Maemo Diablo More Widgets
Training Material

February 9, 2009

Contents

Chapter 1

More Widgets

1.1 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
```

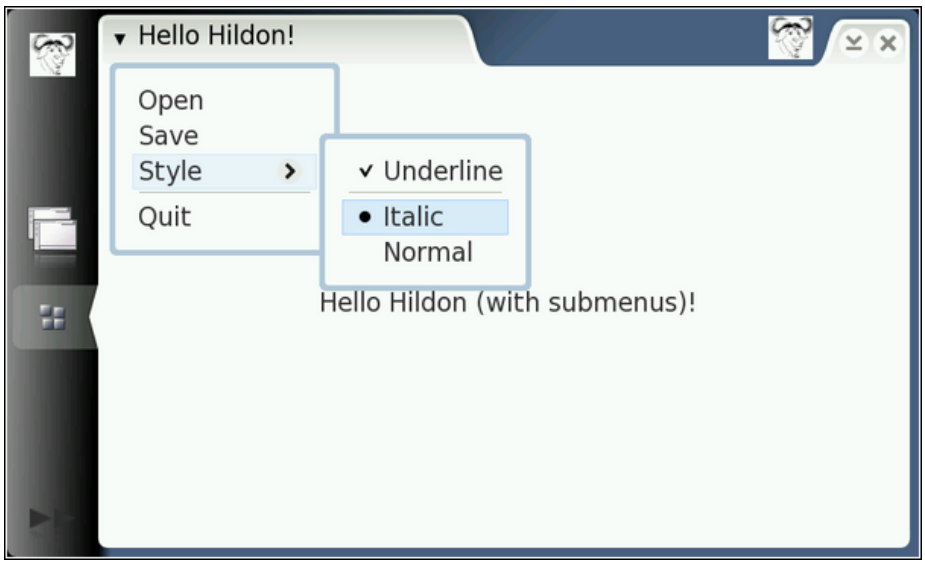
```
         We handle toggles for both radio items in the same code.
         This code will be called twice on each toggle since one of
         the radio items will be toggled off, the other will be
         toggled on.

         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) {

  GtkMenu* 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 1.1: 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.

1.2 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:

