Menus and Toolbars

1 Introduction

Most GUI applications have menus and toolbars. They are an important part of how the user interacts with the application, sometimes the only ways by which the user interacts with it. Although menus and toolbars look like different things, and in some aspects are different things, what they have in common is that they are both containers for widgets that, when clicked, result in the performance of actions. Menus contain menu items, and toolbars usually contain buttons. Although toolbars are actually more general than this in that they can contain arbitrary widgets, they are usually used to provide quick access to frequently used menu items.

We will examine menus first, and then toolbars. After covering the basics of each of these, we will look at a more unified way of creating both, using the `GtkUIManager`.

2 Menus

2.1 Principles

There are three widgets involved in menu creation:

menu items	These are the widgets that the user clicks.
menus	These are containers that contain menu items.
menubars	These are containers that appear to contain menus.

You will find menu creation and menu handling confusing unless you remember the following principles:

- Menus (class `GtkMenu`) and menubars (class `GtkMenuBar`) are containers. They are derived from the same abstract base class, `GtkMenuShell`.
- The only thing you can put into a menu or a menubar is a menu item.
- Menus can be "attached" to menu items so that when the item is activated, the menu *drops down* or *pops up*.
  - If a menu item is a child of a menubar, then the menu attached to it drops down.
  - If it is a child of a menu, then the menu attached to it pops up.
- Menu items are the only things that you can activate, and these emit an activate signal, which must be connected to a callback to handle the user's clicks on that item. Although they emit other signals, this is the one you will normally use.

If you remember these principles going forward, things will be easy to understand. In essence, menus form a *recursively defined hierarchy*. The root of this hierarchy is always a menubar. We usually think of menubars as horizontal, rectangular regions at the top of a window, but they may be vertical as well, and can be placed anywhere you choose. Those labels that you see in the menubar, such as "`File`", "`Edit`" or "`Help`", are menu items[1]. Menu items can have menus attached to them, so that when they get clicked, the menu appears. Each of the menus attached to a menu item may have menu items that have menus attached to them, and these may have items that have menus attached to them, and so on.

We use the term *submenu* to refer to a menu that is attached to a menu item within another menu, but there is no special class of submenus; a submenu is just a menu. Because a menu item always exists as a child of either a menu or a menubar, the menu that is attached to a menu item is always a submenu of something else. This should make it easy to remember the fact that there is but a single way to attach a menu to a menu item:

```
void            gtk_menu_item_set_submenu         ( GtkMenuItem *menu_item,
                                                    GtkWidget *submenu);
```

which we will discuss below. The point is that the attached menu is of necessity a submenu of something else.

2.2 Creating Menus "By Hand"

I call this method "by hand" because the menu is constructed in the same way that a typical house is constructed, by assembling the pieces and attaching them to each other, one by one. The alternative is more like the way an integrated circuit is made, by creating a specification of how everything fits together and using that specification to etch the circuit onto a wafer.

The outline of the steps that must be taken is:

1. Create the menubar.
2. Create the menu items that will be packed into the menubar.
3. Pack the menu items into the menubar.
4. Create the menus that the menu items will activate.
5. Attach these "submenus" to the menu items.
6. For each submenu,
   (a) create the menu items that it will contain, and
   (b) pack these menu items into the submenu.

I have listed these steps in a top-down sequence, but it is conceivable to carry them out in many different permutations. The above sequence has a natural logic to it and that is the order in which I describe them in detail below.

You create the menubar with the function

```
GtkWidget *     gtk_menu_bar_new                  ( void);
```

This creates an empty menubar. Menubars themselves have almost no methods. When menus and menubars were unified by creating a menushell class and deriving them from this parent class, their specific methods were deprecated. The menubar itself should be added to its parent container with an appropriate packing function. Typically you will put it at the top of the content area just below the top-level window's title bar, and there will be other "stuff" underneath it, so the usual sequence is

[1]Technically they are the labels of the items, not the items themselves.

```
vbox = gtk_vbox_new (FALSE, 0);
gtk_container_add (GTK_CONTAINER (window), vbox);
menu_bar = gtk_menu_bar_new ();
gtk_box_pack_start (GTK_BOX (vbox), menu_bar, FALSE, FALSE, 0);
```

For each menu that you want to put into the menubar, you will need a separate menu item. Regular menu items can be created in three different ways:

```
GtkWidget *    gtk_menu_item_new                ( void);
GtkWidget *    gtk_menu_item_new_with_label     ( const gchar *label);
GtkWidget *    gtk_menu_item_new_with_mnemonic  ( const gchar *label);
```

The first of these creates a menu item with no label; you would later use

```
void           gtk_menu_item_set_label          ( GtkMenuItem *menu_item,
                                                  const gchar *label);
```

to create a label for it. The second and third functions create menu items with either a plain label or with a label and a mnemonic, just like is done with buttons. Later you will see that there are four subclasses of menu items, among which are image menu items, which can contain an image instead of or in addition to a label.

There are three different ways to pack menu items into menubars and menus; they are all methods of the `GtkMenuShell` base class.

```
void           gtk_menu_shell_append            ( GtkMenuShell *menu_shell,
                                                  GtkWidget *child);
void           gtk_menu_shell_prepend           ( GtkMenuShell *menu_shell,
                                                  GtkWidget *child);
void           gtk_menu_shell_insert            ( GtkMenuShell *menu_shell,
                                                  GtkWidget *child,
                                                  gint position);
```

All three require the menu or menubar to be cast to the parent class. The second argument in all three is the menu item to be put into the container. The differences are probably obvious. The append method adds the menu item to the end of the list of those already in the menu shell, whereas the prepend method inserts it before all of the items already in it. The insert method takes an integer position as the third argument, which is the position in the item list where child is added. Positions are numbered from 0 to n-1. If an item is put into position k, then all items currently in the list at positions k through n-1 are shifted downward in the list to make room for the new item.

