

Maemo Diablo Application Development  
Training Material  
for maemo 4.1

February 9, 2009

# Contents

<b>1</b>	<b>Introduction</b>	<b>6</b>
1.1	Introduction to Maemo Application Development . . . . .	6
<b>2</b>	<b>GTK+ Basics</b>	<b>7</b>
2.1	GLib basics . . . . .	7
2.2	Signalling mechanism . . . . .	8
2.3	GTK+ terminology . . . . .	10
2.4	Hello World learns to terminate itself . . . . .	10
2.5	GObject interface . . . . .	14
2.6	Adding menus and layout . . . . .	17
2.7	Hildon widgets . . . . .	23
2.8	Using accessors . . . . .	29
2.9	Handling dynamic memory . . . . .	33
2.10	Avoiding deprecated functions . . . . .	33
<b>3</b>	<b>GNU Make and makefiles</b>	<b>35</b>
3.1	What is GNU Make? . . . . .	35
3.2	How does make work? . . . . .	36
3.3	The simplest real example . . . . .	36
3.4	Anatomy of a makefile . . . . .	39
3.5	The default goal . . . . .	40
3.6	On names of makefiles . . . . .	41
3.7	Questions . . . . .	41
3.8	Adding make goals . . . . .	41
3.9	Making a target at a time . . . . .	43
3.10	PHONY-keyword . . . . .	43
3.11	Specifying the default goal . . . . .	44
3.12	Other common phony goals . . . . .	44
3.13	Variables in makefiles . . . . .	45
3.14	Variable flavors . . . . .	45
3.15	Recursive variables . . . . .	45
3.16	Simple variables . . . . .	46
3.17	Automatic variables . . . . .	48
3.18	Integrating with pkg-config . . . . .	49

<b>4</b>	<b>More Widgets</b>	<b>51</b>
4.1	Using menus in Hildon . . . . .	51
4.2	Adding toolbars . . . . .	57
4.3	Designing Application State . . . . .	60
4.4	Processing key events . . . . .	72
4.5	Adding File-dialogs . . . . .	75
4.6	Where to go next? . . . . .	78
4.7	Conclusions . . . . .	79
<b>5</b>	<b>Support Libraries</b>	<b>80</b>
5.1	Doing File I/O . . . . .	80
5.2	GnomeVFS . . . . .	80
5.3	Storing user preferences . . . . .	88
5.4	GConf basics . . . . .	88
5.5	Using GConf . . . . .	89
5.6	Using GConf to read and write preferences . . . . .	91
<b>6</b>	<b>GNU Autotools</b>	<b>98</b>
6.1	Introduction to GNU autotools . . . . .	98
6.2	Brief history of managing portability . . . . .	98
6.3	GNU autoconf . . . . .	99
6.4	Substitutions . . . . .	103
6.5	Introducing automake . . . . .	105
6.6	Checking for distribution sanity . . . . .	110
6.7	Cleaning up . . . . .	111
6.8	Integration with pkg-config . . . . .	111
<b>7</b>	<b>Integration with the Application Framework</b>	<b>113</b>
7.1	Integrating into AF . . . . .	113
7.2	The desktop file . . . . .	113
7.3	The service file . . . . .	114
7.4	Application support . . . . .	114
7.5	Autotools support for the service and desktop file . . . . .	117
7.6	Testing . . . . .	117
<b>8</b>	<b>Packaging Applications</b>	<b>119</b>
8.1	Creating Debian packages . . . . .	119
8.2	Packaging Basics . . . . .	119
8.3	Dependencies . . . . .	120
8.4	Packaging infrastructure . . . . .	120
8.5	Debian packages . . . . .	122
8.6	Installation process . . . . .	124
8.7	Package relationships . . . . .	124
8.8	Package control file (aka Debian control file) . . . . .	125
8.9	A maemo example control file . . . . .	126
8.10	Creating your package . . . . .	127
8.11	Adding debugging support for a package . . . . .	134
8.12	Building the package for a device . . . . .	134
8.13	Installing packages into an Internet Tablet . . . . .	135

<b>A</b>	<b>The Final Program</b>	<b>137</b>
A.1	Appendix contents	137
A.2	Autoconfigure driver	137
A.3	Automake configuration	138
A.4	Desktop file template for AF	138
A.5	Service file template for AF	138
A.6	Development bootstrap (autogen)	139
A.7	Development cleanup (antigen)	139
A.8	Program listing	140

# Preface

## Legal notice

Copyright ©2007-2009 Nokia Corporation. All rights reserved.

Nokia and maemo are trademarks or registered trademarks of Nokia Corporation. Other product and company names mentioned herein may be trademarks or trade names of their respective owners.

## Disclaimer

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

## Licenses



This training material is licensed under a Creative Commons Attribution-Share Alike 3.0 License.

The code examples copyrighted by Nokia Corporation that are included to this training material are licensed to you under following MIT-style License:

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

# Chapter 1

## Introduction

### 1.1 Introduction to Maemo Application Development

Developing graphical applications using the maemo(tm) SDK can seem like a daunting task at first, especially if you haven't used GTK+ with C before. Automating software building processes and creating Debian packages for software distribution can also seem difficult at first. This material aims to provide you with a smooth incremental path through all of the libraries and concepts that you'll need for effective GUI application development using the maemo SDK. It will also cover other common libraries and conventions that are useful when developing graphical applications.

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

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

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

## Chapter 2

# GTK+ Basics

### 2.1 GLib basics

All GTK+ programs (and Hildon programs) use the GLib utility library. This library provides a portable set of types for programs written in the C language as well as a lot of utility functions. Together they make writing portable software much easier and reduce the need to re-invent the wheel for each program and developer.

We'll cover the basic GLib types first since they will be used instead of the "standard" ones from now on:

<code>gboolean</code>	Either TRUE or FALSE. FALSE is equal to zero
<code>gint8</code>	8-bit signed integer
<code>gint16</code>	16-bit signed integer
<code>gint32</code>	32-bit signed integer
<code>gint64</code>	64-bit signed integer (there are no >64-bit ones)
<code>gpointer</code>	Untyped pointer ('void *') (32/64-bit)
<code>gconstpointer</code>	R/O untyped pointer('const void *') (32/64-bit)
<code>gchar</code>	Compiler's 'char' (8-bit in gcc)
<code>guchar</code>	Compiler's 'unsigned char'
<code>gshort</code>	Compiler's 'short' (16-bit in gcc)
<code>gushort</code>	Compiler's 'unsigned short'
<code>gint</code>	Compiler's 'int' (32-bit normally in gcc)
<code>guint</code>	Compiler's 'unsigned int'
<code>glong</code>	Compiler's 'long' (32/64-bit in gcc)
<code>gulong</code>	Compiler's 'unsigned long' (32/64-bit in gcc)
<code>gfloat</code>	Compiler's 'float' (32-bits in gcc)
<code>gdouble</code>	Compiler's 'double' (64/80/81-bits in gcc)

GLib type names and their meaning

All signed integers are stored in the 2s complement system. For each signed integer type there is also an equal size unsigned variant. Add u-character after the g (guint64 is a 64-bit wide unsigned integer).

You should try to stick to the types without the number in their names. The only exception is when you want to optimise memory structure layouts or your data comes from an external source with some inherent structure (raw bitmap images, network protocols, hardware programming).

GLib also provides a lot of macros that contain the maximum values for the above types as well as byte ordering and order changing macros which can be very useful when writing portable software. These are all covered in detail in



the GLib API documentation, which you should install on your development Linux, but can also be read at [maemo.org](http://maemo.org).

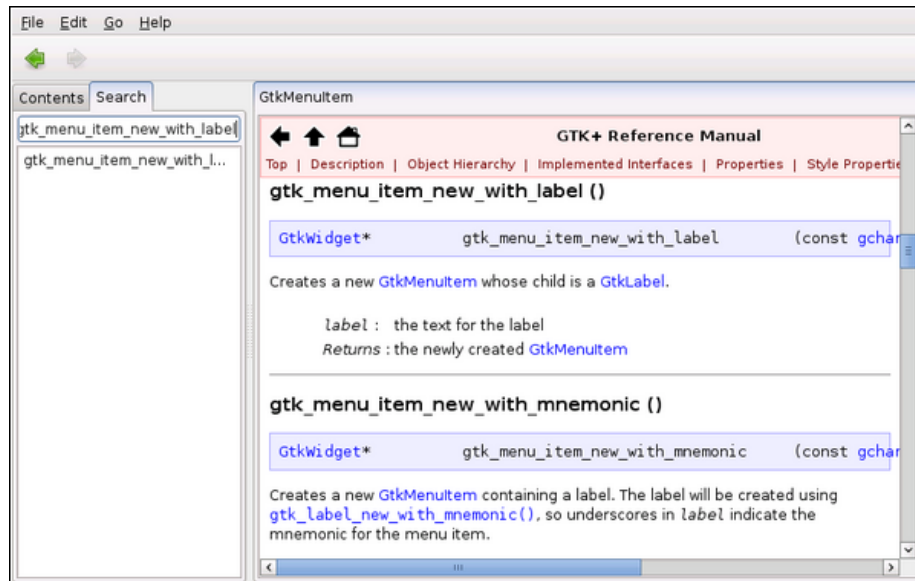
We will concentrate on the bare minimum features to get our applications running and packaged, but with time, you will learn to appreciate all the code that GLib contains (some people depreciate GLib for the same reason, since GLib is quite large for an utility library). Since GLib is a shared library and an Internet Tablet will always have at least one graphical application running, using GLib won't normally cause extra memory use.

Besides the types, GLib also offers the following functionality that you might find useful:

- Memory allocation
- Message output, debugging and logging functions
- String processing (UTF-8 and UCS-4)
- Date and time processing and counting timers
- Data structures:
  - Dynamic strings
  - Linked lists (single and double)
  - Hash tables
  - Dynamic arrays (also with pointers)
  - Binary and N-ary trees
  - Caching support
- And other miscellany

As was mentioned before, the API documentation for GLib can be browsed online at [maemo.org](http://maemo.org). For other libraries that are available in maemo you should use the master API reference index at [maemo.org](http://maemo.org)

If you have the suitable documentation packages installed on your development system, you should also try [devhelp](#), a nice application that allows you to browse through the GNOME library documentation and provides hypertext and search facilities. Please be careful about difference in version numbers between the libraries installed on your SDK and the ones that the distribution provides in the devhelp packages.



## 2.2 Signalling mechanism

In order to handle interaction between components (graphical or otherwise), different libraries have invented various ways of implementing event and change notifications. Each of them have a bit different design and restrictions. The design model used in GTK+ is based on registration/auto-delivery with callbacks as the delivery method.

The mechanism is called GSignal, and it is implemented in a library called GObject. GObject is a framework and infrastructure library for the C language and is used to implement object-oriented constructs similar to the ones used in Java and C++. This is done in a portable way and without special pre-processing step necessary for source files. This is in contrast to some other similar solutions used in libraries. Since no special tools are required, GObject is very portable. It also uses the GLib exclusively for data structure implementations and memory management.

Using GObject (implementing your own classes, extending existing classes and so on) is not covered in this material as this subject is quite complicated and also unnecessary when starting GTK+ programming.

GTK+ widgets (and widgets in Hildon) use the GSignal mechanism to implement notification "messages" that come from widgets when something happens (normally some action caused by HID-event). These messages are then "passed" to interested listeners by means of callback functions. There is no shared event bus (like in Windows GDI) but rather you can think of a system where you can connect as many "links" between widgets and callback functions as you like. GSignal also offers a rich mechanism to specify at which point of signal delivery to invoke some specified callback (so that the handlers can be prioritised), but we'll settle for the default priority since it's enough for normal GUI-programs.

Each class defines what kind of signals an instance of that class can emit (send). Signals are identified by a text string (which allows dynamic introspection) and the signal source (either an object that has something interesting to notify, or some core system component). We "connect a signal" by means of specifying the object that is capable of emitting the signal and giving the name of signal. You can think of the name as type, but it's just used to differentiate between different signals, like "clicked" and "selected". Since we're connecting, we obviously need some target for this connection. This target is a function implemented in C language which will be called when the signal is emitted. Note that the object that emits the signal does not call the callback function directly, but rather uses the generic signal delivery framework that GObject provides. This allows simple API interfaces for both the emitter and the receiver as well as for connecting the signals.

Given that C language was designed before object-oriented programming had matured, GObject and signals are quite elegant. However, if you only have programmed using object-oriented languages, GObject will seem strange to you (at start, since the concepts are the same, just the syntax is different). You will also have to type more when writing programs.

By the way, GObject documentation calls signals as "A means of customisation of object behaviour and a general purpose notification mechanism". Basically it means that signals can be used outside GTK+ and sometimes are.

You can learn more about the GObject by either reading the API or reading a fairly extensive book which spends a lot of time explaining the GObject-system [The Official Gnome 2 Developer's Guide, by Matthias Werkus, published in 2004 by No Starch Press]. Learning from existing code base is probably not always a good idea since the system is quite complex.

## 2.3 GTK+ terminology

Before jumping into GTK+, we need to cover some GTK+ terminology:

**Widget** Is some code that draws on screen and allows the user to interact with itself (there are exceptions to both rules). Examples: Scrollbars, Buttons, Menus and so on.

**Container** A special kind of Widget (that might be invisible) that allocates space for other widgets and groups them together on screen. A Container can contain another container. Examples: Window, VBox, HBox, Toolbar.

**Packing** The action that is done by a container when a widget is added to it. Normally we'd say that "we pack this widget into a container". Packing will execute space allocation and layout code, but if the widget is not currently visible, nothing user perceivable will happen.

**Child Widget** A widget that is packed into a Container. Conversely, a Widget has a parent which is always a container. A widget cannot have more than one parent.

**Widget Tree** All widgets and containers that are children to one root widget (which is a Container by necessity). There is exactly one widget tree for each Window (even if the Window is not visible).

**Event** An notification from outside the process that something has happened. Normally these come from the HID-system and are transformed into signals by GTK+. Normal signal handling mechanisms are used to process them.

**Visibility** Each widget can be told to be invisible or visible. Only if the parent container is visible, will this cause any user perceived graphics on screen. To see all widgets in one widget tree we need to tell all the widgets in the tree to "show" themselves. Normally we do this after we've created the widget tree completely so that the user will not see unnecessary screen redrawing.

**Property** This is actually related to GObject, but is used with GTK+ widgets as well. Generally speaking a property is some named data or value that can be set and retrieved. When setting the property, the object can execute some code to reflect the change of the value. Reading (getting) a property will also normally trigger some code in the widget (class) implementation.

Most graphics toolkits contain similar elements and most toolkits for UNIX (X11) also use similar terminology.

## 2.4 Hello World learns to terminate itself

These concepts can now be combined by extending our Hello World program to terminate itself in a controlled fashion when the window manager tells the program to close its window.

```
/**
 * gtk_helloworld-2.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.
 *
 * This version adds proper mechanisms to end the program.
 *
 * Look for lines with "NEW" or "MODIFIED" in them.
 */

#include <stdlib.h>
#include <gtk/gtk.h>

/**
 * NEW
 *
 * Callback function (event handler) for the "delete" event.
 * This event is emitted by GTK+ main loop after a window manager has
 * requested this window to be closed.
 *
 * NOTES:
 * - This is really a signal handler.
 * - The first parameter to a signal handler is a pointer to the
 *   object that caused the signal to be emitted (when using
 *   gobject_signal_connect).
 * - There are zero or more parameters in between the first one and
```

```

* the last one. The count of parameters depends on the signal
* type.
* - The last parameter is always an untyped pointer (again, when
* using gobject_signal_connect).
*
* Parameters:
* - widget: Pointer to the widget that emitted the delete-event.
* - event: GdkEvent-structure describing what kind of event this
* is. We'll register this handler to handle only the
* "destroy-event", so we'll know what kind of event this
* is without checking the "event"-parameter.
* - data: Untyped pointer for passing data between the main loop
* and handlers. We ignore it for now.
*
* Returns:
* - gboolean: Whether the event has been consumed by this handler or
* not.
* NOTE:
* Signal handlers may exit with an return value. Whether this
* value will be used by anyone, depends on the signal type. For
* events, we must always return a boolean value.
*/
static gboolean delete_event(GtkWidget* widget, GdkEvent event,
                             gpointer data) {
    /* Print out an informational message. */
    g_print("CB:delete_event\n");

    /* We want to tell the gtk_main that we're done with this widget.
    Since it's the top-level widget in our window (it is the
    window), returning FALSE to our caller signifies that the event
    has not been processed by us, and this in turn means that it
    will be propagated to the default window delete-event handler
    which will terminate the program. This is done in by the caller
    so that it will issue a "destroy"-signal next. If we would
    return TRUE, the program wouldn't stop. This behavior is
    specific to the "delete-event" signal. */
    return FALSE;
}

/**
* NEW
*
* Our callback function for the "destroy"-signal which is issued
* when the Widget is going to be destroyed.
* NOTE:
* In this case we don't need to return anything, since we're just
* processing a simple signal.
*
* This is used by widgets that are going away that want to notify us
* that they will not be around for much longer. They will remove
* themselves eventually.
*
* We call gtk_main_quit() which is a routine that will tell gtk_main
* that it's time to terminate.
*
* Note, you must not terminate your program here! (do not use the
* exit()-library function). This would leave the GTK+ library in a
* slightly confused state and might leave other threads hanging
* around. In this material we won't be using threads, but consider
* yourself warned.
*
* Parameters:

```

```

* - widget: Same as with event handler callbacks.
* - data:   Same as with event handler callbacks.
*
* Returns:
* void
*/
static void end_program(GtkWidget* widget, gpointer data) {
    /* Print out an informational message to stdout. */
    g_print("CB:end_program: calling gtk_main_quit()\n");

    /* Tell gtk_main to terminate the application. */
    gtk_main_quit();

    /* Display message about what is happening. */
    g_print("CB:end_program: back from gtk_main_quit & returning\n");
}

/**
 * MODIFIED
 *
 * We connect the signals to the handlers so that the application may
 * be terminated properly from the GUI.
 */
int main(int argc, char** argv) {

    /* We'll have two references to two GTK+ widgets. */
    GtkWidget* window;
    GtkLabel* label;

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

    /* Create a window with window border width of 12 pixels and a
       title text. */
    window = g_object_new(GTK_TYPE_WINDOW,
        "border-width", 12,
        "title", "Hello GTK+",
        NULL);

    /* Create the label widget. */
    label = g_object_new(GTK_TYPE_LABEL,
        "label", "Hello World!",
        NULL);

    /* Pack the label into the window layout. */
    gtk_container_add(GTK_CONTAINER(window), GTK_WIDGET(label));

    /**
     * NEW
     *
     * Here we "install" event and signal handlers.
     *
     * Events are handled by the GTK+ as signals. Their difference is
     * explained in the text that follows.
     *
     * g_signal_connect(GtkWidget*, gchar*, CB*, gpointer*) is the API
     * function to connect signals from some object to some signal
     * receiver (callback function when using C).
     *
     * The second parameter should point to a string giving the event
     * or signal name. Glib treats underscores and dashes in signal
     * names as they would be the same character. The API documentation

```

```

/* uses the dash (-) and so will we. Needless to say, different
   * kind of widgets are capable of emitting different events.
   */
/* Third parameter is the address of the callback function that
   * should receive the signal. We use G_CALLBACK()-macros to force a
   * typecast into the type specification that g_signal_connect
   * expects. Depending on the signal the callback will receive
   * different number of parameters and is expected to return
   * something in some cases. There is no hard and set rule.
   */
/* THIS MEANS THAT YOU HAVE TO BE SURE ABOUT THE CAST VALIDITY.
   */
/* It is possible to pass an extra pointer (the last parameter)
   * that will be carried with the signal to the signal receiver. We
   * will see uses for this later, but in real applications it is
   * quite important. This last parameter is defined as a gpointer
   * (untyped pointer, void*), and if one needs to pass other data
   * in the pointer's place, one will need to do additional
   * typecasting (we'll see this later).
   */
g_signal_connect(window, "delete-event",
                 G_CALLBACK(delete_event), NULL);
g_signal_connect(window, "destroy",
                 G_CALLBACK(end_program), NULL);

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

/* Start the main event loop. */
g_print("main: calling gtk_main\n");
gtk_main();

/* Display a message to the standard output and exit. */
g_print("main: returned from gtk_main and exiting with success\n");

/* Return success as exit code. */
return EXIT_SUCCESS;
}

```

Listing 2.1: Hello World with basic signals (gtk\_helloworld-2.c)

It would be very inefficient to comment your code with such verbosity as has been done here (and for all source code of this material). Use comments for the first couple of simple programs you write so that you can return to them later. Documenting the sources and reasons for signals in for each handler (callback function) is recommended so that you don't have to read all of the source code to find out which event the signal comes from and what kind of a signal you have connected it to.

After building the program, it will be run, and then user will click on the "X" window close button:

```

[sbox-DIABLO_X86: ~/appdev] > gcc -Wall 'pkg-config --cflags gtk+-2.0' \
gtk_helloworld-2.c -o gtk_helloworld-2 'pkg-config --libs gtk+-2.0'
[sbox-DIABLO_X86: ~/appdev] > ./gtk_helloworld-2
main: calling gtk_main
Window closing button engaged by the user
CB:delete_event
CB:end_program: calling gtk_main_quit()
CB:end_program: back from gtk_main_quit & returning
main: returned from gtk_main and exiting with success

```

Running the program

There is no associated screenshot as graphically the application looks exactly the same as before. But in this case we're more interested in the `g_print()`ed output of the program. Notice the order and the names of the functions in the printed lines. This is the sequence one would expect to terminate a GTK+-based application.

If the output of your build or test phase is significantly different, it could be that there is some known issues in this version of the maemo training material. To verify this you can go to the following maemo community maintained training material wiki page on [wiki.maemo.org/MaemoTraining](http://wiki.maemo.org/MaemoTraining)).

## 2.5 GObject interface

We still have some functions to cover, so let's go through the basic functionality that any GObject class must implement and hence the functionality that you can use with any widget in GTK+.

We'll start with an example using the accessor interface and then compare that with the same functionality implemented using the property-interface.

Create a window with window border width of 12 pixels and title of Hello GTK+:

```
window = (GtkWindow*)gtk_window_new(GTK_WINDOW_TOPLEVEL);
gtk_window_set_title(window, "Hello GTK+");
gtk_container_set_border_width((GtkContainer*)window, 12);
```

Listing 2.2: Hello World with basic signals (gtk\_helloworld-2.c)

Instead of using the accessors, we can use the generic object creation framework introduced by GObject, and use class-names (GType-names really) to select which class to use for object creation.

Coupled with this, we can also pass a list of properties to set at the same time as the object is created (just before we receive the object).

This allows us to concentrate on one API instead of multitude of different creation functions and different accessor functions like in the example above. You will see code that uses both ways, but it is largely a matter of taste whether you find the new (GTK+ 2.0) way cleaner and/or more coherent.

As an example, we'll create an object of class GtkWindow by giving the GType value GTK\_TYPE\_WINDOW. We also want to set two properties to this new object, namely the "border-width" (which is inherited from GtkContainer-class and "title" which is implemented in GtkWindow (in the GTK+ API reference, all class specific signals and properties are listed on the same page with the old style accessor functions). We terminate the list of properties to set with the NULL value.

```
window = g_object_new(GTK_TYPE_WINDOW,
    "border-width", 12,
    "title", "Hello GTK+",
    NULL);
```

Listing 2.3: Hello World with basic signals (gtk\_helloworld-2.c)

We later to decide to disable window resizing by the user. For this, we need to set the "resizable"-property. Note that this is done only to illustrate the



property-based API, and does not mean that disabling window resizing is a good idea (it's not).

```
/* Set the value of one property. */  
g_object_set_property(window, "resizable", FALSE);
```

Listing 2.4: Hello World with basic signals (gtk\_helloworld-2.c)

Suppose now that we want to set multiple properties each along the class hierarchy of an object. Using accessor functions for this would be tedious.

Instead we collect the properties in a list and set them all using one function call:

```
gchar* data = "Some random data";  
  
g_object_set(window,  
  "resizable", FALSE, /* GtkWidget-property */  
  "has-focus", TRUE, /* GtkWidget-property */  
  "width", 300, /* GtkWidget-property */  
  "height", 150, /* GtkWidget-property */  
  "user-data", data, /* GObject-property */  
  NULL);
```

Listing 2.5: Hello World with basic signals (gtk\_helloworld-2.c)

To round off, we present the property-reader interface which is very similar to the setting interface, but the direction is reversed.

```
gint width = 0;  
gint height = 0;  
gboolean isVisible = FALSE;  
  
g_object_get(window,  
  "width", &width, /* GtkWidget-property */  
  "height", &height, /* GtkWidget-property */  
  "visible", &isVisible, /* GtkWidget-property */  
  NULL);
```

Listing 2.6: Hello World with basic signals (gtk\_helloworld-2.c)

Note that the API reference doesn't always list all properties that a class implements, and in some cases they're not implemented as properties, and you'll have to use old style accessor functions. This is a sad thing but hopefully it will be fixed with time. Also, sometimes there is an property but no suitable accessor function.

