state of the "Show toolbar" check item.
 *
 * Returns:
 * void (will attach to the HildonProgram directly).
 */
static void buildMenu(ApplicationState* app) {

  GtkMenu*   menu = NULL;
  GtkWidget* miOpen = NULL;
  GtkWidget* miSave = NULL;
  GtkWidget* miSep1 = NULL;
  GtkWidget* miStyle = NULL;
  GtkWidget* subMenu = NULL;
  GtkWidget* mciShowToolbar = NULL;
  GtkWidget* miFullscreen = NULL;
  GtkWidget* miSep2 = NULL;
  GtkWidget* miQuit = NULL;

  miOpen = gtk_menu_item_new_with_label("Open");
  miSave = gtk_menu_item_new_with_label("Save");
  miSep1 = gtk_separator_menu_item_new();
  miStyle = gtk_menu_item_new_with_label("Style");
  mciShowToolbar =
    gtk_check_menu_item_new_with_label("Show toolbar");
  miFullscreen = gtk_menu_item_new_with_label("Fullscreen");
  miSep2 = gtk_separator_menu_item_new();
  miQuit = gtk_menu_item_new_with_label("Quit");

  /* Set the initial state of check item according to visibility
     setting of the main toolbar (from appstate). */
  gtk_check_menu_item_set_active(GTK_CHECK_MENU_ITEM(mciShowToolbar),
                                 app->mainToolbarIsVisible);

  subMenu = buildSubMenu(app);
  gtk_menu_item_set_submenu(GTK_MENU_ITEM(miStyle), subMenu);

  menu = GTK_MENU(gtk_menu_new());

  hildon_program_set_common_menu(app->program, menu);

  gtk_container_add(GTK_CONTAINER(menu), miOpen);
  gtk_container_add(GTK_CONTAINER(menu), miSave);
  gtk_container_add(GTK_CONTAINER(menu), miSep1);
  gtk_container_add(GTK_CONTAINER(menu), miStyle);
  gtk_container_add(GTK_CONTAINER(menu), mciShowToolbar);
  gtk_container_add(GTK_CONTAINER(menu), miFullscreen);
  gtk_container_add(GTK_CONTAINER(menu), miSep2);
  gtk_container_add(GTK_CONTAINER(menu), miQuit);

  g_signal_connect(G_OBJECT(miOpen), "activate",
                   G_CALLBACK(cbActionOpen), app);
  g_signal_connect(G_OBJECT(miSave), "activate",
                   G_CALLBACK(cbActionSave), app);
```

```
  g_signal_connect(G_OBJECT(miQuit), "activate",
                   G_CALLBACK(cbActionQuit), app);
  g_signal_connect(G_OBJECT(mciShowToolbar), "toggled",
                   G_CALLBACK(cbActionMainToolbarToggle), app);
  g_signal_connect(G_OBJECT(miFullscreen), "activate",
                   G_CALLBACK(cbActionGoFullscreen), app);

  /* Make all menu elements visible. */
  gtk_widget_show_all(GTK_WIDGET(menu));
}

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

  /* Allocate the application state on stack of main and initialize
     it to zero. This will also cause all the pointers to be set to
     NULL. */
  ApplicationState aState = {};

  GtkWidget* label = NULL;
  GtkWidget* vbox = NULL;
  /* We'll need temporary access to the toolbars. */
  GtkWidget* mainToolbar = NULL;
  GtkWidget* findToolbar = NULL;

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

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

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

  /* Create the Hildon program. */
  aState.program = HILDON_PROGRAM(hildon_program_get_instance());
  /* Set the application title using an accessor function. */
  g_set_application_name("Hello Hildon!");
  /* Create a window that will handle our layout and menu. */
  aState.window = HILDON_WINDOW(hildon_window_new());
  /* Bind the HildonWindow to HildonProgram. */
  hildon_program_add_window(aState.program,
                            HILDON_WINDOW(aState.window));

  /* Create a LibOSSO context (which will also attach this
     application to the D-Bus.
     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");

  /* Create the label widget, with Pango marked up content. */
```

```
label = gtk_label_new("<b>Hello</b> <i>Hildon</i> "
                      "(with LibOSSO!)");

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

/* Tell the GtkLabel widget to support the Pango markup. */
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. */
aState.textLabel = label;

/* Build the menu */
buildMenu(&aState);

/* Create a layout box for the window. */
vbox = gtk_vbox_new(FALSE, 0);

/* Add the vbox as a child to the Window. */
gtk_container_add(GTK_CONTAINER(aState.window), vbox);

/* Pack the label into the VBox. */
gtk_box_pack_end(GTK_BOX(vbox), label, TRUE, TRUE, 0);

/* Create the main toolbar. */
mainToolbar = buildToolbar(&aState);
/* Create the Find toolbar. */
findToolbar = buildFindToolbar(&aState);

/* NOTE:
     If you want to test how the error handling inside
     cbActionMainToolbarToggled works, comment the following code
     lines. */
aState.mainToolbar = mainToolbar;
aState.findToolbar = findToolbar;

/* Connect the termination signals. The application state is given
   as the user data parameter to the callback registration. This
   makes it possible for the callbacks to access the application
   state structure. */
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 for keypresses inside GTK+.
   NOTE:
     A key-event handler connected to the top-level widget will get
     all the keypresses first. If you want to pass the event deeper
     into the widget hierarchy, you'll need to tell (inside your
     callback function) that you didn't handle it. This is a
     different model from most other graphical toolkits. */
```

```
  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). */
  osso_deinitialize(ctx);

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

Listing 1.7: Contents of full.c (hhwX.c with most comments left in)