The following code fragment creates a few labeled menu items, and packs them into the menubar in left-to-right order.

```
file_item = gtk_menu_item_new_with_label( "File" );
view_item = gtk_menu_item_new_with_label( "View" );
tools_item = gtk_menu_item_new_with_label( "Tools" );
help_item = gtk_menu_item_new_with_label( "Help" );

gtk_menu_shell_append( GTK_MENU_SHELL( menu_bar ), file_item );
gtk_menu_shell_append( GTK_MENU_SHELL( menu_bar ), view_item );
gtk_menu_shell_append( GTK_MENU_SHELL( menu_bar ), tools_item );
gtk_menu_shell_append( GTK_MENU_SHELL( menu_bar ), help_item );
```

The next step is to create the menus that will be dropped down when these menu items are activated. Menus are created with the function

```
GtkWidget *    gtk_menu_new                        ( void);
```

For the above menu items, we would create four menus:

```
file_menu = gtk_menu_new();
view_menu = gtk_menu_new();
tools_menu = gtk_menu_new();
help_menu = gtk_menu_new();
```

Having created the menus, we have to attach them to the menu items using

```
void           gtk_menu_item_set_submenu          ( GtkMenuItem *menu_item,
                                                    GtkWidget *submenu);
```

This takes the menu item (cast to a menu item, since it is declared as a widget), and the menu to be attached. We attach the four menus as follows:

```
gtk_menu_item_set_submenu( GTK_MENU_ITEM( file_item ), file_menu );
gtk_menu_item_set_submenu( GTK_MENU_ITEM( view_item ), view_menu );
gtk_menu_item_set_submenu( GTK_MENU_ITEM( tools_item ), tools_menu );
gtk_menu_item_set_submenu( GTK_MENU_ITEM( help_item ), help_menu );
```

The next step is to create the menu items to populate each of the menus, and add them to these menus. There are no new functions to learn for these steps. It is simply a matter of creating menu items and packing them. In the example we are using, our `File` menu will have an `Open` item, a `Close` item, and an `Exit` item. We will also use a separator between the `Close` item and the `Exit` item. Separators are members of the class `GtkSeparatorMenuItem` and are created with

```
GtkWidget *    gtk_separator_menu_item_new         ( void);
```

The `File` menu's items will be simple labeled items. The code to create them and pack them is:

```
open_item  = gtk_menu_item_new_with_label( "Open");
close_item = gtk_menu_item_new_with_label( "Close");
separator1 = gtk_separator_menu_item_new();
exit_item  = gtk_menu_item_new_with_label( "Exit");

gtk_menu_shell_append( GTK_MENU_SHELL(file_menu), open_item );
gtk_menu_shell_append( GTK_MENU_SHELL(file_menu), close_item );
gtk_menu_shell_append( GTK_MENU_SHELL(file_menu), separator1 );
gtk_menu_shell_append( GTK_MENU_SHELL(file_menu), exit_item );
```

To create a menu that contains submenus does not involve anything other than descending a level in the menu hierarchy and repeating these steps. To illustrate, we will design the `Help` menu so that it has two items, one of which is a menu item that, when activated, pops up a submenu. The first two steps are to create the two menu items and pack them into the `Help` menu:

```
query_item = gtk_menu_item_new_with_label( "What's This?");
separator2 = gtk_separator_menu_item_new();
about_help_item = gtk_menu_item_new_with_label( "About this program" );

gtk_menu_shell_append( GTK_MENU_SHELL(help_menu), query_item );
gtk_menu_shell_append( GTK_MENU_SHELL(help_menu), separator2 );
gtk_menu_shell_append( GTK_MENU_SHELL(help_menu), about_help_item );
```

The next step is to create a submenu and attach it to the about_help_item:

```
about_help_menu = gtk_menu_new();
gtk_menu_item_set_submenu ( GTK_MENU_ITEM( about_help_item ), about_help_menu );
```

The last step is to create menu items for the about_help_menu and pack them into this menu:

```
about_tool_item = gtk_menu_item_new_with_label( "About Tools" );
about_stuff_item = gtk_menu_item_new_with_label( "About Other Stuff" );

gtk_menu_shell_append( GTK_MENU_SHELL(about_help_menu), about_tool_item );
gtk_menu_shell_append( GTK_MENU_SHELL(about_help_menu), about_stuff_item );
```

The preceding steps create the menu items, but they are not yet connected to the "activate" signal. Menu items that have submenus do not need to be connected to the "activate" signal; GTK+ arranges for that signal to open the submenu. But the others need to be connected. For example, we would connect the Exit menu item to a callback to quit the application with

```
g_signal_connect (G_OBJECT (exit_item), "activate",
              G_CALLBACK (gtk_main_quit),
                  (gpointer) NULL);
```