In the next code snippet you will notice GObject casting macros. These are used to make proper casts for functions that expect a certain type arguments.

For example, below we use `gtk_container_add()` which accepts two parameters:

- `GtkContainer*` : A Container into which to add the widget.
- `GtkWidget*` : The widget to add. Any widget will do.

Since a window is also a container, we can cast it safely into a container (along the class hierarchy). Since the `GtkLabel` widget is also a `GtkWidget` we can cast that safely into a widget.

This will keep the compiler happy. Obviously you will need to know the names of the various macros to be able to use them. All of the GTK+ widget casting macros obey the rule that each "word" is written in capital letters and there is an underscore between the "word". Thus, a `GtkWidget` becomes `GTK_WIDGET` and `GtkRadioMenuItem` becomes `GTK_RADIO_MENU_ITEM`.

Besides keeping the compiler from complaining about mismatched types, the macros may implement type checking (and instance checking against the class hierarchy).

```
/* Pack the label into the window layout. */
gtk_container_add(GTK_CONTAINER(window), GTK_WIDGET(label));
```

Listing 2.7: Hello World with basic signals (gtk\_helloworld-2.c)

We'll finish off with an example where using the property-based interface would be too tedious. Sometimes the accessor interface contains nice utility functions that take some work off your shoulders.

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

Listing 2.8: Hello World with basic signals (gtk\_helloworld-2.c)

Start with a widget-tree and set all contained widgets' "visible"-property to TRUE.

There is also `gtk_widget_show(GtkWidget*)`, `gtk_widget_hide(GtkWidget*)` and `gtk_widget_hide_all(GtkWidget*)`.

## 2.6 Adding menus and layout

We'll next test out the concepts covered so far by implement a nice menu for our application. The example includes using signals and using the property-based system.

We'll try to use the property based interface as much as possible in order to demonstrate the compactness of code that can be achieved. Note that each property lookup will involve a hash table lookup inside GObject. Finding the perfect balance between code size and speed is never easy. Smaller code tends to lead to faster programs because there is more CPU cache left to utilise and smaller code also improves cache locality. It is generally quite difficult to say which of the two approaches will end up being faster. Since the interface selection will affect code readability and teamwork, this is something that should be decided within a team or by coding standards if they exist.

```
/**
 * gtk_helloworld-3.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 a menu and callbacks to process menu selections.
 *
 * Look for lines with "NEW" or "MODIFIED" in them.
```

```

*/

#include <stdlib.h>
#include <gtk/gtk.h>

/**
 * NEW
 *
 * We will use a single callback function to process the activation
 * of all of the menu items.
 *
 * We could also use strings to identify which menu item was
 * selected, or if our callback would have a list of pointers to the
 * menu items themselves, we could compare against that. Instead,
 * we'll define an enumeration that will contain all the possible
 * menu items that our application will have. This allows use to use
 * a simple switch statement to determine what to do.
 */
typedef enum {
    MENU_FILE_OPEN = 1,
    MENU_FILE_SAVE = 2,
    MENU_FILE_QUIT = 3
} MenuActionCode;

/**
 * Callback function (event handler) for the "delete" event.
 */
static gboolean delete_event(GtkWidget* widget, GdkEvent event,
                             gpointer data) {

    /* Print out an informational message. */
    g_print("CB:delete_event\n");

    /* Done with the widget. */
    return FALSE;
}

/**
 * Our callback function for the "destroy"-signal which is issued
 * when the Widget is going to be destroyed.
 */
static void end_program(GtkWidget* widget, gpointer data) {
    /* Print out an informational message to stdout. */
    g_print("CB:end_program: calling gtk_main_quit()\n");

    /* Tell gtk_main to terminate the application. */
    gtk_main_quit();

    /* Display message about what is happening. */
    g_print("CB:end_program: back from gtk_main_quit & returning\n");
}

/**
 * NEW
 *
 * Signal handler for the menu item selections.
 *
 * Gets the menu action code as a parameter but since we'll cast it
 * into an pointer on callback registration, we'll need to cast it
 * back internally.
 */
static void cbActivation(GtkMenuItem* mi, gpointer data) {

```

```

/* Convert the pointer into an integer, irrespective of the pointer
size. */
MenuActionCode aCode = GPOINTER_TO_INT(data);

switch(aCode) {
case MENU_FILE_OPEN:
    g_print("Selected open\n");
    break;
case MENU_FILE_SAVE:
    g_print("Selected save\n");
    break;
case MENU_FILE_QUIT:
    g_print("Selected quit\n");
    gtk_main_quit();
    break;
default:
    /* Besides g_print, there are two other output functions in
    GLib: g_warning and g_error. g_error will stop our program
    (using abort() internally), so we'll choose to use g_warning
    instead since we want our program to continue running. The
    situation is not lethal, but should be investigated.

    g_warning prints out a warning message in the same format as
    g_print and printf. */
    g_warning("cbActivation: Unknown menu action code %d.\n", aCode);
}
}

/**
 * NEW
 *
 * A convenience function to create menu items with internal label
 * objects.
 *
 * NOTE:
 * Most existing programs use a convenience function that GTK+
 * provides (gtk_menu_item_new_with_label).
 *
 * We want to use the GObject API for this example to demonstrate its
 * compactness.
 *
 * Parameters:
 * - const gchar*: string to display in the label inside the item.
 *
 * Returns:
 * - A pointer to a new MenuItem that holds the label
 */
static GtkWidget* buildMenuItem(const gchar* labelText) {

    GtkWidget* label;
    GtkWidget* mi;

    /* Create the Label object first. We set the text using its "label"
    property. We also set its alignment using a property that its
    superclass (GtkMisc) has. Namely "xalign". xalign with 0.0 is
    left and 1.0 is right extreme. Default value for xalign is 0.5
    which would align the text using center justification. */
    label = g_object_new(GTK_TYPE_LABEL,
        "label", labelText, /* GtkWidget property */
        "xalign", (gfloat)0.0, /* GtkMisc property */
        NULL);

```

```

/* Next we create the GtkMenuItem.

A MenuItem is a container that can hold one child (this
restriction comes from GtkBin-class, which is an abstract class
for containers that are capable of holding only one child).
We use the "child" property of the Container class to set its
child on creation. Normally a Container would allow us to add
multiple children in a similar way (by setting the "child"-
property multiple times). */
mi = g_object_new(GTK_TYPE_MENU_ITEM, "child", label, NULL);

/* Return the GtkMenuItem to caller. */
return mi;
}

/**
 * NEW
 *
 * Utility function to build a menubar and the menu.
 *
 * Uses the buildMenuItem-function from above and will connect the
 * signals that can be generated by MenuItems when they're selected
 * (the "activate"-signal).
 *
 * Returns:
 * - the completed MenuBar
 */
static GtkMenuBar* buildMenubar(void) {

/* The visible menubar widget that will hold one submenu called
"File". */
GtkMenuBar* menubar;
/* The menu item representing the menu-name in the menubar. It is
necessary in order to attach the menu to the menubar. Will open
the submenu when activated. */
GtkMenuItem* miFile;

/* A container implementing a menu that holds multiple items. */
GtkMenu* fileMenu;
/* The individual menu items and a separator item. */
GtkMenuItem* miOpen;
GtkMenuItem* miSave;
GtkMenuItem* miSep;
GtkMenuItem* miQuit;

/* Create the container for items. */
fileMenu = g_object_new(GTK_TYPE_MENU, NULL);

/* Create the menu items. */
miOpen = buildMenuItem("Open");
miSave = buildMenuItem("Save");
miQuit = buildMenuItem("Quit");
miSep = g_object_new(GTK_TYPE_SEPARATOR_MENU_ITEM, NULL);

/* Add the items into the container.

Here we again use the "child"-property. But since Menu is a
proper container that can hold multiple children, we can use a
list of widgets to add. */
g_object_set(fileMenu,
"child", miOpen,
"child", miSave,

```

```

    "child", miSep,
    "child", miQuit,
    NULL);

/* Connect the signals from the individual menu items.
NOTE:
    We need to cast the CB data into a gpointer since that is what
    the prototype of g_signal_connect tells the compiler to
    enforce. Luckily for us there is a macro to do exactly this
    (in GLib). */
g_signal_connect(G_OBJECT(miOpen), "activate",
    G_CALLBACK(cbActivation), GINT_TO_POINTER(MENU_FILE_OPEN));
g_signal_connect(G_OBJECT(miSave), "activate",
    G_CALLBACK(cbActivation), GINT_TO_POINTER(MENU_FILE_SAVE));
g_signal_connect(G_OBJECT(miQuit), "activate",
    G_CALLBACK(cbActivation), GINT_TO_POINTER(MENU_FILE_QUIT));

/* Create the menu item that will be added to the menubar. */
miFile = buildMenuItem("File");

/* Associate the "File" menu item to the Menu container so that it
will open it as a submenu when activated.

    There is no property to set this, so we use an accessor
    function. */
gtk_menu_item_set_submenu(miFile, GTK_WIDGET(fileMenu));

/* Create the menubar. */
menubar = g_object_new(GTK_TYPE_MENU_BAR, NULL);

/* Add the "File" menu item to the menubar.

    We want to add from left to right, so again, the "child"
    property will be used. */
g_object_set(menubar, "child", miFile, NULL);

/* Return the completed menubar to the caller. */
return menubar;
}

/**
 * MODIFIED
 *
 * We add a container that can hold more than one widget since
 * GtkWidget can hold only one at a time. We need to put our menubar
 * somewhere, and display a label. We select a container that will
 * split the area that it will get into horizontal sections according
 * to the needs to its children (GtkVBox).
 */
int main(int argc, char** argv) {

    GtkWidget* window;
    GtkLabel* label;
    GtkMenuBar* menubar;
    GtkVBox* vbox;

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

    /* Create a window with window border width of 12 pixels and a
    title text. */
    window = g_object_new(GTK_TYPE_WINDOW,

```

```

    "border-width", 12,
    "title", "Hello GTK+",
    NULL);

/* Create the label widget. */
label = g_object_new(GTK_TYPE_LABEL,
    "label", "Hello World! (with menus)",
    NULL);

/* Build the menubar (NEW). */
menubar = buildMenubar();

/* Create a layout box for the window since it can only hold one
   widget (NEW). */
vbox = g_object_new(GTK_TYPE_VBOX, NULL);

/* Add the vbox as the only child for the window (NEW). */
g_object_set(window, "child", vbox, NULL);

/* NEW

Add it to the container that now controls the area inside the
window.

Both VBox and HBox implement an abstract class called GtkBox.
Adding child widgets to GtkBoxes is possible by using an
old-style API (gtk_box_pack_start and _end), or we can use the
context based one with properties.

We need to control three properties (besides the "child", GtkBox
is also a container):
- "pack-type": selects whether to add the next child to the
  start or end of box. Default is GTK_PACK_START.
  For VBox start is top and end is bottom.
- "expand": Controls whether the child will receive extra space
  when the Box will grow (if it get's new space itself). Default
  is TRUE.
- "fill": If "expand" is set, controls whether extra new space
  should be given to the child or used as padding. Default is
  TRUE.

Based on this information we now must select a sequence that will
create a split between the menubar and the label.

We want the menubar to take only the amount of space that it
requires, and for the label to fill up the rest. Should the
window grow, we want the area allocated to label to grow as well.
We'll see a growing window later on.

We have two choices (A or B):
A) Insert the menubar first:
  1) Set "expand" to FALSE.
  2) Use "child" to insert menubar.
  3) Set "pack-type" to GTK_PACK_END so that label will be added
     from bottom-up.
  4) Set "expand" back to TRUE.
  5) Use "child" to insert the label.
B) Insert the Label first:
  1) Set "pack-type" to GTK_PACK_END.
  2) Insert label using "child".
  3) Set "expand" to FALSE.
  4) Set "pack-type" to GTK_PACK_START.

```

5) Use "child" to insert the menubar.

As you can see, both approaches will require an equal amount of property operations, so we can choose either one. There are other possible orders as well, but all of them require at least the five operations. We'll choose the first approach (A):

```
g_object_set(vbox,
    "fill", FALSE,
    "child", menubar,
    "pack-type", GTK_PACK_END,
    "expand", TRUE,
    "child", label,
    NULL);
```

NOTE:

This won't work currently since the VBox implementation does not implement the properties of the abstract GtkWidget class.

This means that we'll need to use the accessor interface which requires us to give both fill and expand each time we add a widget.

The order of parameters for the accessor function:

- GtkWidget into which we're adding the widget.
- The widget that we're adding.
- "fill".
- "expand".
- "padding", which is how many pixels are used around the widget. \*/

```
gtk_box_pack_start(GTK_BOX(vbox), GTK_WIDGET(menubar), FALSE,
    FALSE, 0);
gtk_box_pack_end(GTK_BOX(vbox), GTK_WIDGET(label), TRUE, TRUE, 0);

/* Connect the termination signals */
g_signal_connect(window, "delete-event",
    G_CALLBACK(delete_event), NULL);
g_signal_connect(window, "destroy",
    G_CALLBACK(end_program), NULL);

/* Show all widgets that are contained by the window. This will
    also include the menubar, and its widget tree. */
gtk_widget_show_all(GTK_WIDGET(window));

/* Start the main event loop. */
g_print("main: calling gtk_main\n");
gtk_main();

/* Display a message to the standard output and exit. */
g_print("main: returned from gtk_main and exiting with success\n");

/* Return success as exit code. */
return EXIT_SUCCESS;
}
```

Listing 2.9: Listing of gtk\_helloworld-3.c





Figure 2.1: Without run-standalone.sh

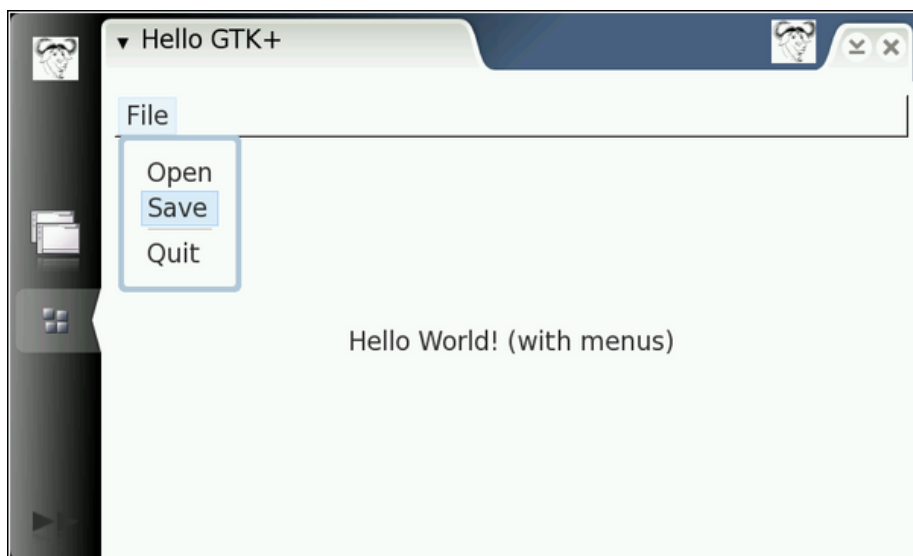


Figure 2.2: With run-standalone.sh

The application will also now quit when "Quit" is selected from the menu.

## 2.7 Hildon widgets

The platform contains a set of widgets that have been optimised for smaller devices with a stylus-like HID. The screen is quite small (physically) compared

to modern desktop TFT-displays, so some thinking needed to be done on how to switch between running GUI applications and whether a traditional window manager or graphical desktop was suitable.

In order to integrate nicely into the AF-environment, we start by switching our application model to the one that Hildon expects us to use.

Mainly, Hildon provides two widgets: `HildonProgram` and `HildonWindow` which replace some of the functionality that `GtkWindow` normally provides.

`HildonProgram` is an "super-window" object that provides a shell through which we can integrate our graphical views into the runtime environment. Views are similar to windows, but only one view can be visible at any given time. You might think of an Hildon application as being a dialog with multiple tabs, but in this case the tabs will be the only interface that the user will see. The views are implemented by using `HildonWindow`-widgets, which basically are layout Containers into which we place our widgets.

This requires some thinking when it comes to UI design, especially if you plan to port existing GUI software to use Hildon widgets.

Other points to keep in mind:

- Each view will have one `GtkMenu` that the application can use. This is in contrast to using a `MenuBar` in a regular desktop environment.
- Each Window has a container to hold a toolbar (automatically).
- Input fields may activate the virtual keyboard when they get focus. This will cause the size of the application layout area to resize. Application dialogs will also be resized if necessary. If your content is not capable of being resized easily, you might want to use a `GtkScrolledWindow` to hold it.
- Avoid deep hierarchies in Menus as the screen estate is limited.
- Design your application so that it will display only one main "Window" at any given time. Applications which normally would have multiple separate windows/dialogs all open simultaneously need to be re-engineered.
- Avoid trying to show too much information at once.

We will now modify our Hello World to use the `HildonProgram` and `HildonWindow` widgets. We'll also use the Menu that `HildonWindow` provides. The Menu starts empty and we'll add `MenuItems` into it directly, instead of creating a separate Menu widget like was done before.

```
/**
 * hildon_helloworld-1.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.
 *
 * Let's hildonize our application. Most of the application stays
 * unchanged, but the menu building utility needs some changes and
 * 'main' will need to be modified.
 *
 * Look for lines with "NEW" or "MODIFIED" in them.
```

```

*/

#include <stdlib.h>
#include <gtk/gtk.h>

/**
 * NEW
 *
 * When using widgets from the Hildon library, we need to use
 * additional include files. We could just pull in <hildon/hildon.h>,
 * but that is unnecessary since we only use a small number of Hildon
 * widgets. Instead we'll opt to pull in only the Hildon header files
 * that are necessary for the widgets that we use.
 *
 * For our first Hildon application, we'll need the Hildon Program
 * framework widgets whose API and types are declared in
 * hildon-program.h
 */
#include <hildon/hildon-program.h>

/* Menu codes. */
typedef enum {
    MENU_FILE_OPEN = 1,
    MENU_FILE_SAVE = 2,
    MENU_FILE_QUIT = 3
} MenuActionCode;

/**
 * Callback function (event handler) for the "delete" event.
 */
static gboolean delete_event(GtkWidget* widget, GdkEvent event,
                             gpointer data) {
    return FALSE;
}

/**
 * Our callback function for the "destroy"-signal which is issued
 * when the Widget is going to be destroyed.
 */
static void end_program(GtkWidget* widget, gpointer data) {
    gtk_main_quit();
}

/**
 * Signal handler for the menu item selections.
 */
static void cbActivation(GtkMenuItem* mi, gpointer data) {
    MenuActionCode aCode = GPOINTER_TO_INT(data);

    switch(aCode) {
        case MENU_FILE_OPEN:
            g_print("Selected open\n");
            break;
        case MENU_FILE_SAVE:
            g_print("Selected save\n");
            break;
        case MENU_FILE_QUIT:
            g_print("Selected quit\n");
            gtk_main_quit();
            break;
        default:

```

```

        g_warning("cbActivation: Unknown menu action code %d.\n", aCode);
    }
}

/**
 * A convenience function to create menu items with internal label
 * objects.
 */
static GtkWidget* buildMenuItem(const gchar* labelText) {

    GtkWidget* label;
    GtkWidget* mi;

    /* Create the Label object. */
    label = g_object_new(GTK_TYPE_LABEL,
        "label", labelText, /* GtkWidget property */
        "xalign", (gfloat)0.0, /* GtkWidget property */
        NULL);

    /* Create the GtkWidget and add the Label as its child. */
    mi = g_object_new(GTK_TYPE_MENU_ITEM, "child", label, NULL);

    /* Return the GtkWidget to caller. */
    return mi;
}

/**
 * MODIFIED
 *
 * This utility adds the menu items into the given HildonProgram.
 * It uses the buildMenuItem-function internally to build each menu
 * item. It will connect the signals that can be generated by
 * MenuItems when they're selected ("activate"-signal).
 *
 * Since there is no menubar, so this becomes quite simple.
 */
static void buildMenu(HildonProgram* program) {

    GtkWidget* menu;
    GtkWidget* miOpen;
    GtkWidget* miSave;
    GtkWidget* miSep;
    GtkWidget* miQuit;

    /* Create the menu items. */
    miOpen = buildMenuItem("Open");
    miSave = buildMenuItem("Save");
    miQuit = buildMenuItem("Quit");
    miSep = g_object_new(GTK_TYPE_SEPARATOR_MENU_ITEM, NULL);

    /* Create a new menu. */
    menu = g_object_new(GTK_TYPE_MENU, NULL);

    /* Add the items to the container. */
    g_object_set(menu,
        "child", miOpen,
        "child", miSave,
        "child", miSep,
        "child", miQuit,
        NULL);

    /* Set the top-level menu for Hildon program (NEW). */

```

```

hildon_program_set_common_menu(program, menu);

/* Connect the signals from the individual menu items. */
g_signal_connect(G_OBJECT(miOpen), "activate",
    G_CALLBACK(cbActivation), GINT_TO_POINTER(MENU_FILE_OPEN));
g_signal_connect(G_OBJECT(miSave), "activate",
    G_CALLBACK(cbActivation), GINT_TO_POINTER(MENU_FILE_SAVE));
g_signal_connect(G_OBJECT(miQuit), "activate",
    G_CALLBACK(cbActivation), GINT_TO_POINTER(MENU_FILE_QUIT));

/**
 * NEW
 *
 * We need to explicitly set the visibility for the menu since by
 * default it's not visible when we get it from window. Safest bet
 * is to show the widget tree that is inside the menu.
 */
gtk_widget_show_all(GTK_WIDGET(menu));
}

/**
 * MODIFIED
 *
 * We switch to the application framework that Hildon provides.
 */
int main(int argc, char** argv) {

    /* Our GtkWindow has been replaced with HildonProgram (NEW). */
    HildonProgram* program;
    /* And we'll have one window, which will act as the container for
       our needs (NEW). */
    HildonWindow* window;
    GtkWidget* label;
    GtkWidget* vbox;

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

    /**
     * NEW
     *
     * Create the Hildon provided window and controller widgets.
     */
    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. */
    window = HILDON_WINDOW(hildon_window_new());
    /* Bind the HildonWindow to HildonProgram. */
    hildon_program_add_window(program, HILDON_WINDOW(window));

    /* Create the label widget. */
    label = g_object_new(GTK_TYPE_LABEL,
        "label", "Hello Hildon (with menus)!",
        NULL);

    /* Build the menu. We'll create the menu for the HildonProgram, so
       we pass the pointer to our revised buildMenu (NEW). */
    buildMenu(program);

    /* Create a layout box for the window since it can only hold one
       widget. */

```

```

vbox = g_object_new(GTK_TYPE_VBOX, NULL);

/* Add the vbox as a child to the Window. */
g_object_set(window, "child", vbox, NULL);

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

/* Connect the termination signals. Note how the HildonWindow
   object has taken the responsibilities that a GtkWindow normally
   would have (NEW). */
g_signal_connect(G_OBJECT(window), "delete-event",
                 G_CALLBACK(delete_event), NULL);
g_signal_connect(G_OBJECT(window), "destroy",
                 G_CALLBACK(end_program), NULL);

/* Show all widgets that are contained by the window. This also
   includes the menu (if it has been setup by this point) (NEW). */
gtk_widget_show_all(GTK_WIDGET(window));

/* Start the main event loop. */
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 2.10: Listing of hildon\_helloworld-1.c

In order to get the required flags for the compiler and linker, we'll use pkg-config again. Since now we'll be pulling flags from multiple packages at the same time, we can combine the package names for each invocation of pkg-config. The package that contains the Hildon widgets is called hildon-1.