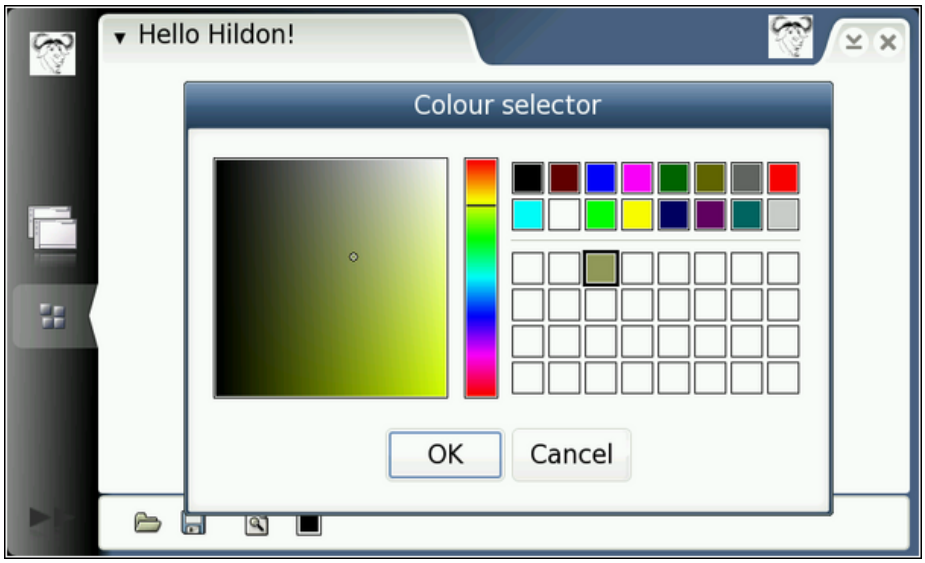
```
/**
 * hildon_helloworld-4.c
 *
```

```
 * 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 GtkWidgets 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 a 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 1.2: Code that adds a toolbar and a color selection button (hildon_helloworld-4.c)



Running the application

1.3 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 `ApplicationState`. 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 1.3: 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.

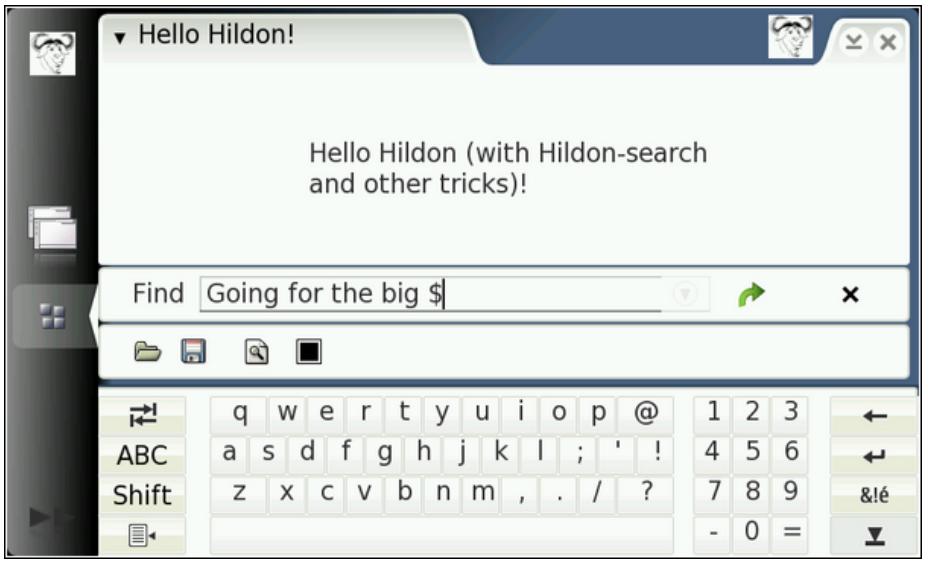


Figure 1.1: Our program with both toolbars and the VKB.

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.

1.4 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 1.4: 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).

1.5 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) {
```

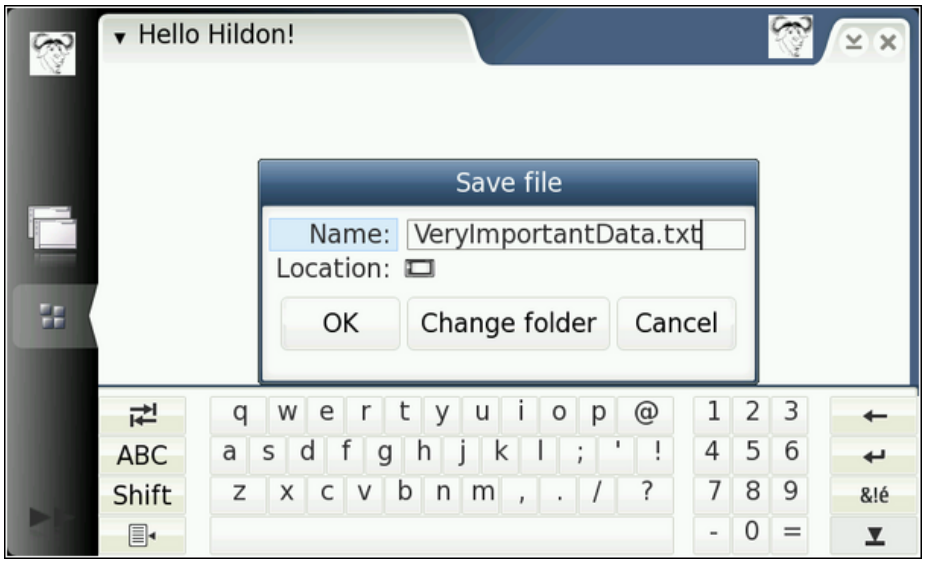
```
  GtkWidget* dialog = NULL;
  gchar* filename = NULL;

  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 1.5: 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.

1.6 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. 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) 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.
- There is no current GDK tutorial (unfortunate).
- There is however the GDK API reference at 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.

1.7 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!