2.3 Pop-Up Menus for Widgets

The same techniques for creating menus rooted in a menubar applies to the creation of pop-up menus for other widgets. For example, if you wanted to create a button, which, when the mouse button is pressed on it, would pop up a menu instead of taking some other action, you would first create the menu using the instructions above. Then you would connect a mouse button press event signal to a callback that popped up the menu, using the g_signal_connect_swapped() function. To illustrate, we begin by creating a small pop-up menu and packing two menu items into it.

```
popupmenu      = gtk_menu_new();
makebig_item   = gtk_menu_item_new_with_label( "Larger");
makesmall_item = gtk_menu_item_new_with_label( "Smaller");
gtk_menu_shell_append (GTK_MENU_SHELL (popupmenu), makebig_item);
gtk_menu_shell_append( GTK_MENU_SHELL (popupmenu), makesmall_item );
```

We can connect the activate signal to these items as we choose. That part is not important now. Next we create a button and connect the generic "event" to a callback, swapping the button and the popup menu:

```
button = gtk_button_new_with_label ( "Push me");
g_signal_connect_swapped (G_OBJECT (button), "event",
                          G_CALLBACK (on_button_press),
                          G_OBJECT (popupmenu));
```

The callback will be responsible for popping up the menu. To pop up a menu, we need a new function:

```
void            gtk_menu_popup                      ( GtkMenu *menu,
                                                      GtkWidget *parent_menu_shell,
                                                      GtkWidget *parent_menu_item,
                                                      GtkMenuPositionFunc func,
                                                      gpointer data,
                                                      guint button,
                                                      guint32 activate_time);
```

This function displays a menu and makes it available for selection. It is exists precisely for the purpose of displaying context-sensitive menus. For normal use, you would supply `NULL` for the `parent_menu_shell`, `parent_menu_item`, `func` and `data` parameters.

The `button` parameter should be the mouse button pressed to initiate the menu popup. If the menu popup was initiated by something other than a mouse button press, such as a mouse button release or a key-press, button should be 0.

The API documentation states that the `activate_time` parameter is used to conflict-resolve initiation of concurrent requests for mouse/keyboard grab requests. To function properly, this needs to be the time stamp of the user event (such as a mouse click or key press) that caused the initiation of the popup.

Putting this together, the callback `on_button_press()` should be

```
static gboolean on_button_press    ( GtkWidget *widget,
                                     GdkEvent *event )
{
    if (event->type == GDK_BUTTON_PRESS) {
        GdkEventButton *bevent = (GdkEventButton *) event;
        gtk_menu_popup (GTK_MENU (widget), NULL, NULL, NULL, NULL,
                        bevent->button, bevent->time);
        // Tell calling code that we have handled this event
        return TRUE;
    }
    // Tell calling code that we have not handled this event; pass it on.
    return FALSE;
}
```

This callback does not care which mouse button was pressed. It gets the event type and checks that it is a button press. If so, it casts it to a button event and uses that event's button and time members in the call to `gtk_menu_popup()`.

The appendix contains a listing for a complete program, named `menu_by_hand.c`, that demonstrates all of the above techniques. All of the menu items share a common callback function for the "activate" signal. This callback pops up a dialog box that shows which item was pressed.

2.4 Specialized Menu Items

The preceding menu items were standard items. The `GtkMenuItem` class has two subclasses, `GtkImageMenuItem` and `GtkCheckMenuItem`, which has a subclass `GtkRadioMenuItem`. We will see how to use these after we cover the `GtkUIManager`.

3 Toolbars

Toolbars provide quick access to commonly used actions. They are containers that should be populated with instances of the `GtkToolItem` class. Usually you will insert toolbar buttons into a toolbar. Toolbar buttons belong to the `GtkToolButton` class, which is a subclass of `GtkToolItem`. There are also two subclasses of the toolbutton class: `GtkMenuToolButton` and `GtkToggleToolButton`, which has a subclass `GtkRadioToolButton`.

A toolbar is created with only a single function:

```
GtkWidget *    gtk_toolbar_new                   ( void);
```

Once it is created, tool items can be inserted into it, using

```
void           gtk_toolbar_insert                ( GtkToolbar *toolbar,
                                                   GtkToolItem *item,
                                                   gint pos);
```

Although the API documentation lists other methods of packing tool items into a toolbar, these have been deprecated since version 2.4 and should be not be used. This inserts the tool item at position `pos`. If `pos` is 0 the item is prepended to the start of the toolbar. If `pos` is negative, the item is appended to the end of the toolbar. Therefore, if items are inserted successively into a toolbar passing -1 as `pos`, they will appear in the toolbar in left to right order.

There are few other things you should do when creating the toolbar itself. Many of the methods of the `GtkToolBar` class should be avoided, either because they are deprecated, or because they violate the principle of respecting the user's style preferences. For this reason, we will not discuss them here.