```

[sbox-DIABLO_X86: ~/appdev] > gcc -Wall `pkg-config --cflags gtk+-2.0 \
hildon-1` hildon_helloworld-1.c -o hildon_helloworld-1 \
`pkg-config --libs gtk+-2.0 hildon-1`

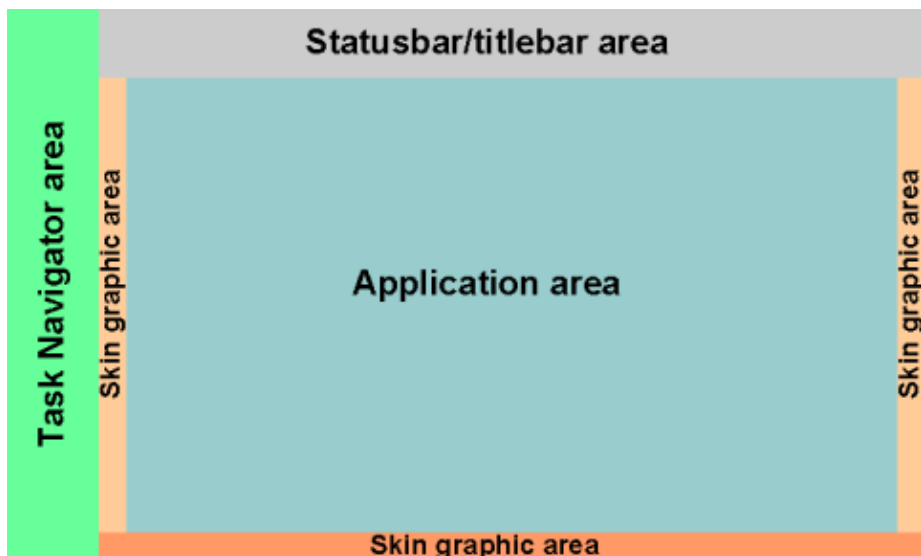
```

Adding new pkg-config package name for programs which use Hildon widgets

When run using **run-standalone.sh** our small program is starting to look like a proper application:



This corresponds to the Normal View layout in the maemo tutorial:



The pixel space available for components:

- The Task navigator takes 80x480 pixels on the left.
- The Statusbar takes 720x60 pixels from the top. This area is partly used by HildonProgram to display the title and by HildonWindow to implement the menu.
- The area left for the application is 672x396 pixels.

Note that these sizes shouldn't be taken for granted. This material will not show you how to place widgets at absolute positions since that wouldn't be

productive in the long run. This is because next generation devices might have different screen sizes, or you might want to run your program in a Linux Desktop environment someday.

## 2.8 Using accessors

You have now gone through the basic program and function structure that you will encounter in GTK+ development. Needless to say, there are many more useful things to learn too. The concept of events and signals is however very important to understand. It is also useful to know the basics of the GObject system even if its use is sometimes difficult.

Most hard-core GTK+ developers will find the use of GObject properties distasteful (at least). They might use the following rationale: since `g_object_set` does not require typecast macros, it will make it impossible for the compiler to check the proper types used when compiling. This is true. However, even by using type-cast macros one might get into trouble.

GTK+ functions check their arguments on use, so in practise when you will pass the wrong types to functions, you will get rather nice warnings during the run time. You should of course fix these, as they are indicators for possible problems. Just bear in mind that according to one leading GTK+ developer "those are just a convenience to the programmer" (meaning the typecast macro expansions and runtime checks) so you should not rely on their presence (indeed, one can use black magic flags to disable their code-expansion and that would lead to somewhat faster code).

The main reason to use the accessor based style is that most existing GTK+ code is written using it. This includes the GTK+ tutorials and the example codes that are sometimes included in the GTK+ API documentation.

Now that you've seen what the GObject-property based style looks like, we will switch to the accessor style for the rest of the material. This is mainly done because that is the style that you'll encounter anyway.

In order to complete this chapter, we round it off with yet another hello world program, which will look and taste exactly the same as the previous one, but we'll be using only the accessor functions. It's possible to mix the styles, but it would be best to pick one and stay consistent. Most projects have already selected their style so you will need to use the existing style in order to facilitate source code comprehension within the project.

```
/**
 * hildon_helloworld-2.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 now convert into accessor functions. Program logic is same as
 * in hildon_helloworld-1.c.
 *
 * Look for lines with "NEW" or "MODIFIED" in them.
 */
#include <stdlib.h>
```



```

#include <gtk/gtk.h>
#include <hildon/hildon-program.h>

/* Menu codes. */
typedef enum {
    MENU_FILE_OPEN = 1,
    MENU_FILE_SAVE = 2,
    MENU_FILE_QUIT = 3
} MenuActionCode;

/**
 * Callback function (event handler) for the "delete" event.
 */
static gboolean delete_event(GtkWidget* widget, GdkEvent event,
                             gpointer data) {
    return FALSE;
}

/**
 * Our callback function for the "destroy"-signal which is issued
 * when the Widget is going to be destroyed.
 */
static void end_program(GtkWidget* widget, gpointer data) {
    gtk_main_quit();
}

/**
 * Signal handler for the menu item selections.
 */
static void cbActivation(GtkMenuItem* mi, gpointer data) {
    MenuActionCode aCode = GPOINTER_TO_INT(data);

    switch(aCode) {
        case MENU_FILE_OPEN:
            g_print("Selected open\n");
            break;
        case MENU_FILE_SAVE:
            g_print("Selected save\n");
            break;
        case MENU_FILE_QUIT:
            g_print("Selected quit\n");
            gtk_main_quit();
            break;
        default:
            g_warning("cbActivation: Unknown menu action code %d.\n", aCode);
    }
}

/**
 * MODIFIED
 *
 * This utility creates the menu for the HildonProgram.
 * GTK+ offers a convenience function to create MenuItems with
 * GtkLabels with alignment already included. We'll use the
 * convenience function instead of the GObject properties.
 */
static void buildMenu(HildonProgram* program) {
    GtkMenu* menu;
    GtkWidget* miOpen;
    GtkWidget* miSave;

```

```

GtkWidget* miSep;
GtkWidget* miQuit;

/* Create the menu items. */
miOpen = gtk_menu_item_new_with_label("Open");
miSave = gtk_menu_item_new_with_label("Save");
miQuit = gtk_menu_item_new_with_label("Quit");
miSep = gtk_separator_menu_item_new();

/* Create a new menu. */
menu = GTK_MENU(gtk_menu_new());

/* Add the items to the container.

Now that we're using accessor functions, things look a bit
different. We're still doing the same logic though.

The prototype for adding children to a container is:
gtk_container_add(GtkContainer*, GtkWidget*)

This is why we don't need to cast the menu items. */
gtk_container_add(GTK_CONTAINER(menu), miOpen);
gtk_container_add(GTK_CONTAINER(menu), miSave);
gtk_container_add(GTK_CONTAINER(menu), miSep);
gtk_container_add(GTK_CONTAINER(menu), miQuit);

/* Connect the signals from individual menu items. */
g_signal_connect(G_OBJECT(miOpen), "activate",
    G_CALLBACK(cbActivation), GINT_TO_POINTER(MENU_FILE_OPEN));
g_signal_connect(G_OBJECT(miSave), "activate",
    G_CALLBACK(cbActivation), GINT_TO_POINTER(MENU_FILE_SAVE));
g_signal_connect(G_OBJECT(miQuit), "activate",
    G_CALLBACK(cbActivation), GINT_TO_POINTER(MENU_FILE_QUIT));

/* Set the top level menu for Hildon program. */
hildon_program_set_common_menu(program, menu);
}

/**
 * MODIFIED
 *
 * We use the accessor functions instead of GObject-properties.
 * Because Hildon API creator functions return GtkWidget pointers,
 * we'll change our pointer types accordingly. This is done for GTK+
 * widgets as well.
 */
int main(int argc, char** argv) {

    HildonProgram* program;
    HildonWindow* window;
    GtkWidget* label;
    GtkWidget* vbox;

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

    /* Create the Hildon program. */
    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. */
    window = HILDON_WINDOW(hildon_window_new());

```

```

/* Bind the HildonWindow to HildonProgram. */
hildon_program_add_window(program, HILDON_WINDOW(window));

/* Create the label widget (NEW). */
label = gtk_label_new("Hello Hildon (with accessors!)");

/* Build the menu and attach it to the HildonProgram. */
buildMenu(program);

/* Create a layout box for the window since it can only hold one
   widget (NEW).

   Using the creator function, we need to pass two parameters to
   it:
   - gboolean: should all children be given equal amount of space?
   - gint:     pixels to leave between the child widgets. */
vbox = gtk_vbox_new(FALSE, 0);

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

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

/* Connect the termination signals. */
g_signal_connect(G_OBJECT(window), "delete-event",
                 G_CALLBACK(delete_event), NULL);
g_signal_connect(G_OBJECT(window), "destroy",
                 G_CALLBACK(end_program), NULL);

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

/* Start the main event loop. */
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 2.11: Listing of hildon\_helloworld-2.c - an accessor-style Hello World

You might be wondering what is the point of having two different APIs for writing GUIs in GTK+? The GObject-property model was written in order to provide an easy way for other programming languages to bind into GTK+ objects. Since some of the languages support dynamic binding (python is a prime example), these language bindings may inspect properties on the fly and keep the glue code quite simple.

## 2.9 Handling dynamic memory

You've seen GTK+ in use so far, but what you haven't encountered is the memory handling mechanisms that GTK+ and Hildon use internally. This material will act cowardly and steer away from the GObject reference counting mechanism and will not tell you how to interact with it. Handling memory allocation and freeing it is done automatically by GTK+ in all normal circumstances for

all the widgets and unless you're creating your own, you will not need to know.

In practise, when ever a widget or any data structure part is created within GTK+ (also Hildon), the routines will allocate some memory from the heap (dynamic memory area) and use GObject-functions to increase the reference count on that memory area. When ever a widget is added to a container (for example), the container code will increase the widget's reference count. When a widget is removed from a container, the removal code will decrease the reference count. When reference count is decreased, it will check whether references hit zero, and if so, it will free the memory. If reference count never goes to zero, that memory will not be freed.

A GTK+ widget will emit a "destroy" signal when it wants to destroy itself (for one reason or another). This is a signal to all the reference holders that they should remove their knowledge about that widget and hence decrease the reference counts. After all other objects have done this, that widget may die in piece and its memory will be freed by the reference-decrement code.

For more information, see the official reference counting documentation of GObject and GObject at [maemo.org](http://maemo.org).

## 2.10 Avoiding deprecated functions

On your journey to the land of existing GTK+ code you will undoubtedly encounter code that uses functions that are about to be removed from GTK+ (or at least the GTK+ developers would like to get rid of them in time). By looking at the API function names it is very difficult to know whether they are deprecated or not (unless you're a GTK+ guru).

For this reason the following compile time flags are useful:

- `GTK_DISABLE_DEPRECATED`: Set to 1 to disable all currently deprecated functions from GTK+
- `GDK_DISABLE_DEPRECATED`: Similar to above but will disable deprecated functions from GDK
- `GDK_PIXBUF_DISABLE_DEPRECATED`: For GDK-Pixbuf
- `G_DISABLE_DEPRECATED`: For GLib
- `GTK_MULTIHEAD_SAFE`: Not really deprecation related, but will disable GTK+ functions that might cause problems in a multihead system (one X display over multiple physical displays).

To use these, pass them as parameters to your compiler (they affect the include files that you use) like this: `-DGTK_DISABLE_DEPRECATED=1`

If you're dealing with software that uses the older version of GTK+ (the older version is actually called Gtk, version 1.2), then you should expect some problems. The older version has a lot of functions whose prototype has changed in the modern GTK+ and also some functions have been implemented in a totally different way. The widget model is about the same, but without GObject and without the property-based access API.

## Chapter 3

# GNU Make and makefiles

### 3.1 What is GNU Make?

Now for a small diversion from our graphical endeavours. We'll introduce a tool that we'll be using from now on to automate our software builds. You'll find it most useful once your project consists of more than one file.

If you already feel comfortable using GNU Make and writing your own Makefiles, feel free to skip this chapter.

The *make* program from the GNU project is a powerful tool to aid implementing automation in your software building process. Beside this, it can be used to automate any task which uses files and in which these files are transformed into some other form. *make* by itself does not know what the files contain or what they represent, but using simple syntax we can teach *make* how to handle them.

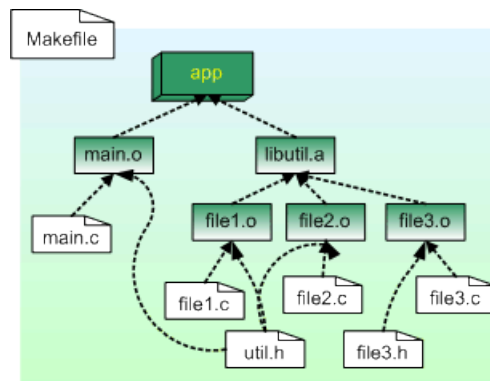
When developing software with *gcc* (and other tools), you will often invoke *gcc* repeatedly with the same parameters and flags. After changing one source file, you'll notice that you have to rebuild other object files and the whole application if some interface changed between the functions. This might happen whenever declarations change, new parameters are added to function prototypes and so on.

These tasks could of course be always done manually, but after a while you'll find yourself wondering whether there is some nicer way of doing things.

GNU *make* is a software building automation tool that will execute repetitive tasks for you. It is controlled via a **Makefile** that contains lists of dependencies between different source files and object files. It also contains lists of commands that should be executed to satisfy these dependencies. *make* uses the **timestamps** of files and the information of the files' existence to automate rebuilding of applications (targets in *make*) as well as the rules that we specify in the **Makefile**.

*make* can be used for other purposes as well. You can easily create a target for installing the built software on destination computer, a target for generating documentation by using some automatic documentation generation tool and so forth. Some people use *make* to keep multiple Linux systems up to date with the newest software and various system configuration changes. In short, *make* is flexible enough to be generally useful.

## 3.2 How does make work?

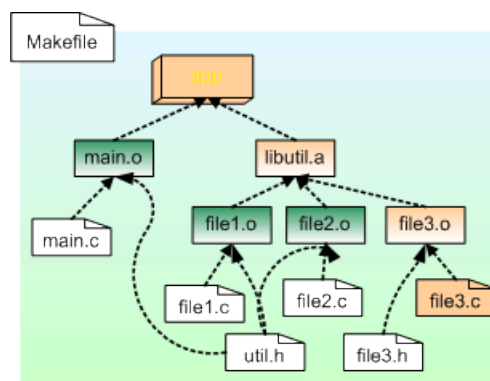


The dependencies between different files making up a software project.

The aim of make is to satisfy the target. Each target has its own dependencies. A user will generally select a target for make to satisfy by supplying the target name on the command line. make will start by checking whether all of the dependencies exist and have an older timestamp than the target. If so, make will do nothing as nothing has changed. However, since a header file (that an application is not dependent on directly) might change, make will propagate the changes to the 'root' of the target as shown in the above picture.

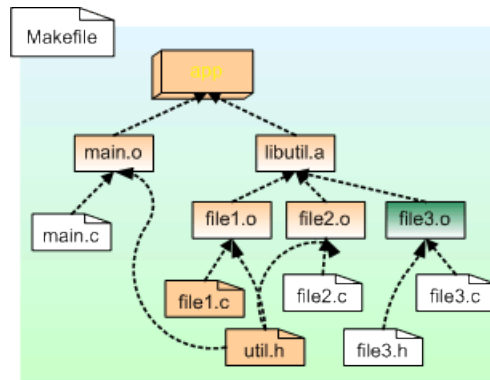
make will rebuild all of the necessary dependencies and targets on the way towards the target. In this way, make will only rebuild those dependencies that actually affect something, and thus, will save time and effort. In big projects the amount of time saved is significant.

To illustrate this, suppose that we modify **file3.c** in the above picture. We then run *make*, and it will automatically rebuild the necessary targets (**file3.o**, **libutil.a** and **app**):



Source file file3.c has been modified. make rebuilds the depending objects and target (**app**)

Now suppose that we add another function to **file1.c**. We would then also need to modify **util.h** accordingly. From the picture you see that quite many objects and targets depend on this header file, so we end up rebuilding a sizable number of objects (but not all):



Header file **util.h** has been modified

Note that in the example pictures we have a project with a custom static library which will be linked against the test application. Developing custom libraries is beyond the scope of this material but the library scenario will continue to serve as an example.

### 3.3 The simplest real example

Before delving too deeply into the syntax of makefiles, it is instructive to first see make in action. For this, we'll use a simple project that is written in the C language.

In C, it is customary to write "header" files (conventional suffix for them is **.h**) and regular source files (**.c**). The header files describe calling conventions, APIs and structures that we want to make usable for the outside world. The **.c**-files contain the implementation of these interfaces. This is nothing new to you hopefully.

We'll start with the following rule: if we change something in the interface file then the binary file containing the code implementation (and other files that use the same interface) should be regenerated. Regeneration in this case means invoking *gcc* to create the binary file out of the C-language source file.

We need to tell make two things at this point:

- If a file's content changes, which other files will that affect? Since a single interface will likely affect multiple other files, make uses a reversed ordering here. For each resulting file, we need to list the files on which this one file depends on.
- What are the commands to regenerate the resulting files when the need arises?

We'll do this as simply as possible. We have a project that consists of two source files and one header file. The contents of these files are listed below:

```
/**
 * The old faithful hello world.
 *
 * This maemo code example is licensed under a MIT-style license,
 * that can be found in the file called "License" in the same
 * directory as this file.
 * Copyright (c) 2007-2008 Nokia Corporation. All rights reserved.
 */

#include <stdlib.h> /* EXIT_* */
#include "hello_api.h" /* sayhello */

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

    return EXIT_SUCCESS;
}
```

Listing 3.1: Contents of simple-make-files/hello.c

```
/**
 * Implementation of sayhello.
 *
 * This maemo code example is licensed under a MIT-style license,
 * that can be found in the file called "License" in the same
 * directory as this file.
 * Copyright (c) 2007-2008 Nokia Corporation. All rights reserved.
 */

#include <stdio.h> /* printf */
#include "hello_api.h" /* sayhello declaration */

void sayhello(void) {
    printf("Hello world!\n");
}
```

Listing 3.2: Contents of simple-make-files/hello\_func.c

```
/**
 * Interface description for the hello_func module.
 *
 * 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.
 */

#ifndef INCLUDE_HELLO_API_H
#define INCLUDE_HELLO_API_H
/* The above is protection against circular header inclusion. */

/* Function to print out "Hello world!\n". */
extern void sayhello(void);

#endif
/* ifndef INCLUDE_HELLO_API_H */
```

Listing 3.3: Contents of simple-make-files/hello\_api.h



So, in effect, we have the main application in **hello.c** which uses a function which is implemented in **hello\_func.c** and declared in **hello\_api.h**. To build an application out of these files we could do it manually like this:

```
gcc -Wall hello.c hello_func.c -o hello
```

Or, we could do it in three stages:

```
gcc -Wall -c hello.c -o hello.o
gcc -Wall -c hello_func.c -o hello_func.o
gcc -Wall hello.o hello_func.o -o hello
```

In both cases, the resulting application (**hello**) is the same.

In the first case, we instruct gcc to process all source files in one go, link the resulting object code and store that code (program in this case) into **hello**.

In the second case, we instruct gcc to create a binary object file for each of the source files. After that, we instruct gcc to link these object files (**hello.o** and **hello\_func.o**) together, and store the linked code into **hello**.

Notice that when gcc reads through the C source files, it will also read in the header files, since the C code uses the `#include` -preprocessor directive. This is because gcc will internally will run all files ending with `.c` through cpp (the preprocessor) first.

The file describing this situation to make is as follows:

```
# define default target (first target = default)
# it depends on 'hello.o' (which will be created if necessary)
# and hello_func.o (same as hello.o)
hello: hello.o hello_func.o
    gcc -Wall hello.o hello_func.o -o hello

# define hello.o target
# it depends on hello.c (and is created from it)
# also depends on hello_api.h which would mean that
# changing the hello.h api would force make to rebuild
# this target (hello.o).
# gcc -c: compile only, do not link
hello.o: hello.c hello_api.h
    gcc -Wall -c hello.c -o hello.o

# define hello_func.o target
# it depends on hello_func.c (and is created from)
# and hello_api.h (since that's its declaration)
hello_func.o: hello_func.c hello_api.h
    gcc -Wall -c hello_func.c -o hello_func.o
```

Listing 3.4: Contents of simple-make-files/Makefile

This file is in the simplest form without using any variables or relying on built-in magic in GNU make. Later we'll see that we could write the same rules in a much shorter way.

You can test this makefile out by running make in the directory which contains the makefile and the source files:

```
user@system:~$ make
gcc -Wall -c hello.c -o hello.o
gcc -Wall -c hello_func.c -o hello_func.o
gcc -Wall hello.o hello_func.o -o hello
```

Running make.

```

user@system:~$ ls -la
total 58
drwxr-xr-x 3  user user 456  Aug 11 15:04 .
drwxr-xr-x 13 user user 576  Aug 11 14:56 ..
-rw-r--r-- 1  user user 699  Jun 1  14:48 Makefile
-rwxr-xr-x 1  user user 4822 Aug 11 15:04 hello
-rw-r--r-- 1  user user 150   Jun 1  14:48 hello.c
-rw-r--r-- 1  user user 824   Aug 11 15:04 hello.o
-rw-r--r-- 1  user user 220   Jun 1  14:48 hello_api.h
-rw-r--r-- 1  user user 130   Jun 1  14:48 hello_func.c
-rw-r--r-- 1  user user 912   Aug 11 15:04 hello_func.o

```

The resulting files after running make.

```

user@system:~$ ./hello
Hello world!

```

Executing the binary.

### 3.4 Anatomy of a makefile

From the simple example above, we can deduce some of make's syntax.

Here are the rules that we can learn:

- Comments are lines that start with the `#`-character. To be more precise, when make reads the makefile, it will ignore the `#`-character and any characters after it up to the end of the line. This means that we could also put comments at the end of lines and make would ignore them, but this is considered bad practise as it would lead to subtle problems later on.
- The backslash character(`\`) can be used to escape the special meaning of next character. The most important special character in makefiles is the dollar-character(`$`), which is used to access the contents of variables. There are also other special characters. To continue a line that is too long to your liking, you may escape the newline character on that line. Place the backslash at the end of your line. make will then ignore the newline when reading input.
- Empty lines by themselves are ignored.
- A line that starts at column 0 (start of the line) and contains a colon character(`:`) is considered a rule. The name on the left side of the colon is created by the commands. This name is called a target. Any filenames specified after the colon are the files that the target depends on. They are called prerequisites (i.e. they're required to exist before make decides to create the target).
- Lines starting with the tabulation character(`tab`) are commands that make will run to achieve the target.

In the printed material the tabs are represented with whitespace, so be careful when reading the example makefiles. Note also that in reality, the command lines are considered as part of the rule.

Using these rules, we can now deduce that:

- make will regenerate **hello** when either or both of its prerequisites change. So, if either **hello.o** or **hello\_func.o** change, make will regenerate **hello** by using the command: `gcc -Wall hello.o hello_func.o -o hello`
- If either **hello.c** or **hello\_api.h** change, make will regenerate **hello.o**.
- If either **hello\_func.c** or **hello\_api.h** change, make will regenerate **hello\_func.o**.

### 3.5 The default goal

Note that nowhere do we explicitly tell make what it should do by default when run without any command line parameters (as we did above). How does it know that creating **hello** is the target for the run?

The first target given in a makefile is the default target. A target that achieves some higher-level goal (like linking all the components into the final application) is sometimes called a goal in GNU make documentation.

So, the first target in the makefile is the default goal when make is run without command line parameters.

Note that in a magic-like way, make will automatically learn the dependencies between the various components and deduce that in order to create **hello**, it will also need to create **hello.o** and **hello\_func.o**. make will regenerate the missing prerequisites when necessary. In fact, this is a quality that causes make do its magic.

Since the prerequisites for **hello** (**hello.o** and **hello\_func.o**) don't exist and **hello** is the default goal, make will first create the files that the **hello**-target needs. You can evidence this from the screen capture of make running without command line parameters. In it, you see the execution of each command in the order that make decides is necessary to satisfy the default goal's prerequisites and finally it will create the **hello**.

```
user@system:~$ make
gcc -Wall -c hello.c -o hello.o
gcc -Wall -c hello_func.c -o hello_func.o
gcc -Wall hello.o hello_func.o -o hello
```

### 3.6 On names of makefiles

The recommended name for your makefile is **Makefile**. This is not the first filename that GNU make will try to open, but it is the most portable one. In fact, the order of filenames that make attempts to open is: **GNUMakefile**, **Makefile** and finally **makefile**.

Unless you're sure that your makefile won't work on other make systems (not GNU), refrain from using **GNUMakefile**. We will be using **Makefile** for the most of this material. The idea behind using **Makefile** instead of **makefile** is related to how shells and commands sort filenames when contents of directories are requested. Since in ASCII the capital letters come before the small case letters, the important files are listed first. Makefiles are considered important since they're the basis for building the project. (The collation order might be different in your locale and your environment.)

You may tell make which file to read explicitly by using the `-f` command line option. This option will be used in the next example.

## 3.7 Questions

Based on common sense, what should make do when we run it after:

- Deleting the **hello** file?
- Deleting **hello.o** file?
- Modifying **hello\_func.c**?
- Modifying **hello.c**?
- Modifying **hello\_api.h**?
- Deleting the **Makefile**?

Think about what would you do if you'd be writing the commands manually.