Although tool items can be created with

```
GtkToolItem *  gtk_tool_item_new                 ( void);
```

we will have little use for this function, as we will be putting only buttons and separators into our toolbars. Each of these has its own specialized constructors. To create a toolbar button, you can use any of two different methods:

```
GtkToolItem *  gtk_tool_button_new               ( GtkWidget *icon_widget,
                                                   const gchar *label);
GtkToolItem *  gtk_tool_button_new_from_stock    ( const gchar *stock_id);
```

The first method requires that you supply a custom icon and label; the second lets you pick a stock id. You can use any stock item from the documentation. As we have not yet covered how to create icons, we will stay with the second method in the examples that follow. The following code fragment creates a toolbar and a few toolbar buttons using stock items and puts them into a toolbar.

```
GtkWidget    *toolbar     = gtk_toolbar_new();
GtkToolItem  *new_button  = gtk_tool_button_new_from_stock(GTK_STOCK_NEW);
GtkToolItem  *open_button = gtk_tool_button_new_from_stock(GTK_STOCK_OPEN);
GtkToolItem  *save_button = gtk_tool_button_new_from_stock(GTK_STOCK_SAVE);
gtk_toolbar_insert(GTK_TOOLBAR(toolbar), new_button, -1);
gtk_toolbar_insert(GTK_TOOLBAR(toolbar), open_button, -1);
gtk_toolbar_insert(GTK_TOOLBAR(toolbar), save_button, -1);
```

You can create separator items using

```
GtkToolItem *  gtk_separator_tool_item_new        ( void);
```

This creates a vertical separator in horizontal toolbar. If for some reason you want the buttons to the right of the separator to be grouped at the far end of the toolbar, you can use the separator like a "spring" to push them to that end by setting its "expand" property to `TRUE` and its "draw" property to `FALSE`, using the sequence

```
separator = gtk_separator_tool_item_new();
gtk_tool_item_set_expand(GTK_TOOL_ITEM(separator), TRUE);
gtk_separator_tool_item_set_draw(GTK_SEPARATOR_TOOL_ITEM(separator), FALSE);
```

The "expand" property is inherited from `GtkToolItem` whereas the "draw" property is specific to the separator. If we want to add a separator followed by an EXIT button, we would append the sequence below to the preceding code (provided we declared the separator and button)

```
separator   = gtk_separator_tool_item_new();
exit_button = gtk_tool_button_new_from_stock(GTK_STOCK_QUIT);
gtk_toolbar_insert(GTK_TOOLBAR(toolbar), exit_button, -1);
```

If we want the button to be pushed to the right, we would add the two function calls above to this.

Toolbar buttons are buttons, not items, and therefore they emit a "clicked" signal. To respond to button clicks, connect a callback to the button as if it were an ordinary button, such as

```
g_signal_connect(G_OBJECT(exit_button), "clicked",
                 G_CALLBACK(gtk_main_quit), NULL);
```

A complete program showing how to create a simple toolbar using this manual method is in the Appendix.

4 The GtkUIManager

A `GtkUIManager` is an object that can dynamically construct a user interface consisting of menus and toolbars from a *UI description*. A UI description is a specification of what menu and toolbar widgets should be present in an application and is described in an XML format. A `GtkUIManager` makes it possible to change menus and toolbars dynamically using what is called *UI merging*.

4.1 Actions

The principal objects manipulated by a `GtkUIManager` are *actions*, which are instances of the `GtkAction` class. Actions represent operations that the user can perform. Associated with an action are

- a callback function
- its name
- a label
- an accelerator
- a flag indicating whether the label is a stock id

- a tooltip
- a toolbar label
- a flag indicating whether it is sensitive
- a flag indicating whether it is visible

The callback function is the function that is executed when the action is activated. The action name is how it is referred to, not what appears in a menu item or toolbar button, which is its label. Actions can have associated keyboard accelerators and tooltips. Their visibility and sensitivity can be controlled as well. The idea is that you can create actions that the `GtkUIManager` can bind to proxies such as menu items and toolbar buttons.

The `GtkAction` class has methods to create icons, menu items and toolbar items representing itself, as well as get and set methods for accessing and changing its properties.

The `GtkAction` class also has two subclasses: `GtkToggleAction` and `GtkRecentAction`. The `GtkToggleAction` class has a `GtkRadioAction` subclass. These correspond to toggle buttons and radio buttons respectively.

4.2 UI Definitions

You can specify the set user interface action elements in your application with an XML description called a UI definition. A UI definition is a textual description that represents the actions and the widgets that will be associated with them. It must be bracketed by the pair of tags

```
<ui>
</ui>
```

Within these tags you describe your user interface in a hierarchical way, by defining menubars, which would contain menus, which in turn contain menus and menu items, toolbars, which would contain tool items, and popup menus, which can contain menus and menu items. The set of tags that can be used in these UI definitions, with their descriptions and attributes, is

Tag	**Description**	**Attributes**	**Closing Tag**
`<menubar>`	a `GtkMenuBar`	name, action	yes
`<toolbar>`	a `GtkToolbar`	name, action	yes
`<popup>`	a toplevel `GtkMenu`	name, action, accelerators	yes
`<menu>`	a `GtkMenu` attached to a menuitem	name, action, position	yes
`<menuitem>`	a `GtkMenuItem` subclass, the exact type depends on the action	name, action, position, always-show-image	no
`<toolitem>`	a `GtkToolItem` subclass, the exact type depends on the action.	name, action, position	no
`<separator>`	a `GtkSeparatorMenuItem` or `GtkSeparatorToolItem`	name, action, expand	no
`<accelerator>`	a keyboard accelerator	name, action	no
`<placeholder>`	a placeholder for dynamically adding an item	name, action	yes

Example

The following example shows a UI definition of a menubar and its submenus.

```
<ui>
  <menubar name='MainMenu'>
    <menu name='File' action='FileMenu'>
      <menuitem name='Open'  action='Open'  always-show-image='true'/>
      <menuitem name='Close' action='Close' always-show-image='true'/>
      <separator/>
      <menuitem name='Exit' action='Exit'/>
    </menu>
    <menu action='ViewMenu'>
      <menuitem name='ZoomIn' action='ZoomIn'/>
      <menuitem name='ZoomOut' action='ZoomOut'/>
      <separator/>
      <menuitem name='FullScreen' action='FullScreen'/>
      <separator/>
      <menuitem name='JustifyLeft' action='JustifyLeft'/>
      <menuitem name='JustifyCenter' action='JustifyCenter'/>
      <menuitem name='JustifyRight' action='JustifyRight'/>
      <menu action='IndentMenu'>
        <menuitem action='Indent'/>
        <menuitem action='Unindent'/>
      </menu>
    </menu>
  </menubar>
</ui>
```

Notes

- Some tags must have a closing tag and some do not. Those with no closing tag are `<menuitem>`, `<toolitem>`, `<separator>`, and `<accelerator>`. A tag that does not have a closing tag must have a forward slash preceding its right bracket: "`/>`". All other tags can have content.
- All attribute values are plain text strings.
- All UI elements have a *name* and *action* attribute. The name is optional; if a name is not specified, the action is used as its own name. If for some reason, neither the name nor the action attribute are specified, the name of the element is used when referring to it. The name and action attributes must not contain '/' characters after parsing, nor double quotes.
- Menus, menu items, and toolitems have a *position* attribute with two possible values: "top" and "bottom". If this attribute is missing, its default value of "bottom" is used. This attribute determines where the element is placed relative to its siblings. If position="top" then when it is added to the parent container, it will be placed before its siblings, meaning to the left in a horizontal container or above the siblings in a vertical container. If it is "bottom" then it will be placed to the right of the siblings or below them in horizontal and vertical containers respectively.
- The elements are added to the UI interface in the order in which they appear in the XML string. In the above example, the JustifyLeft action precedes the JustifyCenter action, so the former will appear above the latter. If however, the UI was defined as follows:

```
    <menuitem name='JustifyLeft'  action='JustifyLeft'    position='top'/>
    <menuitem name='JustifyCenter action='JustifyCenter'  position='top'/>
    <menuitem name='JustifyRight' action='JustifyRight'   position='top'/>
```

then they would appear in the order

```
JustifyRight
JustifyCenter
JustifyLeft
```

because each time the element is inserted, the `position='top'` attribute forces it to be above its siblings. The 'top" attribute converts the packing into a stack push operation in effect.

- Menu items have a "always-show-image" attribute with two possible values: "true" and "false". If this attribute is true, then menu item icons will always be visible, overriding any user settings in the desktop environment.
- Separators can have an expand attribute, with the value "true" or "false". If it is set to "true" then the separator will expand to take up extra space in the parent container and become invisible. Otherwise it is drawn as a thin line, depending on the user's theme..
- Submenus are created in a different way using the XML than they are when constructing these programmatically. Remember that submenus are attached to menu items, which are contained in the parent menu. Here, a `<menu>` element can be a direct child of a parent `<menu>` element. In the example above, the menu whose action is '`IndentMenu`' is a child of the '`ViewMenu`'.
- Placeholders are merged into their parent containers invisibly. If a placeholder has child elements X, Y, and Z, these will be at the same level of the tree as the placeholder itself. An example later will illustrate the utility of this feature.
- Finally, observe the hierarchy implicit in the UI definition. As a matter of style, you should indent these using standard rules of indentation, to make them easier to read.

We can create a toolbar definition in a similar way:

```
<ui>
  <toolbar name = 'ToolBar' action="ToolBarAction">
      <placeholder name="ExtraToolItems">
      <separator/>
      <toolitem name="ZoomIn" action="ZoomIn"/>
      <toolitem name="ZoomOut" action="ZoomOut"/>
      <separator/>
      <toolitem name='FullScreen' action='FullScreen'/>
      <separator/>
      <toolitem name='JustifyLeft' action='JustifyLeft'/>
      <toolitem name='JustifyCenter' action='JustifyCenter'/>
      <toolitem name='JustifyRight' action='JustifyRight'/>
    </placeholder>
  </toolbar>
</ui>
```

Notice that the tool items have the same action names as some of the menu items. This is how you can create multiple proxies for the same action. When the `GtkUIManager` loads these descriptions, and you take the appropriate steps in your program, they will be connected to the same callback functions.

Notice also that there is a placeholder in the toolbar defined above. We can use that placeholder to dynamically add more tool items in that position. It does not occupy space in the toolbar widget; it just marks a position to be accessed, so there is no downside to putting these placeholders into the UI definition.

4.3 Action Groups

Actions are organized into *action groups*. An action group is essentially a map from names to `GtkAction` objects. Action groups are the easiest means for adding actions to a UI manager object.

In general, related actions should be placed into a single group. More precisely, since the UI manager can add and remove actions as groups, if the interface is supposed to change dynamically, then all actions that should be available in the same state of the application should be in the same group. It is typical that multiple action groups are defined for a particular user interface. Most nontrivial applications will make use of multiple groups. For example, in an application that can play media files, when a media file is open, the playback actions (play, pause, rewind, etc) would be in a group that could be added and removed as needed.

4.4 Creating the UI

The basic steps in creating the UI are to

1. Define the UI in an XML format, either in a separate file or in a constant string within the source code.
2. Create the actions and action groups.
3. Create a UI manager.
4. Add the action groups to the UI manager.
5. Extract the accelerators from the UI manager and add them to the top-level window.
6. Add the UI definition to the UI manager from the file or string.
7. Get the menubar and toolbar widgets from the UI manager and pack them into the window.
8. Create the callbacks referenced in the action objects created in step 2.