## 3.8 Adding make goals

Sometimes it's useful to add targets whose execution doesn't result in a file but instead cause some commands to be run. This is commonly used in makefiles of software projects to get rid of the binary files, so that building can be attempted from a clean state. This kind of targets can also be justifiably called goals. GNU documentation uses the name "phony target" for these kinds of targets since they don't result in creating a file like normal targets do.

We'll next create a copy of the original makefile and add a goal that will remove the binary object files and the application. Note that other parts of the makefile do not change.

```
# add a cleaning target (to get rid of the binaries)

# define default target (first target = default)
# it depends on 'hello.o' (which must exist)
# and hello_func.o
hello: hello.o hello_func.o
    gcc -Wall hello.o hello_func.o -o hello

# define hello.o target
# it depends on hello.c (and is created from)
# also depends on hello_api.h which would mean that
# changing the hello.h api would force make to rebuild
# this target (hello.o).
# gcc -c: compile only, do not link
hello.o: hello.c hello_api.h
    gcc -Wall -c hello.c -o hello.o

# define hello_func.o target
# it depends on hello_func.c (and is created from)
# and hello_api.h (since that's its declaration)
hello_func.o: hello_func.c hello_api.h
    gcc -Wall -c hello_func.c -o hello_func.o

# This is the definition of the target 'clean'
# Here we'll remove all the built binaries and
# all the object files that we might have generated
# Notice the -f flag for rm, it means "force" which
```

```
# in turn means that rm will try to remove the given
# files, and if there are none, then that's ok. Also
# it means that we have no writing permission to that
# file and have writing permission to the directory
# holding the file, rm will not then ask for permission
# interactively.
clean:
    rm -f hello hello.o hello_func.o
```

Listing 3.5: Contents of simple-make-files/Makefile.2

In order for make to use this file instead of the default Makefile, we need to use the `-f` command line parameter as follows:

```
user@system:~$ make -f Makefile.2
gcc -Wall -c hello.c -o hello.o
gcc -Wall -c hello_func.c -o hello_func.o
gcc -Wall hello.o hello_func.o -o hello
```

We tell make to use Makefile.2 instead of the default one.

To control which target make will pursue (instead of the default goal), we need to give the target name on the command line like this:

```
user@system:~$ make -f Makefile.2 clean
rm -f hello hello.o hello_func.o
```

Test the clean target.

```
user@system:~$ ls -la
total 42
drwxr-xr-x 3 user user 376 Aug 11 15:08 .
drwxr-xr-x 13 user user 576 Aug 11 14:56 ..
-rw-r--r-- 1 user user 699 Jun 1 14:48 Makefile
-rw-r--r-- 1 user user 1279 Jun 1 14:48 Makefile.2
-rw-r--r-- 1 user user 150 Jun 1 14:48 hello.c
-rw-r--r-- 1 user user 220 Jun 1 14:48 hello_api.h
-rw-r--r-- 1 user user 130 Jun 1 14:48 hello_func.c
```

No binary files remain in the directory.

## 3.9 Making a target at a time

Sometimes it's useful to ask make to process only one target. Similar to the clean-case, we just need to give the target name on the command line:

```
user@system:~$ make -f Makefile.2 hello.o
gcc -Wall -c hello.c -o hello.o
```

Making a single target.

We can also supply multiple target names as individual command line parameters:

```
user@system:~$ make -f Makefile.2 hello_func.o hello
gcc -Wall -c hello_func.c -o hello_func.o
gcc -Wall hello.o hello_func.o -o hello
```

Making two targets at a time.

You can do this with any number of targets, even phony ones. `make` will try to complete all of them in the order that they're on the command line. Should any of the commands within the targets fail, `make` will abort at that point, and will not pursue the rest of the targets.

### 3.10 PHONY-keyword

Suppose that for some reason or another, a file called `clean` appears in the directory in which we run `make` with `clean` as the target. In this case, `make` would probably decide that since `clean` already exists, there's no need to run the commands leading to the target, and in our case `make` would not run the `rm` command at all. Clearly this is something we want to avoid.

For these cases, GNU `make` provides a special target called `.PHONY` (note the leading dot). We list the real phony targets (`clean`) as a dependency to `.PHONY`. This will signal to `make` that it should never consider a file called `clean` to be the result of the phony target.

In fact, this is what most open source projects that use `make` do.

This leads in the following makefile (comments omitted for brevity):

```
hello: hello.o hello_func.o
    gcc -Wall hello.o hello_func.o -o hello

hello.o: hello.c hello_api.h
    gcc -Wall -c hello.c -o hello.o

hello_func.o: hello_func.c hello_api.h
    gcc -Wall -c hello_func.c -o hello_func.o

.PHONY: clean
clean:
    rm -f hello hello.o hello_func.o
```

Listing 3.6: Using `.PHONY` (simple-make-files/Makefile.3)

### 3.11 Specifying the default goal

We know that `make` will use the first target in a makefile as its default goal. What if we explicitly want to set the default goal instead? Since it is not possible to actually change the "first target is the default goal"-setting in `make`, we will have to take this into account. So, we'll just add a new target and make sure that it will be processed as the first target in a makefile.

In order to achieve this, we create a new phony target and list the wanted targets as the phony target's prerequisites. You can use whatever name you want for the target but `all` is a very common name for this use. The only thing you need to remember is that this target needs to be the first one in the makefile.

```
.PHONY: all
all: hello

hello: hello.o hello_func.o
    gcc -Wall hello.o hello_func.o -o hello
```

```

hello.o: hello.c hello_api.h
    gcc -Wall -c hello.c -o hello.o

hello_func.o: hello_func.c hello_api.h
    gcc -Wall -c hello_func.c -o hello_func.o

.PHONY: clean
clean:
    rm -f hello hello.o hello_func.o

```

Listing 3.7: Explicit default goal (simple-make-files/Makefile.4)

You will notice something peculiar in the listing above. Since the first target is the default goal for make, won't `.PHONY` now be the default target? Since `.PHONY` is a special target in GNU make, we're safe. Also, because of compatibility reasons, GNU make will ignore any targets that start with a leading dot (`.`) when looking for the default goal.

## 3.12 Other common phony goals

You will encounter many other phony goals in makefiles that projects use.

Some of the more common ones include:

- `install`: Prerequisites for `install` is the software application binary (or binaries). The commands (normally *install* is used on Linux) will specify which binary file to place under which directory on the system (`/usr/bin`, `/usr/local/bin`, etc).
- `distclean`: Similar to `clean`, remove object files and such, but remove other files as well (sounds scary). Normally this is used to implement the removal of **Makefile** in the case that it was created by some automatic tool (*configure* for example).
- `package`: prerequisites are as in `install`, but instead of installing, create an archive file (*tar*) with the files in question. This archive can then be used to distribute the software.

You can find other common phony targets in the GNU make manual (not the man-page!).

## 3.13 Variables in makefiles

So far our files have been listing filenames explicitly and hopefully you've noticed that writing makefiles in this way can get rather tedious if not error prone.

This is why all make programs (not just the GNU variant) provide variables. Some other make programs call them macros.

Variables work almost as they do inside regular UNIX command shells. They are a simple mechanism by which we can associate a piece of text with a name that can be referenced later on in multiple places in our makefiles. make variables are based on text-replacement just like shell variables are.

## 3.14 Variable flavors

The variables that we can use in makefiles come in two basic styles or flavors.

The default flavor is where referencing a variable will cause make to expand the variable contents at the point of reference. This means that if the value of the variable is some text in which we reference other variables, their contents will also be replaced automatically. This flavor is called *recursive*.

The other flavor is *simple*, meaning that the content of a variable is evaluated only once, when we define the variable.

Deciding on which flavor to use might be important when the evaluation of variable contents needs to be repeatable and fast. In these cases simple variables are often the correct solution.

## 3.15 Recursive variables

Here are the rules for creating recursive variables:

- Names must contain only ASCII alphanumeric characters and underscores (to preserve portability).
- You should use only lower case letters in the names of your variables. make is case sensitive and variable names written in capital letters are used when we want to communicate the variables outside make or their origin is from outside (environment variables). This is however a convention only, and you will also see variables that are local to the makefile, but still written in all capital letters.
- Values may contain any text. The text will be evaluated by make when the variable is used. Not when it's defined.
- Long lines may be broken up with a backslash character ( \ ) and newline combination. This same mechanism can be used to continue other long lines as well (not just variables). Do not put any whitespace after the backslash.
- Do not reference the variable that you're defining in its text portion. This would result in an endless loop whenever this variable would be used. GNU make will stop on error in this case.

We'll re-use the same makefile as before, but introduce some variables. You will also see the syntax on how to define the variables and how to use them (reference them):

```
# define the command that we use for compilation
CC = gcc -Wall

# which targets do we want to build by default?
# note that 'targets' is just a variable, its name
# does not have any special meaning to make
targets = hello

# first target defined will be the default target
# we use the variable to hold the dependencies
```



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

hello: hello.o hello_func.o
    $(CC) hello.o hello_func.o -o hello

hello.o: hello.c hello_api.h
    $(CC) -c hello.c -o hello.o

hello_func.o: hello_func.c hello_api.h
    $(CC) -c hello_func.c -o hello_func.o

# we'll make our cleaning target more powerful
# we remove the targets that we build by default and also
# remove all object files
.PHONY: clean
clean:
    rm -f $(targets) *.o
```

Listing 3.8: The makefile with some variables (simple-make-files/Makefile.5)

The CC variable is the standard variable to hold the name of the C-compiler executable. GNU make comes pre-loaded with a list of tools which can be accessed in similar way (we'll see \$(RM) shortly, but there are others). Most UNIX-tools related to building software already have similar pre-defined variables in GNU make. Here we override one of them for no other reason than to demonstrate how it's done. Overriding variables like this is not recommended since the user running make later might want to use some other compiler, and would have to edit the makefile to do so.

## 3.16 Simple variables

Suppose you have a makefile like this:

```
CC = gcc -Wall

# we want to add something to the end of the variable
CC = $(CC) -g

hello.o: hello.c hello_api.h
    $(CC) -c hello.c -o hello.o
```

Listing 3.9: The makefile with some variables (simple-make-files/Makefile.5)

What might seem quite logical to a human reader, won't seem very logical to make.

Since the contents of recursive variables are evaluated when they're referenced, you will notice that the above fragment will lead to an infinite loop.

How do we correct this? make actually provides two mechanisms for this. This is the solution with simple variables:

```
CC := gcc -Wall

# we want to add something to the end of the variable
CC := $(CC) -g

hello.o: hello.c hello_api.h
```

```
$(CC) -c hello.c -o hello.o
```

Listing 3.10: The makefile with some variables (simple-make-files/Makefile.5)

Notice that the equals character(=) has been changed into := .

This is how we create simple variables. Whenever we reference a simple variable, make will just replace the text that is contained in the variable without evaluating it. It will do the evaluation only when we define the variable. In the above example, CC is created with the content of gcc -Wall (which is evaluated, but is just plain text) and when we next time define the CC-variable, make will evaluate \$(CC) -g which will be replaced with gcc -Wall -g as one might expect.

So, the only two differences between the variable flavors are:

- We use := when defining simple variables.
- make evaluates the contents when the simple variable is defined and not when it's referenced later.

Most of the time you'll want to use the recursive variable flavor since it does what you want.

There was a mention about two ways of appending text to an existing variable. The other mechanism is the += operation as follows:

```
CC = gcc -Wall

# we want to add something to the end of the variable
CC += -g

hello.o: hello.c hello_api.h
    $(CC) -c hello.c -o hello.o
```

Listing 3.11: The makefile with some variables (simple-make-files/Makefile.5)

You can use the prepending operation with simple variables too. make will not change the type of variable on the left side of the += operator.

### 3.17 Automatic variables

There is a predefined set of variables inside make that can be used to avoid repetitive typing when writing out the commands in a rule.

This is a list of the most useful ones:

- \$< : replaced by first prerequisite of the rule.
- \$^ : replaced by the list of all prerequisites of the rule.
- \$@ : replaced by the target name.
- \$? : list of prerequisites that are newer than the target is (if target doesn't exist, they are all considered newer).

By rewriting our makefile using automatic variables we get:

```

# add the warning flag to CFLAGS-variable
CFLAGS += -Wall

targets = hello

.PHONY: all
all: $(targets)

hello: hello.o hello_func.o
    $(CC) $^ -o $@

hello.o: hello.c hello_api.h
    $(CC) $(CFLAGS) -c $< -o $@

hello_func.o: hello_func.c hello_api.h
    $(CC) $(CFLAGS) -c $< -o $@

.PHONY: clean
clean:
    $(RM) $(targets) *.o

```

Listing 3.12: Contents of simple-make-files/Makefile.9

Note the cases when we use `$^` instead of `$<` in the above snippet. We don't want to pass the header files for compilation to the compiler since the source file already includes the header files. For this reason we use `$<`. On the other hand, when we're linking programs and have multiple files to put into the executable, we'd normally use `$^`.

We're relying on a couple of things here:

- `$(RM)` and `$(CC)` will be replaced with paths to the system compiler and removal commands.
- `$(CFLAGS)` is a variable that contains a list of options to pass whenever make will invoke the C compiler.

You might also notice that all these variable names are in capital letters. This signifies that they have been communicated from the system environment to make. In-fact, if you create an environment variable called `CFLAGS` make will create it internally for any makefile that it will process. This is the mechanism to communicate variables into makefiles from outside.

Writing variables in all capital letters is a convention signalling external variables or environmental variables, so this is the reason why you should use lower case letters in your own private variables within a **Makefile**.

### 3.18 Integrating with pkg-config

You now have the knowledge to write a simple **Makefile** for the Hello World example. In order to get the the result of *pkg-config*, we will use the GNU make `$(shell command params)`-function. Its function is similar to the backtick operation of the shell.

```

# define a list of pkg-config packages we want to use
pkg_packages := gtk+-2.0 hildon-1

```

```

# get the necessary flags for compiling
PKG_CFLAGS := $(shell pkg-config --cflags $(pkg_packages))
# get the necessary flags for linking
PKG_LDFLAGS := $(shell pkg-config --libs $(pkg_packages))

# additional flags
# -Wall: warnings
# -g: debugging
ADD_CFLAGS := -Wall -g

# combine the flags (so that CFLAGS/LDFLAGS from the command line
# still work).
CFLAGS := $(PKG_CFLAGS) $(ADD_CFLAGS) $(CFLAGS)
LDFLAGS := $(PKG_LDFLAGS) $(LDFLAGS)

targets = hildon_helloworld-1

.PHONY: all
all: $(targets)

hildon_helloworld-1: hildon_helloworld-1.o
    $(CC) $^ -o $@ $(LDFLAGS)

hildon_helloworld-1.o: hildon_helloworld-1.c
    $(CC) $(CFLAGS) -c $< -o $@

.PHONY: clean
clean:
    $(RM) $(targets) *.o

```

Listing 3.13: Contents of simple-make-files/Makefile.10

You might be wondering where the CC and RM variables come from. We certainly didn't declare them anywhere in the Makefile. GNU make comes with a list of predefined variables for many programs and their arguments. GNU make also contains a preset list of pattern rules (which we won't be going in here), but basically these are predefined rules that are used when (for example) one needs an .o file from an .c file. The rules that we specified manually above are actually the same rules that GNU make already contains, so we can make our **Makefile** even more compact by only specifying the dependencies between files.

This leads to the following, slightly simpler version:

```

# define a list of pkg-config packages we want to use
pkg_packages := gtk+-2.0 hildon-1

PKG_CFLAGS := $(shell pkg-config --cflags $(pkg_packages))
PKG_LDFLAGS := $(shell pkg-config --libs $(pkg_packages))

ADD_CFLAGS := -Wall -g

# combine the flags
CFLAGS := $(PKG_CFLAGS) $(ADD_CFLAGS) $(CFLAGS)
LDFLAGS := $(PKG_LDFLAGS) $(LDFLAGS)

targets = hildon_helloworld-1

.PHONY: all
all: $(targets)

```

```
# we can omit the rules and just specify the dependencies  
# for our simple project this doesn't seem like a big win  
# but for larger projects where you have multiple object  
# files, this will save considerable time.  
hildon_helloworld-1: hildon_helloworld-1.o  
hildon_helloworld-1.o: hildon_helloworld-1.c  
  
.PHONY: clean  
clean:  
    $(RM) $(targets) *.o
```

Listing 3.14: Contents of simple-make-files/Makefile.11

An up-to-date manual for GNU make can be found in [gnu.org](http://gnu.org)

## Chapter 4

# More Widgets

### 4.1 Using menus in Hildon

Since each application view will have only one menu, how can you implement multiple menus in an application as you'd do by using a menu bar? One solution to this (taken in the maemo framework) is to build hierarchical menus and put them under the top level menu.

In this way, you might think that the top-level menu is actually acting as a menu bar for your application, but since you have multiple view possibility, you will have to think about how to organise them around your application. Try to keep your GUI design consistent with existing applications that use Hildon, so that the user will not have to learn different models of doing things.

We will now extend our menu so that it will include a sub-menu for selecting styles and the style menu will also demonstrate how to use checkbox and radio style menu items.

We'll also introduce a GTK+ convenience function while building the sub-menu and extend our menu callback function to handle the new items from the style menu.

Since there is a lot of new code, the whole source for the new version is presented below:

```
/**
 * hildon_helloworld-3.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 now add a submenu with radio and check menu items.
 *
 * Look for lines with "NEW" or "MODIFIED" in them.
 */

#include <stdlib.h>
#include <gtk/gtk.h>
#include <hildon/hildon-program.h>

/* Menu codes. */
```

```

typedef enum {
    MENU_FILE_OPEN = 1,
    MENU_FILE_SAVE = 2,
    MENU_FILE_QUIT = 3,
    /* NEW:

        We will have three new signals handled by our signal handler. */
    MENU_STYLE_UNDERLINE_TOGGLED = 4,
    MENU_STYLE_NORMAL_TOGGLED = 5,
    MENU_STYLE_ITALIC_TOGGLED = 6
} MenuActionCode;

/**
 * Callback function (event handler) for the "delete" event.
 */
static gboolean delete_event(GtkWidget* widget, GdkEvent event,
                             gpointer data) {
    return FALSE;
}

/**
 * Our callback function for the "destroy"-signal which is issued
 * when the Widget is going to be destroyed.
 */
static void end_program(GtkWidget* widget, gpointer data) {
    gtk_main_quit();
}

/**
 * MODIFIED
 *
 * Signal handler for the menu item selections.
 */
static void cbActivation(GtkMenuItem* mi, gpointer data) {

    MenuActionCode aCode = GPOINTER_TO_INT(data);

    switch(aCode) {
        case MENU_FILE_OPEN:
            g_print("Selected open\n");
            break;
        case MENU_FILE_SAVE:
            g_print("Selected save\n");
            break;
        case MENU_FILE_QUIT:
            g_print("Selected quit\n");
            gtk_main_quit();
            break;
        case MENU_STYLE_UNDERLINE_TOGGLED:
            /* NEW

                Check items can be in two states. In order to get the
                current (new) state, we use an API function. The API
                function expects a GtkCheckMenuItem widget as its parameter,
                hence the typecast. */
            g_print("Style underline has been toggled. New state is: %s\n",
                    gtk_check_menu_item_get_active(
                        GTK_CHECK_MENU_ITEM(mi))?"on":"off");
            break;
        case MENU_STYLE_NORMAL_TOGGLED:
        case MENU_STYLE_ITALIC_TOGGLED:
            /* NEW

```

```

We handle toggles for both radio items in the same code.
This code will be called twice on each toggle since one of
the radio items will be toggled off, the other will be
toggled on.

The order is as follows:
1) Both go to 'off'-state
2) One of the toggles goes into 'on'-state.

Our logic is constructed as follows:
1) Find out whether the radio item that emitted this signal
   is 'on' (active).
2) If so, determine which of the radio items it is. For this
   we use another switch and then print the respective
   message.
3) If not, we don't do anything since this signal will be
   shortly repeated from the other radio item. */
{
    gboolean isActive = FALSE;

    isActive = gtk_check_menu_item_get_active(
        GTK_CHECK_MENU_ITEM(mi));
    /* This is only for debugging. aCode for NORMAL is 5, and for
       ITALIC it is 6. This will demonstrate that at some point,
       both radio items are "off" at the same time. */
    g_print(" style: aCode=%d isActive=%d\n", aCode, isActive);
    if (isActive) {
        g_print("Style 'slant' has changed: ");
        switch (aCode) {
            case MENU_STYLE_NORMAL_TOGGLED:
                g_print("Normal selected\n");
                break;
            case MENU_STYLE_ITALIC_TOGGLED:
                g_print("Italic selected\n");
                break;
            /* This keeps gcc from complaining. */
            default: break;
        }
        break;
    }
    default:
        g_warning("cbActivation: Unknown menu action code %d.\n", aCode);
}

/**
 * NEW
 *
 * We create a submenu which will contain three items:
 * - A checkbox item for selecting whether underlining is requested.
 * - A group of two radio items out of which only one can be selected
 *   at any given time (as is common for radio button groups).
 *
 * Returns:
 * - The sub-menu (as GtkWidget*)
 */
static GtkWidget* buildSubMenu(void) {
    GtkWidget* subMenu;

```



```

GtkWidget* miUnderline;
GtkWidget* miSep;
GtkWidget* miItalic;
GtkWidget* miNormal;

/* Create a checkbox menu item. */
miUnderline = gtk_check_menu_item_new_with_label("Underline");
/* Set it as "checked" (default is FALSE). */
gtk_check_menu_item_set_active(GTK_CHECK_MENU_ITEM(miUnderline),
    TRUE);

/* Create the radio items.

    If you're wondering what this extra brace is doing here, don't
    worry :-). It's a scope so that we can define a variable and
    forget about it quickly when we leave the scope. You should
    check your style guidelines whether this is a good idea or not.

    It was used here so that we won't access group by mistake since
    its memory and reference counts are controlled by GTK+. */
{
    GSList* group = NULL;

    /* Normally the first parameter would be the group into which
       this new radio menu item would be added. If we leave it as
       NULL, GTK+ will create a new group for that which is of the
       type GSList (GLib single-linked list). */
    miItalic = gtk_radio_menu_item_new_with_label(NULL, "Italic");
    /* Get the group so that we can add another radio menu item. */
    group = gtk_radio_menu_item_get_group(
        GTK_RADIO_MENU_ITEM(miItalic));
    /* Create another radio menu item and add it as well. */
    miNormal = gtk_radio_menu_item_new_with_label(group, "Normal");
    /* You will also sometimes see code like this:
       miItalic = gtk_rad..el(NULL, "Normal");
       miNormal = gtk_rad..el_from_widget(
           GTK_RADIO_MENU_ITEM(miItalic, "Normal"));

       This is a shortcut so that we don't need to get the group for the
       widget every time. For multiple radio menu items (or buttons
       too), this will get rather tedious. */
}

/* Create the separator item. */
miSep = gtk_separator_menu_item_new();

/* Create the menu to hold the items. */
subMenu = gtk_menu_new();

/* Add the items to the menu.

    If you see code like gtk_menu_append, re-write that to use this
    version since gtk_menu_append is deprecated. GtkMenuShell is an
    abstract superclass for menus-style widgets that can hold
    selectable child-widgets.