We will next describe how to program each of steps 2 through 7.

4.4.1 Creating Actions and Action Groups

The function to create an action group is

```
GtkActionGroup *gtk_action_group_new             ( const gchar *name);
```

The name argument can be used by various methods for accessing this particular action group. It should reflect what this particular group's purpose or common feature is.

Actions are added to an action group in one of two ways. You can add them one at a time, or as an array of related actions:

```
void           gtk_action_group_add_action       ( GtkActionGroup *action_group,
                                                   GtkAction *action);

void           gtk_action_group_add_actions      ( GtkActionGroup *action_group,
                                                   const GtkActionEntry *entries,
                                                   guint n_entries,
                                                   gpointer user_data);
```

The problem with the first method is that it is tedious to add actions one by one, and that this method does not provide a means to add the accelerators for the actions without additional steps. Even if there is just a single action in the group, it is more convenient to use the second function. To use `gtk_action_group_add_actions()`, you first have to create a constant array of `GtkActionEntry` structures. The function requires the name of the array and its length as second and third arguments, and the user data to be passed to all callback functions that can be called for the actions in this group.

A `GtkActionEntry` structure is defined by

```
typedef struct  {
   const gchar     *name;
   const gchar     *stock_id;
   const gchar     *label;
   const gchar     *accelerator;
   const gchar     *tooltip;
   GCallback  callback;
} GtkActionEntry;
```

The members have the following meanings:

`name`	The name of the action.
`stock_id`	The stock id for the action, or the name of an icon from the icon theme.
`label`	The label for the action. This field should typically be marked for translation, see `gtk_action_group_set_translation_domain()`. If label is `NULL`, the label of the stock item with id `stock_id` is used.
`accelerator`	The accelerator for the action, in the format understood by `gtk_accelerator_parse()`.
`tooltip`	The tooltip for the action. This field should typically be marked for translation, see `gtk_action_group_set_translation_domain()`.
`callback`	The function to call when the action is activated.

The `name` must match the name of the action to which it corresponds in the UI definition. The stock_id can be `NULL`, as can the label. The accelerator syntax is very flexible. You can specify control keys, function keys and even ordinary characters, for example, using "`<Control>a`", "`Ctrl>a`", "`<ctrl>a`", or "`<Shift><Alt>F1`", "`<Release>z`", or "`minus`", to name a few. If you use a stock item, it is not necessary to supply an accelerator, unless you want to override the default one. The `tooltip` is a string that will appear when the cursor hovers over the proxy for this action entry.

Below is an example of a declaration of a small array of `GtkActionEntry` structures.

```
static const GtkActionEntry file_entries[] = {
  { "FileMenu", NULL, "_File" },

  { "Open", GTK_STOCK_OPEN,
    "_Open", "<control>O",
    "Open a file", G_CALLBACK (on_open_file) },

  { "Close", GTK_STOCK_CLOSE,
    "_Close", "<control>W",
    "Close a file", G_CALLBACK (on_close_file) },

  { "Exit", GTK_STOCK_QUIT,
    "E_xit", "<control>Q",
    "Exit the program", G_CALLBACK (gtk_main_quit) },
};
```

Notice that the `FileMenu` action does not have a tooltip nor a callback. The `Open`, `Close`, and `Exit` actions have both a mnemonic label and an accelerator.

Having defined this array, it can be added to a group as follows:

```
action_group = gtk_action_group_new ("Common_Actions");
gtk_action_group_add_actions (action_group, file_entries,
                              G_N_ELEMENTS (file_entries),
                              (gpointer) (&appState));
```

The `G_N_ELEMENTS()` macro returns the number of elements in its array argument. In the code snippet above, the address of a variable named `AppState` would be passed as user data to the callback functions of each of the actions added to the group. Because the same user data pointer will be passed to all of the callbacks of the actions in this group, you need to design a structure that will contain the data that all of these callbacks need, and pass a pointer to that common structure to this function. The alternative is to attach data properties to the various objects and access these properties within the callbacks. That approach results in code that is harder to maintain.

Multiple action entry arrays can be added to a single action group. In fact, you probably will need to do this, because toggle actions and radio actions must be defined differently. A `GtkToggleActionEntry` contains all of the members of a `GtkActionEntry`, as well as an additional boolean flag:

```
typedef struct  {
    const gchar     *name;
    const gchar     *stock_id;
    const gchar     *label;
    const gchar     *accelerator;
    const gchar     *tooltip;
    GCallback  callback;
    gboolean   is_active;
} GtkToggleActionEntry;
```

The `is_active` flag indicates whether or not the action is active or inactive. To add toggle action entries to an action group you need to use a separate function designed for that purpose:

```
void        gtk_action_group_add_toggle_actions (GtkActionGroup *action_group,
                                                 const GtkToggleActionEntry *entries,
                                                 guint n_entries,
                                                 gpointer user_data);
```

As you can see, its prototype differs from gtk_action_group_add_actions() only in that it expects an array of type `GtkToggleActionEntry`. To illustrate, we could define an array with a single toggle action entry:

```
static const GtkToggleActionEntry toggle_entries[] = {
  { "FullScreen",
     GTK_STOCK_FULLSCREEN,
    "_Full Screen", "F11",
    "Switch between full screen and windowed mode",
     G_CALLBACK (on_full_screen), FALSE }
};
```

and add it to the same group as above with

```
gtk_action_group_add_toggle_actions (action_group, toggle_entries,
                                     G_N_ELEMENTS (toggle_entries),
                                     (gpointer) (&appState));
```

GTK+ defines radio action entries separately. Usually you use radio buttons when there are three or more alternatives. If there are just two, a toggle is the cleaner interface element. Because radio actions can have more than two values, the structure's last element is an integer instead of a boolean. It is defined by

```
struct GtkRadioActionEntry {
  const gchar *name;
  const gchar *stock_id;
  const gchar *label;
  const gchar *accelerator;
  const gchar *tooltip;
  gint   value;
};
```

Unlike ordinary actions and toggle actions, which can have different callbacks for each action, radio action entries do not specify a callback function. Furthermore, the last member of this structure is the value that that particular radio action has. If for example, there are three radio actions for how text is to be aligned, left, right, or centered, then one would have the value 0, the next, 1, and the third, 2. An example of an array of radio action entries is below.

```
static const GtkRadioActionEntry radio_entries[] = {
  { "JustifyLeft", GTK_STOCK_JUSTIFY_LEFT,
    "_Left", NULL, "Left justify text", 0 },

  { "JustifyCenter", GTK_STOCK_JUSTIFY_CENTER,
    "_Center", NULL, "Center the text", 1 },

  { "JustifyRight", GTK_STOCK_JUSTIFY_RIGHT,
    "_Right", NULL, "Right justify the text", 2 }
};
```

Because radio action entries do not have a callback function as a member, the function to add radio actions to an action group specifies a single callback to be used for all of the actions in the array of radio actions being added. This is the callback that will be called in response to the "changed" signal:

```
void        gtk_action_group_add_radio_actions ( GtkActionGroup *action_group,
                                                  const GtkRadioActionEntry *entries,
                                                  guint n_entries,
                                                  gint value,
                                                  GCallback on_change,
                                                  gpointer user_data);
```

Also, this function has another parameter that specifies the value that should be active initially. It is either one of the values in the individual radio action entries, or -1 to indicate that none should be active to start. We could add the `radio_entries` action array to our group with

```
gtk_action_group_add_radio_actions (action_group,
                                    radio_entries,
                                    G_N_ELEMENTS (radio_entries),
                                    0, G_CALLBACK (on_radio_changed),
                                    NULL);
```

specifying that the `JustifyLeft` action is the initial value.

4.4.2 Creating the UIManager and Adding the Action Groups

A `GtkUIManager` is created with the function

```
GtkUIManager *  gtk_ui_manager_new                ( void);
```

This creates a UI manager object that can then be used for creating and managing the application's user interface. It is now ready to be populated with the action groups that you already defined. To insert an action group into the UI manager, use

```
void           gtk_ui_manager_insert_action_group ( GtkUIManager *self,
                                                    GtkActionGroup *action_group,
                                                    gint pos);
```

The first argument is the pointer returned by the call to create the UI manager. The second is a pointer to the group to be inserted. The `pos` argument specifies the position in the list of action groups managed by this UI manager. Action groups that are earlier in this list will be accessed before those that are later in this list. A consequence of this is that, if an action with the same name, e.g. "`Open`", is in two different groups, the entry in the group with smaller position will hide the one in the group with larger position. For example, if an "`Open`" action is in groups named `action_group1` and `action_group2`, and `action_group1` is inserted at position 1, and `action_group2` is at position 2, then the entry for the "`Open`" action in `action_group1` will be used by the UI manager when its proxy is activated. If it has a different callback or label or accelerator, these will be associated with this action, not the one in `action_group2`. You can use this feature if you need to change the semantics of a menu item or toolbar button, but not the menu item or button itself.

While we are on the subject of inserting actions, we might as well look at how you can remove an action group, if you have need to do that dynamically. That function is

```
void           gtk_ui_manager_remove_action_group ( GtkUIManager *self,
                                                    GtkActionGroup *action_group);
```

This searches the list of action groups in the UI manager and deletes the one whose pointer is passed to it.

4.4.3 Extracting Accelerators and Adding Them to the Top-Level Window

Accelerators are key combinations that provide quick access to the actions in a window. They are usually associated with the top-level window so that key-presses while that window has focus can be handled by the top-level window's `key_press_event` handler, which can propagate it through the chain of widgets.

The problem is that the accelerators are stored within the UI manager, not the top-level window, when you insert the action groups into it. The UI manager aggregates the accelerators into its private data as action groups are added to it. However it provides a method of extracting them. The set of accelerators can be extracted into a `GtkAccelGroup` object that can be added into a top-level window. The function that does this is

```
GtkAccelGroup *gtk_ui_manager_get_accel_group    ( GtkUIManager *self);
```

The function that adds this group into a top-level window is

```
void           gtk_window_add_accel_group        ( GtkWindow *window,
                                                   GtkAccelGroup *accel_group);
```

The following code-snippet will extract the accelerators and add them to the top-level window:

```
accel_group = gtk_ui_manager_get_accel_group (ui_manager);
gtk_window_add_accel_group (GTK_WINDOW (window), accel_group);
```

4.4.4 Loading the UI Definition

If the UI definition is in a separate file, it can be loaded using the function

```
guint          gtk_ui_manager_add_ui_from_file   ( GtkUIManager *self,
                                    const gchar  *filename,
                                    GError      **error)
```