    You could also use gtk_container_add for adding the menu items
    but gtk_menu_shell also contains possibilities for using
    different orders for the items so it's useful to know that it
    exists. */
gtk_menu_shell_append(GTK_MENU_SHELL(subMenu), miUnderline);
gtk_menu_shell_append(GTK_MENU_SHELL(subMenu), miSep);

```

```

gtk_menu_shell_append(GTK_MENU_SHELL(subMenu), miItalic);
gtk_menu_shell_append(GTK_MENU_SHELL(subMenu), miNormal);

/* Connect the signals. */
g_signal_connect(G_OBJECT(miUnderline), "toggled",
    G_CALLBACK(cbActivation),
    GINT_TO_POINTER(MENU_STYLE_UNDERLINE_TOGGLED));
g_signal_connect(G_OBJECT(miItalic), "toggled",
    G_CALLBACK(cbActivation),
    GINT_TO_POINTER(MENU_STYLE_ITALIC_TOGGLED));
g_signal_connect(G_OBJECT(miNormal), "toggled",
    G_CALLBACK(cbActivation),
    GINT_TO_POINTER(MENU_STYLE_NORMAL_TOGGLED));

/* The submenu is ready, return it to caller. */
return subMenu;
}

/**
 * MODIFIED
 *
 * This utility creates the menu for the HildonProgram.
 */
static void buildMenu(HildonProgram* program) {

    GtkWidget* menu;
    GtkWidget* miOpen;
    GtkWidget* miSave;
    /* Menu item whichs opens the style submenu (NEW). */
    GtkWidget* miStyle;
    GtkWidget* subMenu;
    GtkWidget* miSep;
    GtkWidget* miQuit;

    /* Create the menu items. */
    miOpen = gtk_menu_item_new_with_label("Open");
    miSave = gtk_menu_item_new_with_label("Save");
    miStyle = gtk_menu_item_new_with_label("Style");
    miQuit = gtk_menu_item_new_with_label("Quit");
    miSep = gtk_separator_menu_item_new();

    /* Build the submenu (NEW). */
    subMenu = buildSubMenu();

    /* Connect the style-item so that it will pop up the submenu (NEW).
     */
    gtk_menu_item_set_submenu(GTK_MENU_ITEM(miStyle), subMenu);

    /* Create a new menu. */
    menu = GTK_MENU(gtk_menu_new());

    /* Add the items to the container. */
    gtk_container_add(GTK_CONTAINER(menu), miOpen);
    gtk_container_add(GTK_CONTAINER(menu), miSave);
    gtk_container_add(GTK_CONTAINER(menu), miStyle);
    gtk_container_add(GTK_CONTAINER(menu), miSep);
    gtk_container_add(GTK_CONTAINER(menu), miQuit);

    /* Connect the signals from individual menu items. */
    g_signal_connect(G_OBJECT(miOpen), "activate",
        G_CALLBACK(cbActivation), GINT_TO_POINTER(MENU_FILE_OPEN));
    g_signal_connect(G_OBJECT(miSave), "activate",

```

```

        G_CALLBACK(cbActivation), GINT_TO_POINTER(MENU_FILE_SAVE));
    g_signal_connect(G_OBJECT(miQuit), "activate",
        G_CALLBACK(cbActivation), GINT_TO_POINTER(MENU_FILE_QUIT));

    /* Set the top level menu for Hildon program. */
    hildon_program_set_common_menu(program, menu);
}

/**
 * The main program remains the same as before.
 */
int main(int argc, char** argv) {

    HildonProgram* program;
    HildonWindow* window;
    GtkWidget* label;
    GtkWidget* vbox;

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

    /* Create the Hildon program. */
    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. */
    window = HILDON_WINDOW(hildon_window_new());
    /* Bind the HildonWindow to HildonProgram. */
    hildon_program_add_window(program, HILDON_WINDOW(window));

    /* Create the label widget. */
    label = gtk_label_new("Hello Hildon (with submenus!)");

    /* Build the menu and attach it to the HildonProgram. */
    buildMenu(program);

    /* 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(window), vbox);

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

    /* Connect the termination signals. */
    g_signal_connect(G_OBJECT(window), "delete-event",
        G_CALLBACK(delete_event), NULL);
    g_signal_connect(G_OBJECT(window), "destroy",
        G_CALLBACK(end_program), NULL);

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

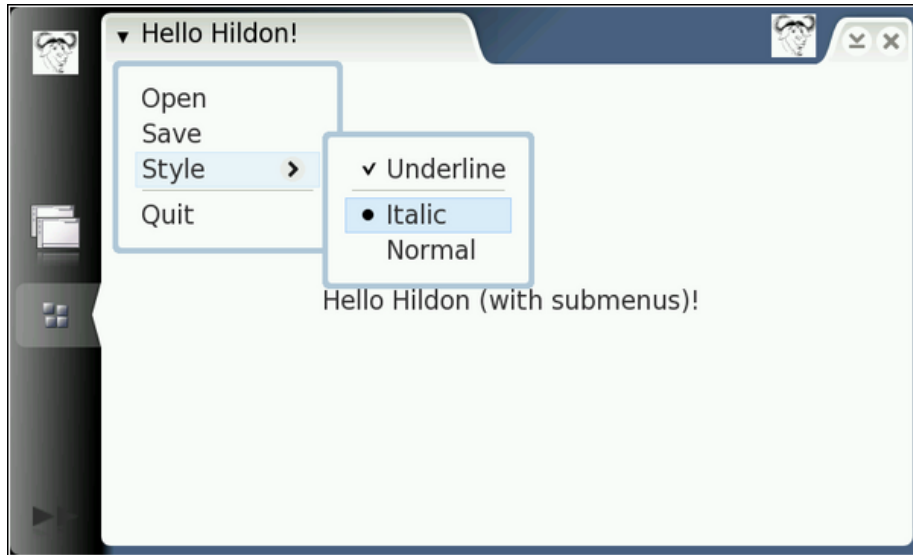
    /* Start the main event loop. */
    g_print("main: calling gtk_main\n");
    gtk_main();

    g_print("main: returned from gtk_main and exiting with success\n");

    return EXIT_SUCCESS;
}

```

Listing 4.1: New version with sub-menu and toggle items (hildon\_helloworld-3.c)



As you can see, implementing even simple GUI programs with GTK+ is somewhat tedious if you've used graphical dialog editors before. Also the programming language isn't really object-oriented, so you end up typing more.

This is why there are several approaches into building GUIs in a more automatic way (XML-based descriptions seem to be rather the hype right now). These tools will not be covered in this material because it still is useful to know what the code generated by the tools does. Also, tools get out of date with respect to GTK+ itself quite often and might use deprecated APIs or have other issues.

## 4.2 Adding toolbars

GTK+ has a rich toolbar widget which can hold different kinds of sub-widgets. All of these sub-widgets must be of the `GtkToolItem`-class. So, even when adding labels, icons or ordinary `GtkButtons`, you will need an `GtkToolItem` widget to "hold them". This is because toolbars support tool hiding, detaching and other finer actions of a modern graphical environment, and these actions need partly to be implemented at the individual toolbar item level. Most of the more advanced features of `GtkToolbar` will have limited value when running applications that use maemo.

We'll add buttons for couple of the file-menu operations first. We'll also add a button to do searching and for selecting a color:

```
/**
 * hildon_helloworld-4.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.
*
* Adds a toolbar and a color chooser.
*
* Look for lines with "NEW" or "MODIFIED" in them.
*/

#include <stdlib.h>
#include <gtk/gtk.h>
#include <hildon/hildon-program.h>
/* New Hildon widget. */
#include <hildon/hildon-color-button.h>

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

/**
 * NEW
 *
 * Invoked by "clicked" signal from Hildon Color Button when the user
 * closes the dialog (even when user closes the dialog via cancel).
 * This is because the "clicked" is sent by the GtkButton-code, so it
 * will not know that the color hasn't actually changed at all.
 */
static void cbColorChanged(HildonColorButton* colorButton,
                           gpointer dummy) {
    /* Initialize one GdkColor structure on the stack with zero. */
    GdkColor color = {};

    /* Get the new color information from the button and print it out.
       Note that colors are represented as 16-bit unsigned integers in
       GDK. */
    hildon_color_button_get_color(colorButton, &color);
    g_print("Got new color: r=%d g=%d b=%d\n", color.red, color.green,
            color.blue);
}

/**
 * NEW
 *
 * Utility function that will create the toolbar for us.
 */
static GtkWidget* buildToolbar(void) {

    GtkToolbar* toolbar;
    /* We use GtkToolItems here since that is the type that is returned
       by the gtk_tool_* functions. */
    GtkToolItem* tbOpen;
    GtkToolItem* tbSave;
    GtkToolItem* tbSep;
    GtkToolItem* tbFind;
    /* We'll use one Hildon-widget in this example. */
    GtkWidget* colorButton;
    /* Since we cannot just put GtkWidget into the toolbar we'll need
       to create a holder (GtkToolItem) for the button. */
    GtkToolItem* tbColorButton;

    /* GTK+ includes a facility using which we may reuse existing icons
       in our buttons and other widgets (that support icons). These are
       called "stock items" and are referenced via defines in GTK+.
    */
}

```

```

    Note also that gtk_tool_button_new_from_stock is a convenience
    function that will create a GtkToolItem and add a widget into it
    (in this case a GtkButton which will contain a GtkImage). */
tbOpen = gtk_tool_button_new_from_stock(GTK_STOCK_OPEN);
tbSave = gtk_tool_button_new_from_stock(GTK_STOCK_SAVE);
/* Separator between the file operations and others. */
tbSep = gtk_separator_tool_item_new();
tbFind = gtk_tool_button_new_from_stock(GTK_STOCK_FIND);
/* Create a ToolItem and put the color button inside of it.*/
tbColorButton = gtk_tool_item_new();
colorButton = hildon_color_button_new();
gtk_container_add(GTK_CONTAINER(tbColorButton), colorButton);

/* Create the toolbar. */
toolbar = GTK_TOOLBAR(gtk_toolbar_new());

/* Add the tool items to the toolbar.
NOTE:
    We could use gtk_container_add as well, but we'd need to use
    even more casting macros, so we'll use this API function
    instead. */
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);

/* Connect the signals from the tool items.
NOTE:
    We are a bit evil here since we assume that the callback
    function will not assume or use the widget pointer that is
    passed with these events. In our callback we don't use it for
    open/save so it's safe in this case. */
g_signal_connect(G_OBJECT(tbOpen), "clicked",
                 G_CALLBACK(cbActivation),
                 GINT_TO_POINTER(MENU_FILE_OPEN));
g_signal_connect(G_OBJECT(tbSave), "clicked",
                 G_CALLBACK(cbActivation),
                 GINT_TO_POINTER(MENU_FILE_SAVE));

/* We also connect a signal from the colorbutton when a user has
selected a color. Note that we connect the button itself, not
the toolitem that is holding it inside toolbar (it doesn't emit
signals). */
g_signal_connect(G_OBJECT(colorButton), "clicked",
                 G_CALLBACK(cbColorChanged), NULL);

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

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

/**
 * MODIFIED
 *
 * We use the toolbar creation code and add it to our program.
 */
int main(int argc, char** argv) {
    HildonProgram* program;

```

```

HildonWindow* window;
GtkWidget* label;
GtkWidget* vbox;

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

/* Create the Hildon program. */
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. */
window = HILDON_WINDOW(hildon_window_new());
/* Bind the HildonWindow to HildonProgram. */
hildon_program_add_window(program, HILDON_WINDOW(window));

/* Create the label widget. */
label = gtk_label_new("Hello Hildon (with toolbar)!");

/* Build the menu and attach it to the HildonProgram. */
buildMenu(program);

/* 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(window), vbox);

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

/* Add the toolbar to the Hildon window (NEW). */
hildon_window_add_toolbar(HILDON_WINDOW(window),
                        GTK_TOOLBAR(buildToolbar()));

/* Connect the termination signals. */
g_signal_connect(G_OBJECT(window), "delete-event",
                G_CALLBACK(delete_event), NULL);
g_signal_connect(G_OBJECT(window), "destroy",
                G_CALLBACK(end_program), NULL);

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

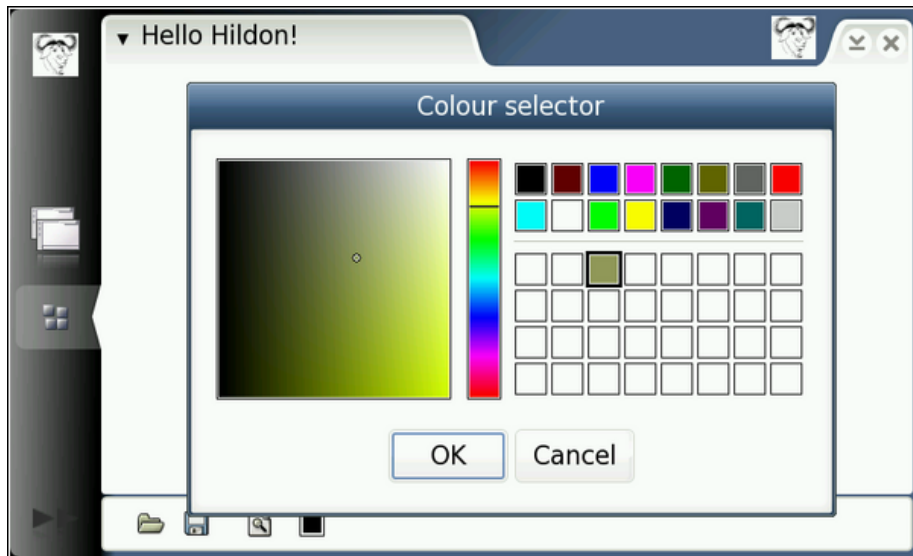
/* Start the main event loop. */
g_print("main: calling gtk_main\n");
gtk_main();

g_print("main: returned from gtk_main and exiting with success\n");

return EXIT_SUCCESS;
}

```

Listing 4.2: Code that adds a toolbar and a color selection button (hildon\_helloworld-4.c)



Running the application

### 4.3 Designing Application State

Next we'd like our Find-button to actually do something. We could create a callback that would be called when we'd press the button (signal name for catching that is "clicked"). The problem is this: what do we pass to the callback so that it can actually control the search toolbar? In our program we want to be able to change the visibility of the toolbar. We also want to let the user control whether the main toolbar is visible or not since the user might not need it all of the time.

How to achieve this? This is where our program will start looking more like a real program instead of a toy. So far we've used the `gpointer` parameter in `g_signal_connect` to pass immediate data related to the particular signal connection. In many cases the callbacks need to modify some data structures that our application has and do other more complicated things as well, so clearly this approach has some limitations.

So, we create a structure that will hold our application data and call it `AppState`. In our main, we'll store one instance of this structure into `main`'s stack (which doesn't go away during execution of our program) and will fill that with useful data. We'll start with pointers to some widgets whose visibility we want to control and we'll also store the color that user has selected.

The following changes will be made to the program:

- We need to restructure our menu-handling callback because we'll pass the application data structure as a parameter each time. This means that we cannot use the last parameter to pass the menu command/action anymore. This will also cause the enumeration to go away.
- The restructuring also makes it quite difficult to reuse the same callback



function to handle multiple signals, so we'll switch into having one callback function per signal and fill in the code for them later on.

- Some of our callbacks will switch visibility of widget trees. In order to get a pointer to the widget tree, we'll need to save pointers to the root widgets of the widget trees.
- We'll add an option to the main menu to switch visibility of the main toolbar.
- We'll add a specialised toolbar for searching.
- We'll demonstrate some way of printing debugging and informational messages.

Since most of our program changes now, we'll present the whole code for `hildon_helloworld-5.c` (yes, we're quickly becoming masters of the Hello Worlds). Be sure to experiment with the code as well.

```
/**
 * hildon_helloworld-5.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.
 *
 * Adds proper an application state and modifies all of the existing
 * functions to use application state. Also adds a Hildon Find
 * toolbar. Most of the existing code needs to be modified or split
 * up.
 */

#include <stdlib.h>
#include <hildon/hildon-program.h>
#include <hildon/hildon-color-button.h>
/* Pull in the Hildon Find toolbar declarations (NEW). */
#include <hildon/hildon-find-toolbar.h>

/* 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;
}
```

```

/* 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;

/**
 * The following functions more or less re-implement the same logic
 * as before, but have been modified to accept/work using the passed
 * ApplicationState. Some of them have also been renamed for
 * consistency.
 */

/**
 * 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();
}

/**
 * This will be implemented properly later.
 */
static void cbActionOpen(GtkWidget* widget, ApplicationState* app) {
    g_print("cbActionOpen invoked (feature unimplemented)\n");
}

/**
 * Placeholder for the file saving implementation.
 */
static void cbActionSave(GtkWidget* widget, ApplicationState* app) {
    g_print("cbActionSave invoked (feature unimplemented)\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");
    }
}

/**
 * Invoked when the user selects a color (or will cancel the dialog).
 */
static void cbActionColorChanged(HildonColorButton* colorButton,
                                ApplicationState* app) {

    g_assert(app != NULL);

```

```

g_print("cbActionColorChanged invoked\n");
hildon_color_button_get_color(colorButton, &app->currentColor);
g_print(" New color is r=%d g=%d b=%d\n", app->currentColor.red,
        app->currentColor.green, app->currentColor.blue);
}

/**
 * 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;
}

/**
 * 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();
    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;
}

/**
 * MODIFIED
 *
 * 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* 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");
    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), 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);

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

/**
 * There are main two changes compared to the previous version:
 * 1) There is now an application state (on the stack) that is passed
 *    to the functions that build up the GUI and to the callbacks.
 * 2) Two toolbars are created and added to the application.
 */
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;

    /* 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 the label widget. */
    label = gtk_label_new("Hello Hildon (with Hildon-search\n"
                          "and other tricks)!");

    /* 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));

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 4.3: Hello World as an proper application! (hildon\_helloworld-5.c)

When you're testing the program, be sure to notice how visibility of the toolbars affects the area available for the rest of the application. The label will always be centered inside the area that has been allocated to it.

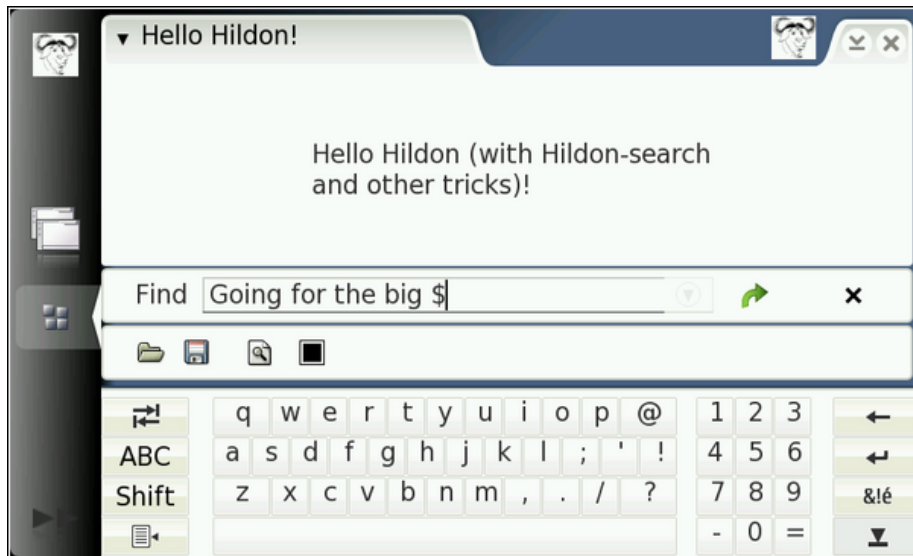


Figure 4.1: Our program with both toolbars and the VKB.

The Virtual Keyboard (VKB from now on) will be displayed automatically with Hildon widgets whenever the user will activate a widget used to input text. On Internet Tablets, displaying the VKB will not be done if the device has a hardware keyboard.

## 4.4 Processing key events

In order to implement actions when the user will press the hardware buttons on target devices, we need to handle the button events somehow. We'll next demonstrate this by catching the fullscreen keypress and implement "manual" switching of the fullscreen state for our application. Implementing fullscreen support is much easier in real life than the following code, see the end of this section for an explanation. We'll do it "manually" so that we can demonstrate hardware key handling in a simple way.

We'll add an item into our main menu that can be used to switch into fullscreen mode. To get back from fullscreen, we'll need to learn how to capture keyboard events (the hardware button is implemented in GDK as a ordinary keyboard key press). We will process the key press so that it will toggle fullscreen mode on and off.

```
/**
 * hildon_helloworld-6.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.
 *
 * Continuing from helloworld-5, we add fullscreen toggling and key
 * press handling (to get back from fullscreen).
```

```

*
* Look for lines with "NEW" or "MODIFIED" in them.
*/

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

/**
 * NEW
 *
 * 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;
}

/**
 * NEW
 *
 * 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;
}

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

/**
 * MODIFIED
 *
 * Register the key-press-event handler to handle fullscreen
 * switching.
 */
int main(int argc, char** argv) {

    ApplicationState aState = {};

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

    /* 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));

    label = gtk_label_new("Hello Hildon (with fullscreen)\n");

    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 for keypresses inside GTK+ (NEW).
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 and exiting with success\n");

return EXIT_SUCCESS;
}

```

Listing 4.4: Fullscreen and keypress handling (hildon\_helloworld-6.c)



Fullscreen mode with main toolbar open

The point of the previous program was to demonstrate how you can catch keypresses. The topic in reality is quite complex because keypresses travel through so many different software layers. A keypress will first be detected and processed by the kernel input device driver. The kernel driver will then forward the event to the X server which will hand the event to the application that has the current window focus. Inside the application, the GDK layer will see the keypress and propagate it onwards as a GTK+ signal, at which point it might arrive at some callback (depending on which widget had the focus at the time of the keypress).

## 4.5 Adding File-dialogs

In order to let the user to select which file to open or save, we need some kind of a dialog to choose files. Instead of building our own (which would be counter productive even in GTK+), we'll use the one that is included in Hildon.

Since we'll want to create both kinds of file choosers (one for saving, the other for opening) we'll create yet another utility function which will display the dialog with the required style and return a filename. Note that the filename will be allocated by GTK+ and we'll need to free it after we're done with it.

```
/**
 * hildon_helloworld-7.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.
 *
 * This version adds a file chooser and fills up some code to the
 * "Open" and "Save" callbacks.
 *
 * 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>
/* We need HildonFileChooserDialog (NEW).
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>

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

/**
 * NEW
 *
 * 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;
}

/**
 * MODIFIED
 *
 * Asks the user to select a file to open.
 */
static void cbActionOpen(GtkWidget* widget, ApplicationState* app) {

    gchar* filename = NULL;

    g_assert(app != NULL);

    g_print("cbActionOpen invoked\n");

    /* Ask the user to select a file to open. */
    filename = runFileChooser(app, GTK_FILE_CHOOSER_ACTION_OPEN);
    if (filename) {
        g_print(" you chose to load file '%s'\n", filename);
        /* Process the file load .. (will be implemented later). */

        /* Free up the filename when it's not needed anymore. */
        g_free(filename);
        filename = NULL;
    } else {

```

```

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

/**
 * MODIFIED
 *
 * Asks the user to select a file to save into (same logic as in
 * cbActionOpen above, just the style of the dialog is different).
 */
static void cbActionSave(GtkWidget* widget, ApplicationState* app) {
    gchar* filename = NULL;

    g_assert(app != NULL);

    g_print("cbActionSave invoked\n");

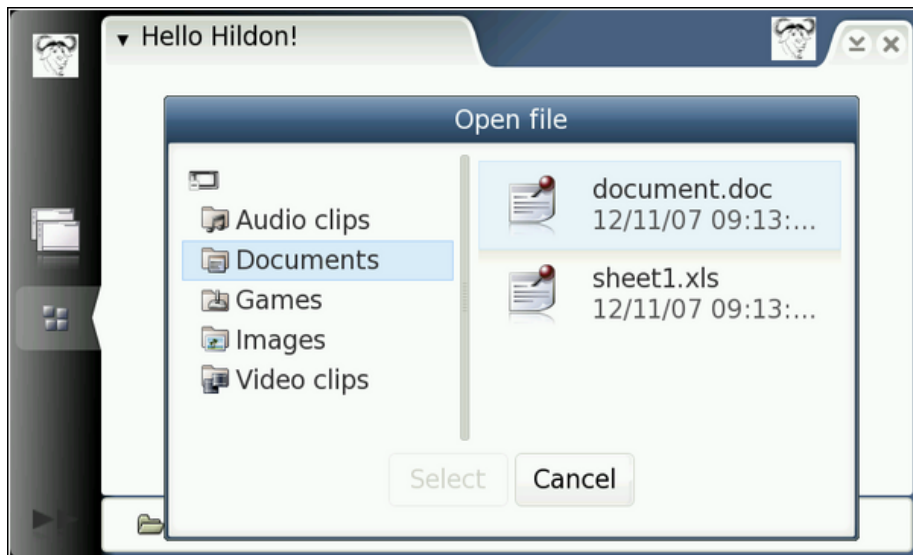
    filename = runFileChooser(app, GTK_FILE_CHOOSER_ACTION_SAVE);
    if (filename) {
        g_print(" you chose to save into '%s'\n", filename);
        /* Process saving .. */

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

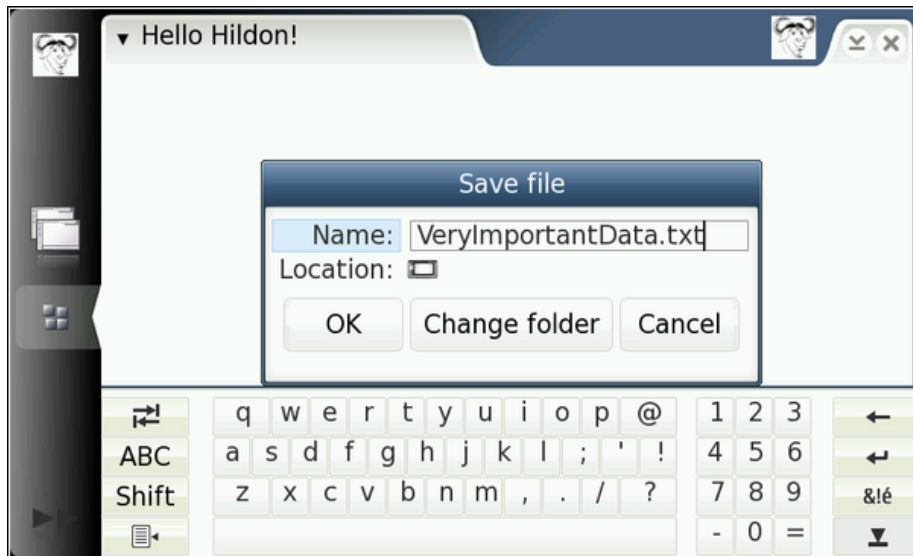
```

Listing 4.5: Adding file choosing dialogs (hildon\_helloworld-7.c)

In order to use the file chooser widget, you'll also need to link against the hildon-fm-2 library (using pkg-config, just like you do with hildon-1).



Dialog for opening files



Dialog for selecting filename to save to

You really need to experiment with the dialogs to see what they offer. Considering the limited screen estate available on an Internet Tablet, the dialogs manage to provide quite a lot.

## 4.6 Where to go next?

Needless to say both GTK+ and Hildon are full of useful widgets that you should start exploring and using.

Links into deeper recesses of GTK+ and Hildon

- The master API documentation index at [maemo.org](http://maemo.org). It includes to the versions of documentation which are used in maemo SDK, so it should be the preferred starting point when looking for more information.
- The GTK+ 2.0 tutorial ([gtk.org](http://gtk.org)) is definitely worth going through. It is quite large and tries to be comprehensive but at the same time fails to be current with respect the GTK+ API. Please try to verify that the API-functions that you see used in the tutorial are current by checking the GTK+ API documentation. The obsolete functions are marked as "deprecated" in their documentation part. Not many people do this and that leads to obsolete functions being used quite liberally.
- The GTK+ reference can be found at [maemo.org](http://maemo.org).
- There is no current GDK tutorial (unfortunate).
- There is however the GDK API reference at [maemo.org](http://maemo.org). Intermingled with the documentation there are some examples as well.

- The GTK+ and GDK library source distribution contains a lot of example programs which are kept almost to date (at least you see the modification timestamps on when of their last modifications ;-).
- Reading the header files has helped more than once as well, so get acquainted with `/usr/include/gtk-2.0/subdirectories` as well as `/usr/include/hildon-* directories`.
- When all else fails, you can read the source code of GTK+ (which is written in a clear and consistent manner, albeit without too many comments) and of course the source code for most of the Hildon widgets is available with `apt-get source hildon-1`.

If you know that you will be working with Hildon and GTK+ a lot, you should join the respective mailing lists (**`maemo-developers@maemo.org`** and **`gtk-app-devel@gnome.org`**) and read the archives of mailing lists. Nowadays search engines are able to locate snippets of valuable information from these lists as well.

## 4.7 Conclusions

It will take you a while to get used to the GTK+ way of doing graphics (or indeed anything). However, time spent learning will pay off quite soon since the API for most parts is quite simple.

You just need to remember to check and recheck that API functions that you use. Try to avoid deprecated functions as your software may be compiled with flags that will disable the deprecated functions altogether and then you'll need to fix a lot of problems (and learn the new API anyway).

Rules of thumb with GTK+:

- Familiarise yourself with widget inheritance trees. Sometimes the function you're looking for is really implemented at a higher level up the inheritance.
- There are probably multiple ways of achieving what you're trying to do. Try to use the simplest interface that is not obsolete (not always easy).
- Select your pointer types according to their use. If you will be passing your widgets to functions that accept mostly `GtkWidget*`-types, it will be easier to type and read if you use a `GtkWidget*`-pointer to start with (or using `GtkToolItem*` as we did with our example).
- If you don't think that the API documentation is complete, read the GTK+ source (it's not that bad, compared to many other open source projects).
- Ask your colleagues
- Read what others have used (but again be very careful about obsolete code).

Good luck!

## Chapter 5

# Support Libraries

### 5.1 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.

### 5.2 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](http://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 successfully, 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));

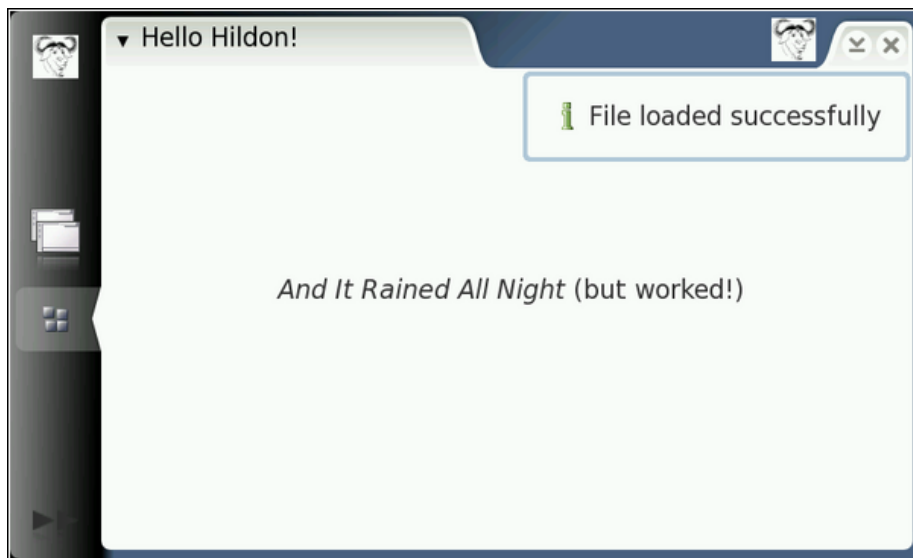
/* Register a callback to handle key presses. */
g_signal_connect(G_OBJECT(aState.window), "key_press_event",
                 G_CALLBACK(cbKeyPressed), &aState);

g_print("main: calling gtk_main\n");
gtk_main();
g_print("main: returned from gtk_main and exiting with success\n");

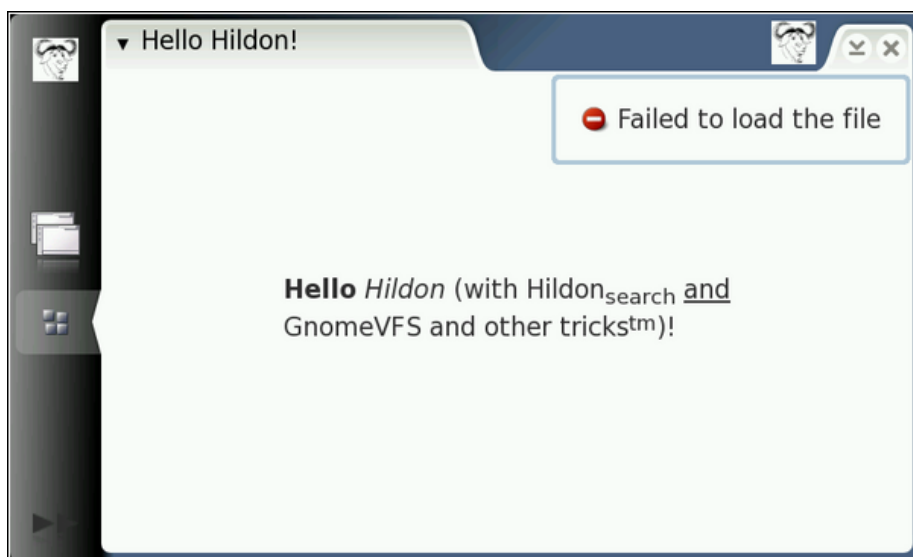
return EXIT_SUCCESS;
}

```

Listing 5.1: 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 "<b>Hello world</b>" > 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](http://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.

### 5.3 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").

### 5.4 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.

## 5.5 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](http://maemo.org).

## 5.6 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) {

    GtkWidget* toolbar = NULL;
    GtkWidget* tbOpen = NULL;
    GtkWidget* 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 5.2: 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.

## Chapter 6

# GNU Autotools

### 6.1 Introduction to GNU autotools

If you already feel comfortable using *GNU autotools* and creating your own *autotools* configuration files, feel free to skip this chapter.

Creating portable software written in the C language has historically been challenging. The portability issues are not restricted just to the differences between binary architectures, but also encompass differences in system libraries and different implementations of UNIX APIs in different systems. This chapter introduces GNU autotools as one solution for managing this complexity. There are also other tools available but you are most likely to encounter autotoolized projects and also Debian packaging supports autotools without too much effort.

### 6.2 Brief history of managing portability

When discussing portability challenges with C code, various issues will crop up:

- Same library function has differing prototypes in different UNIX-style systems.
- Same library function has differing implementations between systems (this is a hard problem).
- Same function can be found in different libraries in different UNIX systems, and because of this, we need to modify our linker parameters when we're making the final linking of our project.
- Besides these not so obvious problems, there is also of course the problem of potentially different architecture and hence different sizes for the standard C data types. This problem is best fixed by using GLib (in our case), so it will not be covered here.
- GLib also provides the necessary macros to do endianness detection and conversion.

The historical solutions to solve these problems can be roughly grouped as follows:



- The whole software is built with one big shell script which will try to pass the right options to different compilers and tools. Most of these scripts have luckily remained in-house and never seen the light of the world. They are very hard to maintain.
- Makefiles (GNU or other kind) allow some automation in building but do not directly solve any of the portability issues. The classical solution was to ship various makefiles already tailored for different UNIX systems and it was the end-user's responsibility to select the appropriate **Makefile** to use for their system. This quite often also required editing the **Makefile**, so the end-user needed to be knowledgeable in UNIX programming tools. A good example of this style is found in the game nethack (if you can find a suitably old version).
- We can automate some parts of the **Makefile** selection above by a suitable script that tries to guess the system on which it's running and select a suitable prepared **Makefile**. Maintaining such scripts is quite hard as there were historically so many different kinds of systems (different library versions, different compilers, and so on).
- We can go one step further, and automate the creation of Makefiles from such guessing script. Such scripts are normally called **configure** or **Configure** and are marked executable for your convenience. The name of the script doesn't automatically mean that the script is related to GNU autotools.
- Two major branches of such master configure scripts evolved, each operating a bit differently and suiting differing needs.
- These two scripts and a third guessing script were then combined into GNU autoconf. Since this happened many many years ago, most of the historical code has already been purged from GNU autoconf.

Besides autoconf, it became evident that a more general **Makefile** generation tool could be useful as part of the whole process. This is how GNU automake was born. It can also be used outside autoconf. We'll cover automake a bit later, but first we'll take a look at a simple autoconf configuration file (historically called as driver, but this is an obsolete term now).

### 6.3 GNU autoconf

autoconf is a tool that will use the GNU m4 macro preprocessor to process your configuration file and output a shell script based on the macros used in your file. Anything that m4 doesn't recognise as a macro will be passed verbatim to the output script. This means that you can include almost any shell script fragments that you want into your **configure.ac** (the modern name for the default configuration file for autoconf).

We'll start by a simple example and see how the basic configuration file works. Then we'll cover some limitations of *m4* syntax and how to avoid problems with the syntax.

```

# Specify the "application name" and application version
AC_INIT(hello, version-0.1)

# Since autoconf will pass through anything that it doesn't recognize
# into the final script ('configure'), we can use any valid shell
# statements here. Note that you should restrict your shell to
# standard features that are available in all UNIX shells, but in our
# case, we're content with the most used shell on Linux systems
# (bash).
echo -e "\n\nHello from configure (using echo)!\n\n"

# We can use a macro for this messages. This is much preferred as it
# is more portable.
AC_MSG_NOTICE([Hello from configure using msg-notice!])

# Check that the C Compiler works.
AC_PROG_CC

# Check what is the AWK-program on our system (and that one exists).
AC_PROG_AWK

# Check whether the 'cos' function can be found in library 'm'
# (standard C math library).
AC_CHECK_LIB(m, cos)

# Check for presence of system header 'unistd.h'.
# This will also test a lot of other system include files (it is
# semi-intelligent in determining which ones are required).
AC_CHECK_HEADER(unistd.h)

# You can also check for multiple system headers at the same time,
# but notice the different name of the test macro for this (plural).
AC_CHECK_HEADERS([math.h stdio.h])

# A way to implement conditional logic based on header file presence
# (we don't have a b0rk.h in our system).
AC_CHECK_HEADER(b0rk.h, [echo "b0rk.h present in system"], \
    [echo "b0rk.h not present in system"])

echo "But that doesn't stop us from continuing!"

echo "Directory to install binaries in is '$bindir'"
echo "Directory under which data files go is '$datadir'"
echo "For more variables, check 'config.log' after running configure"
echo "CFLAGS is '$CFLAGS'"
echo "LDFLAGS is '$LDFLAGS'"
echo "LIBS is '$LIBS'"
echo "CC is '$CC'"
echo "AWK is '$AWK'"

```

Listing 6.1: Contents of autoconf-automake/example1/configure.ac

The listing is verbosely commented, so it should be pretty self-evident what the different macros do. The macros that test for a feature or an include file will normally cause the generated configure script to generate small C code test programs that will be run as part of the configuration process. If these programs run successfully, the relevant test will succeed and the configuration process will continue to the next test.

The following convention holds with respect to the names of macros that are commonly used and available:

AC\_\* A macro that is included in autoconf or is meant for it.

AM\_\* A macro that is meant for automake.

Others The autoconf system can be expanded by writing your own macros which can be stored in your directory. Also some development packages install new macros for you to use. We'll see one example later on.

Let's now run autoconf. Without any parameters it will read **configure.ac** by default. If **configure.ac** doesn't exist, it will try to read **configure.in** instead. Note that using the name **configure.in** is considered obsolete and reason will become clear later on.

```
[sbox-DIABLO_X86: ~/example1] > autoconf
[sbox-DIABLO_X86: ~/example1] > ls -l
total 112
drwxr-xr-x 2 user user 4096 Sep 16 05:14 autom4te.cache
-rwxrwxr-x 1 user user 98683 Sep 16 05:14 configure
-rw-r--r-- 1 user user 1825 Sep 15 17:23 configure.ac
[sbox-DIABLO_X86: ~/example1] > ./configure

Hello from configure (using echo)!

configure: Hello from configure using msg-notice!
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ANSI C... none needed
checking for gawk... gawk
checking for cos in -lm... yes
checking how to run the C preprocessor... gcc -E
checking for egrep... grep -E
checking for ANSI C header files... yes
checking for sys/types.h... yes
checking for sys/stat.h... yes
checking for stdlib.h... yes
checking for string.h... yes
checking for memory.h... yes
checking for strings.h... yes
checking for inttypes.h... yes
checking for stdint.h... yes
checking for unistd.h... yes
checking for unistd.h... (cached) yes
checking math.h usability... yes
checking math.h presence... yes
checking for math.h... yes
checking stdio.h usability... yes
checking stdio.h presence... yes
checking for stdio.h... yes
checking b0rk.h usability... no
checking b0rk.h presence... no
checking for b0rk.h... no
b0rk.h not present in system
But that doesn't stop us from continuing!
Directory to install binaries in is '${exec_prefix}/bin'
Directory under which data files go is '${prefix}/share'
For more variables, check 'config.log' after running configure
CFLAGS is '-g -O2'
LDFLAGS is ''
LIBS is '-lm '
CC is 'gcc'
AWK is 'gawk'
```

Running autoconf

Autoconf will not output information about its progress. Only errors are output on stdout normally. Instead, it will create a new script in the same

directory called **configure** and will set it as executable for you.

The **configure** script is the result of all of the macro processing. If you look at the generated file with an editor or by using **less**, you'll note that it contains a lot of shell code.

Unless you are experienced in reading convoluted shell code, it is best not to try to understand what is attempted at the various stages. Normally we don't try fix or modify the generated file, since it would be overwritten anyway the next time that someone will run **autoconf**.

When we execute the generated script, you will see the output of various parts in the order that they were used in the configuration file.

Note that **autoconf** (because of **m4**) is an imperative language which means that it will execute the commands one by one when it detects them. This is in contrast to declarative languages, like **Makefiles**.

M4-syntax notes:

- Public **autoconf** M4 macros all start with **A[CST]\_\***.
- Private macros start with an underscore, but they shouldn't be used since they will change from one **autoconf** version to the other (using undocumented features is bad style anyway).
- **m4** uses **[** and **]** to quote arguments to functions, but in most cases we can leave them out as well. It's best to avoid using braces unless the macro doesn't seem to work properly otherwise. When writing new macros, using braces will become more important, but this material does not cover creating custom macros.
- If you absolutely need to pass braces to the generated script, you have three choices:
  - **@<:@** is same as **[**, and **@>:@** is same as **]**
  - **[[ ]]** will expand into **[ ]** (most of time)
  - avoid **[** and **]** (most command line tools and shell don't really require them)
- Since **m4** will use braces for its own needs, we cannot use the **[** command to test things in our scripts, but instead must use **test** (which is more clear anyway). This is why we escape the braces with the rules given above if we really need them to be output into the generated shell script.

If you introduce bad shell syntax into your configuration file, the bad syntax will cause errors only when you will run the generated script file (not when **autoconf** generates the script). In general, **autoconf** will almost always succeed, but you might find that the generated script will not. It's not always simple to know which error in the shell script corresponds to which line in the original **configure.ac** but you will learn with experience.

You might notice a line that reports the result of testing for **unistd.h**. It will appear twice, the first time because it's the default test to run whenever we start testing for headers and a second time because we test for it explicitly. The second test output contains text (cached), which means that the test has been already run and the result has been saved into a cache (the mysterious **autom4te.cache** directory). This means that for large projects which might do

the same tests over and over, the tests are only run once and this will make running the script quite a bit faster.

The last lines output above contain the values of variables. When the configure script runs, it will automatically create shell variables for you that you can use in your shell code fragments. The macros for checking what programs are available for compiling should illustrate that point. Here we used `awk` as an example.

The configure script will take the initial values for variables from your environment but also contains a lot of options that you can give to your script and using those will introduce new variables that you can also use. Anyone compiling modern open source projects will be familiar with options like `--prefix` and other similar ones. Both of these cases are illustrated below:

```
[sbox-DIABLO_X86: ~/example1] > CFLAGS='-O2 -Wall' ./configure --prefix=/usr
...
Directory to install binaries in is '${exec_prefix}/bin'
Directory under which data files go is '${prefix}/share'
For more variables, check 'config.log' after running configure
CFLAGS is '-O2 -Wall'
LDFLAGS is ''
LIBS is '-lm '
CC is 'gcc'
AWK is 'gawk'
```

Modifying configure runs

It might not seem that giving the prefix changes anything, but this is because the shell doesn't expand the value in this particular case. It would expand the value later on if we'd use the variable in our script for doing something. If you want to see some effect, you can try passing the `--datadir` option (because we print that out explicitly).

If you're interested in what other variables are available for configure, you can read the generated **config.log** file, since the variables are listed at the end of that file.

## 6.4 Substitutions

Besides creating the configure script, autoconf can do other useful things as well. Some people say that autoconf is at least as powerful as emacs, if not more so! Unfortunately with all this power comes awfully lot of complexity when things don't quite work.

Sometimes it is useful to use our variables within text files that are otherwise non-related to our **configure.ac**. These might be configuration files or files that will be used in some part of the building process later on. For this, autoconf provides a mechanism called *substitution*. There is a special macro that will read in an external file, replace all instances of variable names in it, and then store the resulting file into a new file. The convention in naming the input files is to add a suffix `.in` to the names. The name of generated output file will be the same, but without this suffix. Note that the substitution will be done when you run the generated configure script, not when autoconf is run.

The generated configure script will replace all occurrences of the variable name surrounded with '@' characters with the variable value when it reads through each of the input files.

This is best illustrated with a small example. The input file contents are listed after the autoconf configuration file. In this example we'll only do substitution for one file, but it's also possible to process multiple files using the substitution mechanism.

```
# An example showing how to use a variable-based substitution.

AC_INIT(hello, version-0.1)

AC_PROG_CC
AC_PROG_AWK
AC_CHECK_HEADER(unistd.h)
AC_CHECK_HEADERS([math.h stdio.h])
AC_CHECK_HEADER(b0rk.h, [echo "b0rk.h present in system"], \
    [echo "b0rk.h not present in system"])

echo "But that doesn't stop us from continuing!"

# Run the test-output.txt(.in) through autoconf substitution logic.
AC_OUTPUT(test-output.txt)
```

Listing 6.2: Contents of autoconf-automake/example2/configure.ac

```
This file will go through autoconf variable substitution.

The output file will be named as 'test-output.txt' (the '.in'-suffix
is stripped).

All names surrounded by '@'-characters will be replaced by the values
of the variables (if present) by the configure script when it is run
by end user.

Prefix: @prefix@
C compiler: @CC@

This is a name for which there is no variable.
Stuff: @stuff@
```

Listing 6.3: Contents of autoconf-automake/example2/test-output.txt.in

We then run autoconf and configure:

```

[sbox-DIABLO_X86: ~/example2] > autoconf
[sbox-DIABLO_X86: ~/example2] > ./configure
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ANSI C... none needed
checking for gawk... gawk
checking how to run the C preprocessor... gcc -E
checking for egrep... grep -E
checking for ANSI C header files... yes
checking for sys/types.h... yes
checking for sys/stat.h... yes
checking for stdlib.h... yes
checking for string.h... yes
checking for memory.h... yes
checking for strings.h... yes
checking for inttypes.h... yes
checking for stdint.h... yes
checking for unistd.h... yes
checking for unistd.h... (cached) yes
checking math.h usability... yes
checking math.h presence... yes
checking for math.h... yes
checking stdio.h usability... yes
checking stdio.h presence... yes
checking for stdio.h... yes
checking b0rk.h usability... no
checking b0rk.h presence... no
checking for b0rk.h... no
b0rk.h not present in system
But that doesn't stop us from continuing!
configure: creating ./config.status
config.status: creating test-output.txt
[sbox-DIABLO_X86: ~/example2] > cat test-output.txt
This file will go through autoconf variable substitution.

The output file will be named as 'test-output.txt' (the '.in'-suffix
is stripped).

All names surrounded by '@'-characters will be replaced by the values
of the variables (if present) by the configure script when it is run
by end user.

Prefix: /usr/local
C compiler: gcc

This is a name for which there is no variable.
Stuff: @stuff@

```

File contents substitution example

We'll use this feature later on. When you run your own version of the file, you might notice the creation of file called **config.status**. It is the file that actually does the substitution for external files, so if your configuration is otherwise complex and you only want to re-run the substitution of the output files, you can run the **config.status** script.

## 6.5 Introducing automake

We'll next create a small project that consists of yet another hello world. This time we have a file implementing the application (main), a header file describing the API (printHello()) and the implementation of the function.

As it happens, GNU automake is designed so that it can be easily integrated into autoconf. We'll utilise this in the next example so that we don't need to write our **Makefile** by hand anymore. Instead, the **Makefile** will be generated by the configure script and it will contain the necessary settings for the system on which **configure** is run.

In order for this to work, we'll need two things:

- We need a configuration file for automake (conventionally `Makefile.am`).
- We need to tell our autoconf that it should create `Makefile.in` based on `Makefile.am` by using automake.

We'll start with the autoconf configuration file:

```
# Any source file name related to our project is ok here.
AC_INIT(helloapp, 0.1)

# We're using automake, so we init it next. The name of the macro
# starts with 'AM' which means that it is related to automake ('AC'
# is related to autoconf).
# Initiating automake means more or less generating the .in file from
# the .am file although it can also be generated at other steps.
AM_INIT_AUTOMAKE

# Compiler check.
AC_PROG_CC
# Check for 'install' program.
AC_PROG_INSTALL
# Generate the Makefile from Makefile.in (using substitution logic).
AC_OUTPUT(Makefile)
```

Listing 6.4: Contents of `autoconf-automake/example3/configure.ac`

Then we present the configuration file for automake:

```
# Automake rule primer:
# 1) Left side_ tells what kind of target this will be.
# 2) _right side tells what kind of dependencies are listed.
#
# As an example, below:
# 1) bin = binaries
# 2) PROGRAMS lists the programs to generate Makefile.ins for.
bin_PROGRAMS = helloapp

# Listing source dependencies:
#
# The left side_ gives the name of the application to which the
# dependencies are related to.
# _right side gives again the type of dependencies.
#
# Here we then list the source files that are necessary to build the
# helloapp -binary.
helloapp_SOURCES = helloapp.c hello.c hello.h

# For other files that cannot be automatically deduced by automake,
# you need to use the EXTRA_DIST rule which should list the files
# that should be included. Files can also be in other directories or
# even whole directories can be included this way (not recommended).
#
# EXTRA_DIST = some.service.file.service some.desktop.file.desktop
```



---

Listing 6.5: Contents of autoconf-automake/example3/Makefile.am

This material tries to introduce just enough of automake for it to be useful for small projects. Because of this, detailed syntax of Makefile.am is not explained. Based on the above description automake will know what kind of Makefile.in to create and autoconf will take it over from there and fill in the missing pieces.

The source files for the project are as simple as possible, but note the implementation of `printHello`. Is there something fishy going on?

```
/**
 * This maemo code example is licensed under a MIT-style license,
 * that can be found in the file called "License" in the same
 * directory as this file.
 * Copyright (c) 2007-2008 Nokia Corporation. All rights reserved.
 */

#include <stdlib.h>
#include "hello.h"

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

    printHello();

    return EXIT_SUCCESS;
}
```

Listing 6.6: The main application in autoconf-automake/example3/helloapp.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.
 */

#ifndef INCLUDE_HELLO_H
#define INCLUDE_HELLO_H

extern void printHello(void);

#endif
```

Listing 6.7: Prototypes of projects in autoconf-automake/example3/hello.h

```
/**
 * This maemo code example is licensed under a MIT-style license,
 * that can be found in the file called "License" in the same
 * directory as this file.
 * Copyright (c) 2007-2008 Nokia Corporation. All rights reserved.
 */

#include <stdio.h>
#include "hello.h"

void printHello(void) {
    /* Note that these are available as defines now. */
    printf("( PACKAGE " " VERSION " )\n");
    printf("Hello world!\n");
}
```

```
}
```

Listing 6.8: Implementation of printHello in autoconf-automake/example3/hello.c

The `PACKAGE` and `VERSION` defines will be passed to the build process automatically and we'll see their use later. For now, we're armed and dangerous, so let's try our luck:

```
[sbox-DIABLO_X86: ~/example3] > autoconf
configure.ac:10: error: possibly undefined macro: AM_INIT_AUTOMAKE
If this token and others are legitimate, please use m4_pattern_allow.
See the Autoconf documentation.
```

Uh oh

Seems that we were a bit too dangerous. `autoconf` is complaining about a macro for which it cannot find a definition (actually `m4` is the program that does the complaining). What gives? The problem is that by default, `autoconf` only knows about built-in macros. If we want to use a macro for integration or a macro that comes with another package, we need to tell `autoconf` about it. Luckily this process is quite painless.

We'll need to create a local **aclocal.m4** file into the same directory with our **configure.ac**. When it will start, `autoconf` will read this file and it's enough to put the necessary macros there.

We'll use an utility program called **aclocal**, which will scan our **configure.ac** and copy the relevant macros into the local **aclocal.m4**. The directory for all the extension macros is usually **/usr/share/aclocal/** which you might want to check out at some point.

We'll run **aclocal**, and then try to run `autoconf` again. Note that the messages that are introduced into our process from now on are inevitable since some macros use obsolete features or have incomplete syntax and thus trigger warnings. There is no easy solution to this other than to fix the macros themselves.

```
[sbox-DIABLO_X86: ~/example3] > aclocal
/scratchbox/tools/share/aclocal/pkg.m4:5: warning:
underquoted definition of PKG_CHECK_MODULES
run info '(automake)Extending aclocal' or see
http://sources.redhat.com/automake/automake.html#Extending%20aclocal
/usr/share/aclocal/pkg.m4:5: warning:
underquoted definition of PKG_CHECK_MODULES
/usr/share/aclocal/gconf-2.m4:8: warning:
underquoted definition of AM_GCONF_SOURCE_2
/usr/share/aclocal/audiofile.m4:12: warning:
underquoted definition of AM_PATH_AUDIOFILE
[sbox-DIABLO_X86: ~/example3] > autoconf
[sbox-DIABLO_X86: ~/example3] > ./configure
configure: error: cannot find install-sh or install.sh in
~/example3 ~/ ../..
```

Much better, but we're not quite there yet.

If you now list the contents of the directory, you might be wondering where is the **Makefile.in**? We need to run `automake` manually, so that the file will be created. At the same time we also need to introduce the missing files to the directory (like the **install.sh** that **configure** seems to complain about).

This is done by executing `automake -ac`, which will create the **Makefile.in** and also copy the missing files into their proper places. **This step will also copy**

a file called **COPYING** into your directory which by default will contain the **GPL**. So, if you're going to distribute your software under some other license, this might be the correct moment to replace the license file with the one that you really want to use.

```
[sbox-DIABLO_X86: ~/example3] > automake -ac
configure.ac: installing './install-sh'
configure.ac: installing './missing'
Makefile.am: installing './depcomp'
[sbox-DIABLO_X86: ~/example3] > ./configure
checking for a BSD-compatible install... /scratchbox/tools/bin/install -c
checking whether build environment is sane... yes
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ANSI C... none needed
checking for style of include used by make... GNU
checking dependency style of gcc... gcc3
checking for a BSD-compatible install... /scratchbox/tools/bin/install -c
configure: creating ./config.status
config.status: creating Makefile
config.status: executing depfiles commands
```

Configure completes

Notice the second to last line of the output which tells us that **autoconf** just created **Makefile** for us (based on the **Makefile.in** that **automake** created).

You might grow tired of doing all these steps manually each time when you want to make sure that all generated files are really generated. You are not alone. Most developers will create a script called **autogen.sh** which will implement the necessary bootstrap-procedures for them. Below is a file that is suitable for our use. Real projects might have more steps because of localisation and other requirements.

```
#!/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 6.9: A typical simple autogen.sh (autoconf-automake/example3/autogen.sh)

You might notice the commented line with touch and wonder why it is commented. There is a target called `distcheck` that automake will create in our **Makefile** and this target checks whether the distribution tarball contains all the necessary files. The files listed on the touch-line are necessary (even if empty), so you'll need to create those at some point. Without these files, the penultimate **Makefile** will complain when you run the `distcheck`-target.

We'll now build our project and test it out:

```
[sbox-DIABLO_X86: ~/example3] > make
if gcc -DPACKAGE_NAME=\"helloapp\" -DPACKAGE_TARNAME=\"helloapp\"
-DPACKAGE_VERSION=\"0.1\" -DPACKAGE_STRING=\"helloapp 0.1\"
-DPACKAGE_BUGREPORT=\"\" -DPACKAGE=\"helloapp\" -DVERSION=\"0.1\"
-I. -Iexample3 -g -O2 -MT helloapp.o -MD -MP -MF ".deps/helloapp.Tpo"
-c -o helloapp.o helloapp.c; \ then \
mv -f ".deps/helloapp.Tpo" ".deps/helloapp.Po";
else rm -f ".deps/helloapp.Tpo"; exit 1;
fi
if gcc -DPACKAGE_NAME=\"helloapp\" -DPACKAGE_TARNAME=\"helloapp\"
-DPACKAGE_VERSION=\"0.1\" -DPACKAGE_STRING=\"helloapp 0.1\"
-DPACKAGE_BUGREPORT=\"\" -DPACKAGE=\"helloapp\" -DVERSION=\"0.1\"
-I. -Iexample3 -g -O2 -MT hello.o -MD -MP -MF ".deps/hello.Tpo"
-c -o hello.o hello.c; \ then \
mv -f ".deps/hello.Tpo" ".deps/hello.Po";
else rm -f ".deps/hello.Tpo"; exit 1;
fi
gcc -g -O2 -o helloapp helloapp.o hello.o
[sbox-DIABLO_X86: ~/example3] > ./helloapp
(helloapp 0.1)
Hello world!
[sbox-DIABLO_X86: ~/example3] > make clean
test -z "helloapp" || rm -f helloapp
rm -f *.o
```

Building, PACKAGE and VERSION defines and cleaning

## 6.6 Checking for distribution sanity

The generated **Makefile** contains various targets that can be used when creating distribution tarballs (tar-files containing the source code and necessary files to build the software). The most important of these is the `dist`-target, which will by default create a **.tar.gz**-file out of your source including the **configure**-script and other necessary files (which are specified in **Makefile.am**).

To test whether it is possible to build your software from such distribution tarball, execute the `distcheck`-target. It will first create a distribution tarball, then extract it in a new subdirectory, run **configure** there and try to build the software with default make target. If it fails, you'll get the relevant error.

It's recommended to do the `distcheck` target each time before doing a `dist` target so that you can be sure that the distribution tarball can be used outside

your source tree. This step is especially critical when we'll start making Debian packages later on.

## 6.7 Cleaning up

For your convenience, the **example3** directory also includes a script called **antigen.sh**, which will try its best to get rid of all generated files (you'll need to **autogen.sh** the project afterwards).

Having a cleanup script is not very common in open source projects, but it's especially useful when starting autotools development, as it allows you test your toolchain easily from scratch.

The contents of **antigen.sh** which is suitable for simple projects:

```
#!/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 6.10: An utility script to clean up all generated files (autoconf-automake/example3/antigen.sh)

## 6.8 Integration with pkg-config

The last part that we cover for autoconf is how to integrate **pkg-config** support into your projects when using **configure.ac**.

**pkg-config** comes with a macro package (**pkg.m4**) which contains some useful macros for integration. The best documentation for these can be found in the **pkg-config** manual pages.

We'll use only one, **PKG\_CHECK\_MODULES** which should be used like this:

```
# 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"

```

Listing 6.11: Part of configure.ac for pkg-config integration

The proper placing for this code is after all the AC\_PROG\_\*-checks and before the AC\_OUTPUT-macros (so that the build flags may affect the substituted files).

You should also check the validity of the package names by using `pkg-config --list-all` to make sure you don't try to get the wrong library information.

## Chapter 7

# Integration with the Application Framework

### 7.1 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 activatable 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.

### 7.2 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 7.1: 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.hhwX`) (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.

## 7.3 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 7.2: 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).

## 7.4 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 7.3: 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.

## 7.5 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 7.4: 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 7.5: Part of `configure.ac` that will cause the templates to be converted into proper files

## 7.6 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.

## Chapter 8

# Packaging Applications

### 8.1 Creating Debian packages

This chapter will introduce you to the Debian packaging system. We will restrict ourselves to cover the tools that are of immediate interest when creating new packages. Issues like building your own package repositories are not covered here, but there is ample information available with Internet search engines.

### 8.2 Packaging Basics

Most modern Linux distributions utilise some kind of packaging system. Two of the most popular ones are the Red Hat package Manager and Debian's dpkg-based system (although it's mainly popular because of a tool called *apt-get*).

All packaging systems strive to solve the same problems. Mainly, how to keep track which file belongs to which software package, so that a group of files may be upgraded in a coherent fashion, or removed all together. Keeping track of files necessitates some kind of local database and packaging systems are quite different in which technical solutions they select.

Before packaging systems existed on Linux, it was common to distribute source code as source tarballs (`make dist` makes it easy now, but back then it was a bit more difficult). This made installing software quite easy when you knew the tools that were used in building the software, but for a novice, it was sometimes painful. Removing the software was still quite hard, since targets like `uninstall` weren't common.

Slackware was the first Linux distribution to think about developing a common mechanism for software distribution, and they chose to distribute software as binary packages (this is what most distributions do now). You could of course get the source code for the software as well, but ordinary users weren't so much interested in source code, but wanted to get the software running. Computers were somewhat slower back then as well, so compiling software sometimes took long times (2.0 series kernel building took over 24 hours). Hence, the idea that one could build the binaries once, collect the relevant configuration and manual files together and then distribute them, was seen as an improvement.

Slackware used tarballs as the distribution mechanism. Each package tarball also contained a script that was executed on software install and this script could

then modify some system configuration files so that the software could run (set the runlevel related symbolic links for example).

Along came Red Hat, and they decided that the Slackware way was ugly and they could do better. They wrote their own packaging system (which also uses compressed binary archives, but not tar.gz). Red Hat's packaging system could relatively easily be used to rebuild the binaries if one so wanted or needed. The RPM system has a Berkeley DB-based local database in which the system keeps track of files that were installed from each package. The RPM system also provides some searching tools into the database as well as other features.

The Debian project started to think about package management at about the same time and implemented a packaging system that would also contain dependency information about what a package required to be installed beforehand before it could be installed safely. The reason why Debian didn't use RPMs was that RPM back then didn't support dependencies and wasn't available when Debian started thinking about these things. Nowadays both RPM and the Debian packaging system contain similar technical capabilities and it can be a matter of taste which one to choose when starting a new Linux distribution.

### 8.3 Dependencies

Soon after introducing the concept of software packages, it was evident that there needed to be some way to codify the rules within the packages which would guard against installing incompatible packages into the same system. The typical example was installing a mail server when there were multiple different choices for one (there have been alternatives to *sendmail* for many years). Suppose you wanted to install an alternative mail server and you already had *sendmail* installed. Since both of the services would access the same directory paths (mail spools), this would create chaos. They could not listen to the same TCP port either (not on the same IP address at least). This would cause problems to the administrator, so a system for describing package conflicts was developed. Soon after this, people noticed that applications started to use shared libraries and it didn't make sense to link libraries in static form into binaries, so a need for describing dependencies between installed packages arose.

In all modern Linux distributions it is the dependencies and their quality that determine how easy or hard it is for the busy system administrator to manage a Linux server. This applies to the desktop arena as well.

We will cover the syntax and operation of the Debian package dependencies and conflicts but other packaging systems have similar mechanisms.

### 8.4 Packaging infrastructure

The one thing that set apart the Debian packaging tools for a long time was a program called *apt-get*. It does not install software packages into the system, neither does it modify the local database. It is a program that reads package dependency and conflict information and tries to solve them in an intelligent way so that users may install or remove the packages that they want to.

One thing that makes apt-get special is its support for multiple ways of accessing repositories of packages (physical storage for packages in Debian). One of the most popular mechanism to use apt-get is the http-method(ftp being available as well). This means that when installing a package (which you don't have to download first), apt-get will first do some calculations and tell you which other packages (and their versions) need to be installed so that the original (and other) packages' dependencies won't be broken. After pressing ENTER, apt-get will download the necessary packages and invoke the low-level package management tool(dpkg) in the order necessary to complete the user's wishes.

One point needs to be re-iterated: writing correct dependencies (not too strict, but complete) is critical for tools like apt-get to work properly. Otherwise things will break and users will be unhappy. This is often the hardest thing to get right.

Some important program options and locations:

- `/etc/apt/sources.list` : a file listing the repositories and methods for apt-get.
- `/var/cache/apt/archives/` : apt-get downloads .deb-files here.
- `/var/lib/dpkg/` : location of the local package database (used with dpkg).
- `/var/lib/dpkg/info/` : control files for already installed packages.
- `apt-get install foo` : install binary package foo.
- `apt-get remove foo` : remove binary package foo.
- `apt-get source foo` : download source package for package foo.
- `apt-get build-dep foo` : download and install the packages required to build foo.
- `dpkg-source -x foo_version-revision.dsc` : extract the source package into a directory.
- `cd foo-version*; dpkg-buildpackage -rfakeroot -b` : build a binary package from a previously extracted directory.
- `dpkg -i foo_version-revision_arch.deb` : install the binary package file.
- `dpkg -l` : list installed packages.
- `dpkg -L foo` : list installed files of package foo.
- `dpkg -S /path/to/file` : find out which package "owns" the given file.
- `dpkg --info foo_version-revision_arch.deb` : show information stored in a package file.
- `dpkg-deb --contents X.deb` : list files inside a package file.

- `dpkg-deb --extract X.deb` : extract files from a package file (does not install).
- `apt-cache search keyword` : scan through the repository package lists while looking for the keyword and display package names which match.
- `apt-cache show packagename` : show information about package packagename (using the repository lists).

Please note that these commands and paths will be present on any real Debian system, but might be absent from special purpose target devices. An Internet Tablet for example might not contain all of these, and also some files have been removed to conserve storage space. The SDK environment contains them and you can even use `apt-get` to install new software from the `maemo.org` public repositories and get source code for most of the programs and libraries. This will require an connection to the Internet. You can sometimes even do an upgrade to a newer version of the SDK (you should check the SDK release notes and installation instructions before doing this).

## 8.5 Debian packages

Packages generally contain all the files necessary to implement a set of related commands and/or features.

There are two kinds of Debian packages:

- Binary packages (one **.deb** file): Contain executables, configuration files, man/info pages, copyright information and other documentation. This file is meant for end-user to install with `dpkg` on their system and contains compiled versions of software. Each deb-file is built for a specific architecture with specific compilation and build flags. This means that a .deb-file built for the i386-architecture will not run on ARM-architecture. The target architecture of the deb-file is given in the filename of the file (**sopwith\_1.7.1-1\_i386.deb** where the last part before .deb is the architecture). Some binary packages consist of files that are architecture independent (scripts or documentation) and their architecture is 'all' (**python-imaging+\_1.1.4-3.1\_all.deb** and **perl-doc\_5.8.4-8\_all.deb**)
- Source packages (at least one **.dsc** file and **.orig.tar.gz** file): These packages are used to distribute source code that can be used to create the binary packages. Most software is not Debian-specific, so you will normally find three files: The .dsc file that describes this source package, the source code of the original version (.tar.gz) and a compressed diff-file that contains the changes to be applied to the source code so that the software will adhere to the Debian policy. Sometimes these changes also include bug fixes. These changes are distributed as a .diff.gz-file (an unified difference file produced by the `diff`-tool).

Combined, these three files contain the source code, information and scripts necessary to build a binary package (or sometimes multiple binary packages from the same source package).

The package filename is structured as follows:



### **foo\_versionNumber-maemoRevision\_arch.deb**

Package filename components:

- **foo**: name of package
- **versionNumber**: upstream software version (i.e., original non-Debian version)
- **maemoRevision**: .diff.gz has changed, while upstream version not.
- **arch**: name of target architecture (for binary package files)

The **maemoRevision** field might change when the control file or installable configuration files change. It will also sometimes change when a quick bug fix is done but the upstream maintainer hasn't yet released a version that includes this bug fix.

Some examples from one release of the SDK follow:

- **libdb4.2\_4.2.52-18osso\_armel.deb**:
  - **libdb4.2**: Binary package name. Package versions with incompatible APIs will normally have their package name contain a number.
  - **4.2.52-18osso**: Upstream version is 4.2.52, with local modifications for maemo at version 18osso.
  - **armel**: Binary package for the armel architecture.
- **libdb4.2\_4.2.52-18osso\_i386.deb**: Binary package for the i386 architecture ("X86").
- **db4.2\_4.2.52-18osso.diff.gz**: Differences between upstream and maemo version of the package. One part of the source package.
- **db4.2\_4.2.52-18osso.dsc**: Debian Source Control file (defines a source package file). One part of the source package.
- **db4.2\_4.2.52.orig.tar.gz**: Source code tarball of the upstream version. One part of the source package.

One noteworthy thing above is that source package names don't always correspond to binary package names. This is especially true with large source package which will produce multiple binary packages, or will package documentation separately.

The binary packages that can be built from one source package are listed using the **Binary** field of the Debian source control file:

```
Format: 1.0
Source: db4.2
Version: 4.2.52-18osso
Binary: libdb4.2+, db4.2-doc, libdb4.2-dev, libdb4.2+-dev, libdb4.2
Maintainer: Debian Berkeley DB Maintainers
<pkg-db-devel@lists.aliases.debian.org>
Architecture: any
Standards-Version: 3.6.1
Build-Depends: procps [!hurd-i386]
Uploaders: Clint Adams <schizo@debian.org>,
Matthew Wilcox <willy@debian.org>,
Andreas Barth <aba@not.so.arh.org>
Files:
cbc77517c9278cdb47613ce8cb55779f 4073147 db4.2_4.2.52.orig.tar.gz
4926da646ea05246767da25aac139aef 80499 db4.2_4.2.52-18osso.diff.gz
```

If a package is significantly different from the upstream version, it won't always have a `.diff.gz` file.

## 8.6 Installation process

When a package is installed by `dpkg`, the installation goes through various stages in the order specified below (from the `dpkg` man-page):

- Extract control files of the new package.
- If this is a package upgrade, execute the `'prerm'`-script of old version.
- Execute `'preinst'`-script if one is provided in the package.
- Backup old files (if upgrading) and unpack new files into their locations.
- If upgrading, execute `'postrm'` of the old package.
- Configure the installed package (`--configure`).

## 8.7 Package relationships

Each package contains a `control`-file which contains all the dependency, conflict and feature information. These are collectively called package relationships.

Debian supports the following kinds of relationships:

- **Pre-depends:** Similar to `'Depends'` (below) but meant for enforcing ordering of installation. **DO NOT USE.**
- **Depends:** Package will not work without specified version (or newer) of another package. Note that `Depends` rules are used after the package has been unpacked just before it's about to be configured. In some cases this might leave the package installed but non-configured (rare).
- **Recommends:** Recommended package will be useful to most users of this package.
- **Suggests:** Suggested package might include additional functionality and be useful to some users.
- **Enhances:** Similar to suggests but works in reverse direction with respect package order.
- **Conflicts:** This package will not operate correctly if the conflicting package is already installed.
- **Replaces:** The referenced package will be replaced by this package (by overwriting of original package files). When used together with `Conflicts`, a package replacing a conflicting package will cause the conflicting package to be removed first.

- **Provides:** Used to note that some function is provided by this package. Not used in maemo.  
Syntax for the package specifications when declaring the relationships is as follows:
- Multiple package names are separated by a comma if the restriction must be against all of them (boolean AND).
- If one or more packages can satisfy the restriction, separate their names with the '|' (pipe) character (boolean OR).
- If version number of some package is significant, you can specify the version restriction by using «, <=, =, >= and » and a version number.

Some examples:

**Depends:** foo (=1.2.0)

Depends on package foo's exact version 1.2.0 (no other installed version will do)

**Conflicts:** foo

Conflicts with all versions of package foo

**Depends:** foo (>=1.2.3) | foobars, foozonkle

Package installation requires that package foo's version 1.2.3 or higher is installed, or package foobars is installed. Package foozonkle must be installed as well (irrespective of the previous restriction) but any version of it will do.

## 8.8 Package control file (aka Debian control file)

```
Package: packagename
Priority: optional (to aid intelligent tools wrt desktop installation)
Section: devel (which part of the ftp archives on Debian this package lives in)
Installed-Size: 45 (in KiB) to aid intelligent inst tools (filled in automatically)
Maintainer: First Lastname
Architecture: i386 (filled in automatically)
Version: 1.3-16 (-16 = Debian revision, filled in automatically)
Depends: libc6 (>= 2.1) (filled automatically when using shlibs:Depends-macro)
Description: The classic greeting, and a good example
Long description starts always with a space
Empty lines are not permitted (they terminate the long description)
So, there is a mechanism to allow this:
.
That was an "empty line"
```

An example Debian control file

For full syntax and canonical explanation, please see the Debian Policy Manual, Section 5.

**Depends** lists the packages which need to be installed for this package to install successfully (explained above).

Debian is split into three main sections: **main** (free software), **non-free** (not really free according to Debian policy), **contrib** (free software that depends on non-free).

However, in maemo, only one section has been defined so far: `user`. This has been further split into application categories which should be used when possible so that automatic localisation can be done in the Application manager.

The subsections as of this moment are:

- `user/accessories`: Accessories
- `user/communication`: Communication
- `user/games`: Games
- `user/multimedia`: Multimedia
- `user/office`: Office
- `user/other`: Other
- `user/programming`: Programming
- `user/support`: Support
- `user/themes`: Themes
- `user/tools`: Tools

If you cannot find a subsection that suits you, you can create a new one of your choosing, but it will not be automatically localised.

If you do not use the `user` section, the Application manager will not show your package as installable.

Each package is assigned a priority by the distribution maintainer (normally) to signify the relative importance with respect to proper functioning of the installed system. These control intelligent installation tools like `apt-get`. However, in maemo, only the priority `optional` should be used for your packages.

## 8.9 A maemo example control file

The Application manager supports special control directives as well as the normal Debian ones. This is how packages are provided with icons that the user will see on installation. These special directives start all with the prefix `XB-Maemo-`.

As example control file from the maemo.org HOWTOs looks like this:

```
Source: myapplication
Section: user/other
Priority: optional
Maintainer: Your Name <your.name@example.org>
Build-Depends: debhelper (>= 5)
Standards-Version: 3.7.2

Package: myapplication
Architecture: any
Depends: libhildon1 (>= 1.0.11)
Description: A simple test application
 A very simple application with a short description.
 Which spans multiple lines actually.
XB-Maemo-Icon-26:
iVBORw0KGgoAAAANSUUhEUgAAABoAAAAACAYAAACpSkzOAAAABmJLR0QA/wD/AP+g
vaeTAAACXBtWXMMAAAsTAAAEwEAMPwYAAAAB3RJTUUH1gURDQoYya0JlWAAAU9J
REFUSMftl1KA0EUhb/NZ1/ggnHQxsJUxt5CUucVJCCKdfgyKdIGG5/A0s5HEBtJ
EdDAQGBgmw0YJmMzgXXYZa5CtNkDW9zZw5z7c+ZCgwb/Ai3i9sV1/Bq8RIs4LRK1
gJDsKvJyNXmJMuYTsMoY1zpgzoaABdYArQNPZQ1kfyGU7SpqVwxzAMwABWhgpIwp
4vWBB+AUWAI3ypjnfEXtPU4bLKx9vErTeCeIRSYF+fTn1j5dp2myE9EiU+DSi3wX
ymeqRQAmZ3EcA5E/fg06BULT8zh0crwXoJdrXRa2Lgps2y2odAUCBUIXQdz78YyC
SldAp8b7+bXrIv91qjZBietqCc2DjbAt4b2WxJkyZLjVujlwp0U0cPxuLcATuC+4
dKxFlsDJarvdAGP/b6hFnDImYs+uG3hb02AB3Jbsur63tQM+fFx3bzZocEB8AdV2
gJBZgKTWAAAAAE1FTkSuQmCC
```

The icon is embedded as an MIME-encoded 26x26 pixel PNG file directly into the control directive.

For more information, please see [maemo.org](http://maemo.org) documentation.

## 8.10 Creating your package

In this section we'll cover step by step instructions on how to convert an autotoolised project into a Debianised project and how to generate the binary package based on the Debian control file.

Step by step instructions (with screen captures shortly):

- Clean up your source package (by running **antigen.sh** or similar tool). The important bit here is to get rid of the **--prefix** information after previous testing. Building software packages is done without the **--prefix** option to configure.
- Bootstrap your source by running **autogen.sh**. Fill in the missing files and be especially careful about the **COPYING** file since that holds the software license that will be included in the package. It is GPL by default. For the example code, a special License is used (which is used for all example code of this material) instead of GPL.
- Run the configure script once without any parameters in order to get the make targets.
- Verify software package coherency by making **distcheck** target. Do not continue past this point if **distcheck** fails for some reason.
- Make a **dist** package of your source.
- Create a separate directory under which you'll do your packaging (**packaging** is not a bad name for it) and copy the **dist** package there.
- Change into the directory and extract your distribution package there. This should result in the proper **package-version** subdirectory if your

autotoolisation went correctly. It is important that the path is of the correct format, otherwise the packaging process will not go smoothly!

- Change your directory into the freshly uncompressed source directory(**package-version**).
- Bootstrap the necessary Debian control files:
  - `DEBFULLNAME="Your Fullname" dh_make -e your@email.com -f ../path-to-dist.tar.gz`  
Here `../path-to-dist.tar.gz` refers to the `dist` target built source tarball. Making a copy of it under **packaging** is not a bad idea. `dh_make` will ask you two questions interactively:
    - \* Answer `s` to the first (single binary package for us).
    - \* Press `ENTER` to approve the configuration (if it's not correct, press `Ctrl+c`).
- List contents of `./debian`. It should contain a lot of files.
- Replace the `debian/copyright` file with the contents of your real license file (take a look at the original version first though).
- Remove the ones that you don't need (you need these: **changelog**, **control**, **copyright**, **compat** and **rules**).
- Edit the control file:
  - Check your `Maintainer` name and email (if they're not correct). If you have to change them, you'll also have to update the **changelog**-file since it was generated automatically as well with the same information.
  - Change the `Section` so that the main section will be `user` and the subsection any of the ones listed previously. The example below will use `user/other`.
  - Change `Build-Depends` to contain all the necessary development packages that need to be installed in order to build your package. In our case: **debhelper (>= 5)**, **libgtk2.0-dev**, **libhildon1-dev**, **libhildonfm2-dev**, **libosso-gnomevfs2-dev**, **libgconf2-dev**, **libosso-dev**. To get such a list, start with the library names that you need for `pkg-config`, then find out which Debian packages the libraries come in(`dpkg -l`). You can also try to find the packages owning the `pkg-config` configuration files (all under `/usr/lib/pkgconfig/`) with `dpkg -S /usr/lib/pkgconfig/hildon-1.pc` for example.
  - Modify the `Depends` to read **\$shlibs:Depends** (so that dependencies will be filled by `dpkg-buildpackage` automatically). The **\$misc:Depends** is not normally necessary unless dealing with complex packages. Please see the manual page for `debhelper` if you think you need it.
  - Leave the `Architecture` as is for now (it will be filled by `dpkg-buildpackage` automatically).

- Consider whether you want to add an icon to your package. If you do, please check the relevant documentation on [maemo.org](http://maemo.org) on how to generate the icon MIME encoded field.
- Change back to the source directory just above **debian**.
- `dpkg-buildpackage -rfakeroot .`
- The binary package is left at the directory above the current one.  
*dpkg-buildpackage* will also automatically create the source package files for your application (in the same directory as the binary package).

Let's see what will happen when we follow these instructions. We have already tested the autotoolised package before and verified that it works with `make distcheck`. We start by from a clean setup, by doing the `dist` target we'll get the source tarball that we'll need to start with the package building process.

```
[sbox-DIABLO_X86: ~/autotoolized-hildon-helloworld] > ls -la
total 80
drwxr-xr-x  2 user user 4096 Nov 18 20:37 .
drwxr-xr-x 10 user root 4096 Nov 18 20:36 ..
-rw-r--r--  1 user user  124 Nov 18 20:36 AUTHORS
-rw-r--r--  1 user user 1174 Nov 18 20:36 COPYING
-rw-r--r--  1 user user  159 Nov 18 20:36 ChangeLog
-rw-r--r--  1 user user 1174 Nov 18 20:36 License
-rw-r--r--  1 user user  847 Nov 18 20:36 Makefile.am
-rw-r--r--  1 user user   85 Nov 18 20:36 NEWS
-rw-r--r--  1 user user   73 Nov 18 20:36 README
-rwxr-xr-x  1 user user  706 Nov 18 20:36 antigen.sh
-rwxr-xr-x  1 user user 1024 Nov 18 20:36 autogen.sh
-rw-rw-r--  1 user user 1542 Nov 18 20:36 configure.ac
-rw-rw-r--  1 user user 23910 Nov 18 20:36 hhwX.c
-rw-r--r--  1 user user  162 Nov 18 20:36 hhwX.desktop.in
-rw-r--r--  1 user user   59 Nov 18 20:36 org.maemo.hhwX.service.in
[sbox-DIABLO_X86: ~/autotoolized-hildon-helloworld] > ./autogen.sh
.. output cut ..
Makefile.am: installing './depcomp'
Ready to go (run configure)
[sbox-DIABLO_X86: ~/autotoolized-hildon-helloworld] > ./configure
checking for a BSD-compatible install... /scratchbox/tools/bin/install -c
checking whether build environment is sane... yes
.. output cut ..
checking pkg-config is at least version 0.9.0... yes
checking for HHW... yes
configure: creating ./config.status
config.status: creating Makefile
config.status: executing depfiles commands
configure: creating ./config.status
config.status: creating Makefile
config.status: creating hhwX.desktop
config.status: creating org.maemo.hhwX.service
config.status: executing depfiles commands
[sbox-DIABLO_X86: ~/autotoolized-hildon-helloworld] > make dist
{ test ! -d hhwX-0.1 || .. output cut..
[sbox-DIABLO_X86: ~/autotoolized-hildon-helloworld] > ls -l *.tar.gz
-rw-rw-r--  1 user user 80229 Nov 18 20:51 hhwX-0.1.tar.gz
```

Preparing the source code for Debian packaging

You might notice that the **antigen.sh** step was omitted above. Since the source code was in a pristine condition, running **antigen.sh** was unnecessary (as running the `make distcheck` -step).

Next, let's create a directory one level up where we'll extract the package and then prepare the extracted source for Debianisation:

```
[sbox-DIABLO_X86: ~/autotoolized-hildon-helloworld] > mkdir ../packaging
[sbox-DIABLO_X86: ~/autotoolized-hildon-helloworld] > cp hhwx-0.1.tar.gz ../packaging
[sbox-DIABLO_X86: ~/autotoolized-hildon-helloworld] > cd ../packaging
[sbox-DIABLO_X86: ~/packaging] > tar xzvf hhwx-0.1.tar.gz
hhwx-0.1/
hhwx-0.1/COPYING
hhwx-0.1/configure.ac
hhwx-0.1/NEWS
hhwx-0.1/INSTALL
hhwx-0.1/Makefile.in
hhwx-0.1/org.maemo.hhwX.service.in
hhwx-0.1/hhwX.desktop.in
hhwx-0.1/hhwX.c
hhwx-0.1/aclocal.m4
hhwx-0.1/Makefile.am
hhwx-0.1/AUTHORS
hhwx-0.1/README
hhwx-0.1/configure
hhwx-0.1/depcomp
hhwx-0.1/missing
hhwx-0.1/install-sh
hhwx-0.1/ChangeLog
[sbox-DIABLO_X86: ~/packaging] > cd hhwx-0.1
```

Creating the packaging working space and extracting the source

Next, we use the `dh_make` program to create the necessary Debian files so that we don't have to write everything ourselves:



```
[sbox-DIABLO_X86: ~/packaging/hhwx-0.1] > DEBFULLNAME="User Universal" \
dh_make -e user@maemo.org -f ../hhwx-0.1.tar.gz

Type of package: single binary, multiple binary, library, kernel module or cdb?
[s/m/l/k/b] s

Maintainer name : User Universal
Email-Address   : user@maemo.org
Date            : Sun, 18 Nov 2007 21:06:26 +0200
Package Name    : hhwx
Version         : 0.1
License         : blank
Type of Package : Single
Hit <enter> to confirm: [ENTER]
Done. Please edit the files in the debian/ subdirectory now. hhwx
uses a configure script, so you probably don't have to edit the Makefiles.
[sbox-DIABLO_X86: ~/packaging/hhwx-0.1] > cd debian
[sbox-DIABLO_X86: ~/packaging/hhwx-0.1/debian] > ls -l
total 104
-rw-rw-r-- 1 user user 171 Nov 18 21:06 README.Debian
-rw-rw-r-- 1 user user 181 Nov 18 21:06 changelog
-rw-rw-r-- 1 user user  2 Nov 18 21:06 compat
-rw-rw-r-- 1 user user 337 Nov 18 21:06 control
-rw-rw-r-- 1 user user 621 Nov 18 21:06 copyright
-rw-rw-r-- 1 user user  77 Nov 18 21:06 cron.d.ex
-rw-rw-r-- 1 user user  17 Nov 18 21:06 dirs
-rw-rw-r-- 1 user user  12 Nov 18 21:06 docs
-rw-rw-r-- 1 user user 1224 Nov 18 21:06 emacs-en-install.ex
-rw-rw-r-- 1 user user  456 Nov 18 21:06 emacs-en-remove.ex
-rw-rw-r-- 1 user user 1111 Nov 18 21:06 emacs-en-startup.ex
-rw-rw-r-- 1 user user  226 Nov 18 21:06 hhwx-default.ex
-rw-rw-r-- 1 user user  486 Nov 18 21:06 hhwx.doc-base.EX
-rw-rw-r-- 1 user user 2106 Nov 18 21:06 init.d.ex
-rw-rw-r-- 1 user user 1731 Nov 18 21:06 manpage.1.ex
-rw-rw-r-- 1 user user 4640 Nov 18 21:06 manpage.sgml.ex
-rw-rw-r-- 1 user user 4597 Nov 18 21:06 manpage.xml.ex
-rw-rw-r-- 1 user user  109 Nov 18 21:06 menu.ex
-rw-rw-r-- 1 user user  956 Nov 18 21:06 postinst.ex
-rw-rw-r-- 1 user user  929 Nov 18 21:06 postrm.ex
-rw-rw-r-- 1 user user  689 Nov 18 21:06 preinst.ex
-rw-rw-r-- 1 user user  876 Nov 18 21:06 prerm.ex
-rwxr-xr-x 1 user user 2515 Nov 18 21:06 rules
-rw-rw-r-- 1 user user  659 Nov 18 21:06 watch.ex
[sbox-DIABLO_X86: ~/packaging/hhwx-0.1/debian] > cat ../COPYING > copyright
[sbox-DIABLO_X86: ~/packaging/hhwx-0.1/debian] > ls -l ../COPYING copyright
-rw-r--r-- 1 user user 1174 Nov 18 20:36 ../COPYING
-rw-rw-r-- 1 user user 1174 Nov 18 21:07 copyright
```

Creating template files for packaging and overwriting the default copyright file.

The **copyright** file was overwritten because the default one (a GPL template) didn't suite the project. You should check the template whether it is suitable for use(**dh\_make** also supports couple of other open source copyright/license templates).

Now we need to remove the files that we don't need:

```
[sbox-DIABLO_X86: ~/packaging/hhwx-0.1/debian] > rm *.ex *.EX dirs docs README.Debian
[sbox-DIABLO_X86: ~/packaging/hhwx-0.1/debian] > ls -l
total 20
-rw-rw-r-- 1 user user 181 Nov 18 21:06 changelog
-rw-rw-r-- 1 user user  2 Nov 18 21:06 compat
-rw-rw-r-- 1 user user 337 Nov 18 21:06 control
-rw-rw-r-- 1 user user 1174 Nov 18 21:07 copyright
-rwxr-xr-x 1 user user 2515 Nov 18 21:06 rules
```

Removing unnecessary example files

Then edit the **control** file to contain the necessary dependencies, the necessary Section and other settings. We're not adding the icon to our package at this time.

```
Source: hhwx
Section: user/other
Priority: extra
Maintainer: User Universal <user@maemo.org>
Build-Depends: debhelper (>= 5), libgtk2.0-dev, libhildon1-dev,
  libhildonfm2-dev, libosso-gnomevfs2-dev, libgconf2-dev, libosso-dev
Standards-Version: 3.7.2

Package: hhwx
Architecture: any
Depends: ${shlibs:Depends}
Description: The ultimate Hello World
  This is the tenth version of Hello World.
  .
  Demonstrates simple GUI things with Hildon and support
  libraries.
```

The modified version of control file

And now everything should be ready for us to build the package:

```

[sbox-DIABLO_X86: ~/packaging/hhwx-0.1] > dpkg-buildpackage -rfakeroot
dpkg-buildpackage: source package is hhwx
dpkg-buildpackage: source version is 0.1-1
dpkg-buildpackage: source changed by User Universal <user@maemo.org>
dpkg-buildpackage: host architecture i386
dpkg-buildpackage: source version without epoch 0.1-1
: Using Scratchbox tools to satisfy builddeps
fakeroot debian/rules clean
dh_testdir
dh_testroot
.. output cut ..
dh_strip
dh_compress
dh_fixperms
dh_installdeb
dh_shlibdeps
dh_gencontrol
dh_md5sums
dh_builddeb
dpkg-deb: building package 'hhwx' in './hhwx_0.1-1_i386.deb'.
dpkg-genchanges
dpkg-genchanges: including full source code in upload
dpkg-buildpackage: full upload (original source is included)
[sbox-DIABLO_X86: ~/packaging/hhwx-0.1] > cd ..
[sbox-DIABLO_X86: ~/packaging] > ls -l
total 216
drwxrwxr-x 4 user user 4096 Nov 18 21:44 hhwx-0.1
-rw-rw-r-- 1 user user 80229 Nov 18 21:00 hhwx-0.1.tar.gz
-rw-rw-r-- 1 user user 24248 Nov 18 21:44 hhwx_0.1-1.diff.gz
-rw-rw-r-- 1 user user 400 Nov 18 21:44 hhwx_0.1-1.dsc
-rw-rw-r-- 1 user user 727 Nov 18 21:44 hhwx_0.1-1_i386.changes
-rw-rw-r-- 1 user user 11750 Nov 18 21:44 hhwx_0.1-1_i386.deb
-rw-rw-r-- 1 user user 80229 Nov 18 21:00 hhwx_0.1.orig.tar.gz
[sbox-DIABLO_X86: ~/packaging] > dpkg --info hhwx_0.1-1_i386.deb
new debian package, version 2.0.
size 11750 bytes: control archive= 773 bytes.
    641 bytes,   13 lines   control
    409 bytes,    6 lines   md5sums
Package: hhwx
Version: 0.1-1
Section: user/other
Priority: extra
Architecture: i386
Depends: libatk1.0-0 (>= 1.12.2), libc6 (>= 2.5.0-1), libcairo2 (>= 1.4.10),
        libdbus-1-3 (>= 0.94), libdbus-glib-1-2 (>= 0.74), libgconf2-6 (>= 2.13.5),
        libglib2.0-0 (>= 2.12.12-10sso3), libgtk2.0-0 (>= 2:2.10.12-0osso15),
        libhildon1 (>= 1.0.11), libhildonfm2 (>= 1:1.9.46), libosso-gnomevfs2-0,
        libosso1 (>= 2.13), libpango1.0-0 (>= 1.16.4)
Installed-Size: 84
Maintainer: User Universal <user@maemo.org>
Description: The ultimate Hello World
 This is the tenth version of Hello World.

.
Demonstrates simple GUI things with Hildon and support
libraries.
[sbox-DIABLO_X86: ~/packaging] > dpkg --contents hhwx_0.1-1_i386.deb
drwxr-xr-x root/root      0 2007-11-18 21:44:44 ./
drwxr-xr-x root/root      0 2007-11-18 21:44:43 ./usr/
drwxr-xr-x root/root      0 2007-11-18 21:44:44 ./usr/bin/
-rwxr-xr-x root/root 20384 2007-11-18 21:44:44 ./usr/bin/hhwx
drwxr-xr-x root/root      0 2007-11-18 21:44:43 ./usr/share/
drwxr-xr-x root/root      0 2007-11-18 21:44:43 ./usr/share/applications/
drwxr-xr-x root/root      0 2007-11-18 21:44:43 ./usr/share/applications/hildon/
-rw-r--r-- root/root    158 2007-11-18 21:44:43 ./usr/share/applications/hildon/hhwx.desktop
drwxr-xr-x root/root      0 2007-11-18 21:44:43 ./usr/share/dbus-1/
drwxr-xr-x root/root      0 2007-11-18 21:44:43 ./usr/share/dbus-1/services/
-rw-r--r-- root/root     55 2007-11-18 21:44:43 ./usr/share/dbus-1/services/org.maemo.hhwx.service
drwxr-xr-x root/root      0 2007-11-18 21:44:43 ./usr/share/doc/
drwxr-xr-x root/root      0 2007-11-18 21:44:44 ./usr/share/doc/hhwx/
-rw-r--r-- root/root    1174 2007-11-18 21:07:54 ./usr/share/doc/hhwx/copyright
-rw-r--r-- root/root    158 2007-11-18 20:36:56 ./usr/share/doc/hhwx/changelog.gz
-rw-r--r-- root/root    189 2007-11-18 21:06:40 ./usr/share/doc/hhwx/changelog.Debian.gz

```

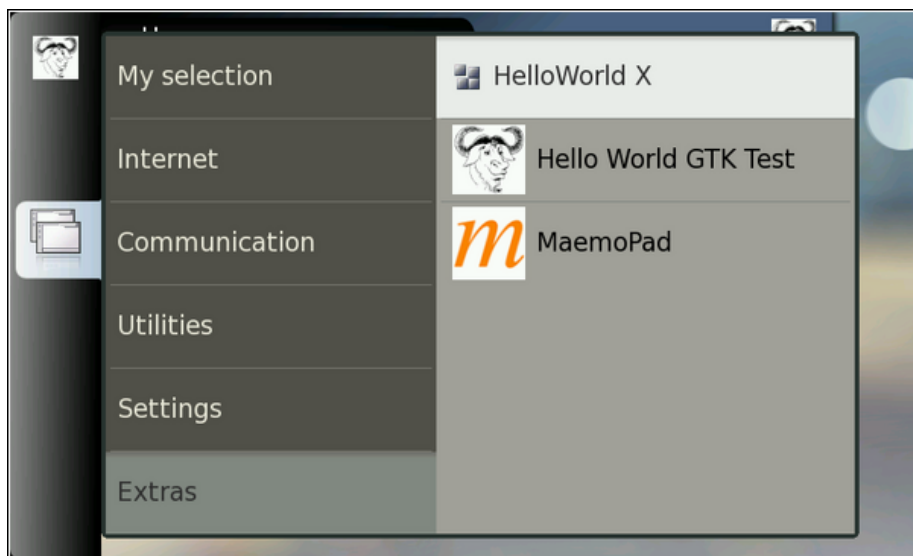
Building the package and listing its information and contents

Now we're ready to install the package using `dpkg`.

```
[sbox-DIABLO_X86: ~/packaging] > fakeroot dpkg -i hhwx_0.1-1_i386.deb
Selecting previously deselected package hhwx.
(Reading database ... 19483 files and directories currently installed.)
Unpacking hhwx (from hhwx_0.1-1_i386.deb) ...
Setting up hhwx (0.1-1) ...
```

Installing the package manually

After the installation, the menu should now be updated and you should be able to start the application.



Everything seems to be in order

We then also remove the package manually.

```
[sbox-DIABLO_X86: ~/packaging] > fakeroot dpkg --purge hhwx
(Reading database ... 19490 files and directories currently installed.)
Removing hhwx ...
```

Removing the package manually.

You can verify that the application has been removed from the menus now.

End users normally will not use `dpkg` manually, but instead they will use the Application manager program in the target device. In order for the Application manager to find the file to install, it will need to be located under `/MyDocs/`. You then need to use the "install from a file..." function from the menu.

## 8.11 Adding debugging support for a package

So that your application can be debugged by others, you should consider adding a separate package for your application which will contain the debugging information. Normally debugging symbols are stripped as part of

the package building process above, so the resulting package ends up being without debugging support. This is an issue on the ARMEL target.

In order to create a debugging version of your package, please see the "Making a Debian Debug Package" section of Maemo Diablo Reference Manual.

## 8.12 Building the package for a device

At this point you should make sure that you have the ARMEL target in your Scratchbox as well as the X86 one which you've (hopefully) been using so far:

```
[sbox-DIABLO_X86: ~] > sb-conf ls -T
DIABLO_ARMEL
DIABLO_X86
```

Listing set up targets in Scratchbox.

If you only have the X86 target setup, you will need to setup the ARMEL target now, before proceeding. Please see the last chapter of "maemo Getting Started" material for instructions. If you used the automatic installation script, you should automatically have both targets.

If you have your software, or the AF running, you'll need to stop both now(`af-sb-init.sh stop` to stop the AF). It is important to do this before switching targets, although sbx normally won't allow you to switch anyway if you have any processes running in the current target.

When you're done with shutting down the current target, switch your target to ARMEL:

```
[sbox-DIABLO_X86: ~] > sb-conf select DIABLO_ARMEL
.. screen clears ..
[sbox-DIABLO_ARMEL: ~] > arch
arm
```

Switching the Scratchbox target.

To build the Debian package in the ARMEL target:

- Rebuild your Debian package(`dpkg-buildpackage -rfakeroot`), which should result in an armel-version deb-file.
- Install the armel-version of Debian package (it's a separate target after all). You'll need to use `dpkg -i` for this. Also check that your package is removable(`apt-get remove`).

One noteworthy thing about the ARMEL target is that it's not really meant to act as a testing environment. How much of your software will work, will depend on the version of Qemu that will be used to emulate the ARM instructions. In short, it's best to test the ARMEL versions of your packages on a real device.

## 8.13 Installing packages into an Internet Tablet

Once you've built the ARM-version of your package, you'll next need to transfer it to the device. Since the device appears as an USB mass storage device to your

Linux, it is normally enough to attach the USB cable and then mount the device into some suitable directory under Linux. Also most modern graphical environments will do the mounting step for you, so don't be surprised if you don't need to do the mounting manually.

The storage that you can access this way is restricted to the memory card inside the device. It is not possible to access the internal flash memory of the device in this way. The filesystem on the memory card will be VFAT, so it will contain all the normal problems that are related to Windows file-names. Copy the package to the mounted directory and then un-mount the USB storage(`pumount` is normally enough, or you can use the graphical interface). After detaching the cable from the device the device will mount the card internally and it can access the contents on the card.

Because the storage space is not available to both the device and your Linux desktop at the same time, some people opt to use an SSH server and use `scp` to copy files directly. Setting up an SSH server on the device is not covered here, but is pretty simple. Just remember to set your device into "R&D"-mode.

Once the package file is on the device (under `/home/user/MyDocs`), use the Application manager and select the package file to install. If something will go wrong, you'll probably need a command line access to the device. Use the supplied X-Term emulator, and `sudo gainroot` to get root privileges (device needs to be in "R&D"-mode for `gainroot` to work).

# Appendix A

## The Final Program

### A.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.

### A.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 A.1: Contents of autotoolized-hildon-helloworld/configure.ac

## A.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 A.2: Contents of autotoolized-hildon-helloworld/Makefile.am

## A.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 A.3: Contents of autotoolized-hildon-helloworld/hhwX.desktop.in

## A.5 Service file template for AF

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



## A.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 A.5: Contents of autotoolized-hildon-helloworld/autogen.sh

## A.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 A.6: Contents of autotoolized-hildon-helloworld/antigen.sh

## A.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 GtkWidget, 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 successfully, 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 the 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 A.7: Contents of full.c (hhwX.c with most comments left in)