The first argument is the UI manager pointer, the second, a filename passed as a UTF-8 string, and the last, a pointer to the address of a `GError` object. You should always provide a non-`NULL` pointer here, because file I/O has great potential to fail for many reasons, and your program should handle those failures. If this function is successful, it will return a positive integer called a *merge-id.* Merge-ids will be explained in the next section. If the function fails, for one reason or another, the return value will be zero. It is possible for the return value to be zero even though the error argument is `NULL`. Therefore it is a good idea to check the return value of the function as well as the value of error. The following code fragment tests both conditions and terminates the program with an error message if there is an error:

```
merge_id = gtk_ui_manager_add_ui_from_file(ui_manager, "menu_1.xml",
                                           &error);
if ( error != NULL) {
    error_notification(GTK_WINDOW(window), error);
    g_error_free(error);
    error = NULL;
    return 1;
}
else if ( 0 == merge_id )  {
    error_notification(GTK_WINDOW(window), NULL);
    return 1;
}
```

The `error_notification()` function displays a message dialog with a suitable message.

An alternative to storing the UI definition in a file in the source code tree is to store the UI definition as a C string within a source code file itself. For example, it could be in a header file that is included in the main program. If the UI definition is in a string, then it can be added with the function

```
guint          gtk_ui_manager_add_ui_from_string ( GtkUIManager *self,
                                                   const gchar *buffer,
                                                   gssize length,
                                                   GError **error);
```

The buffer argument is the name of the string containing the UI definition, and the length argument is either -1 or the length of the string in bytes. If the string is `NULL`-terminated, it can be -1, otherwise it must be the actual length. The error argument serves the same purpose as above. The return value is also either a positive integer on success, in which case it is a valid merge-id, or zero on failure.

The following listing shows how to define a UI definition in a static string.

```
static const gchar *ui_constant =
"<ui>"
"  <menubar name='MainMenu'>"
"      <menu action='FileMenu'>"
"      <placeholder name='FilePlace'/>"
"      <separator/>"
"        <menuitem action='Exit'/>"
"      </menu>"
"      <menu action='ViewMenu'>"
"        <menuitem action='ZoomIn'/>"
"        <menuitem action='ZoomOut'/>"
"        <separator/>"
"        <menuitem action='FullScreen'/>"
"        <separator/>"
"      </menu>"
"    </menubar>"
"</ui>";
```

This would then be added to the UI manager with the fragment

```
merge_id = gtk_ui_manager_add_ui_from_string(
                              ui_manager, ui_constant, -1, &error);
if ( error != NULL) {
     error_notification(GTK_WINDOW(window), error);
     g_error_free(error);
     error = NULL;
     return 1;
}
else if ( 0 == merge_id )  {
    error_notification(GTK_WINDOW(window), NULL);
    return 1;
}
```

4.4.5 Getting the Widgets

The last step is to retrieve the widgets that the UI manager created when the UI definition was loaded into it, and pack those widgets into the window where you want them to be. This is where the names of the UI definition elements come into play. The UI manager can find a widget for you when you give it the absolute pathname of the element that you want to construct. The absolute pathname is a string starting with a forward slash '/', much like a file's absolute pathname, with a sequence of the ancestor elements in the XML tree of that element.

Elements which don't have a name or action attribute in the XML (e.g. `<popup>`) can be addressed by their XML element name (e.g. `"popup"`). The root element (`"/ui"`) can be omitted in the path.

As an example, the absolute pathname of the `FileMenu` in the UI definition above is "`/MainMenu/FileMenu`".

The function

```
GtkWidget *    gtk_ui_manager_get_widget         ( GtkUIManager *self,
                                                   const gchar *path);
```

finds the widget that the UI manager constructed, whose name matches the pathname that you give it. If you give it the name of a menubar, you get a menubar widget with its entire subtree. If you give it the name of a menu, you get the menu item to which the menu is attached, not the menu.

If our UI definition had a menubar and toolbar at the top level named "`MainMenu`" and "`MainToolBar`" respectively, we could get them from the UI manager using

```
menubar = gtk_ui_manager_get_widget (ui_manager, "/MainMenu");
toolbar = gtk_ui_manager_get_widget (ui_manager, "/MainToolBar");
```

We could then pack these into a `GtkVBox` one below the other in our main window, and we would be finished, except of course for defining all of the required callback functions.

Note. The widgets that are constructed by a UI manager are not tied to the life-cycle of that UI manager. It does acquire a reference to them, but when you add the widgets returned by this function to a container or if you explicitly ref them, they will survive the destruction of the UI manager. (Read the notes on memory management in GTK if you are unfamiliar with these concepts.)

Lastly, you can tell the UI manager to create tear-off menus if you want, using

```
void            gtk_ui_manager_set_add_tearoffs   ( GtkUIManager *self,
                                                    gboolean add_tearoffs);
```

By passing `TRUE`, all menus (except popup menus) will have the tear-off property.

4.5 UI Merging

One of the most powerful features of the UI manager is its ability to dynamically change the menus and toolbars by overlaying or inserting menu items or toolbar items over others and removing them later. This feature is called *UI merging*. The ability to merge elements is based on the use of the pathnames to the UI elements defined in the UI definition, and merge-ids.

A merge-id is an unsigned integer value that is associated with a particular UI definition inside the UI manager. The functions that add UI definitions into the UI manager, such as `gtk_ui_manager_add_ui_from_string()` and `gtk_ui_manager_add_ui_from_file()`, return a merge-id that can be used at a later time, for example, to remove that particular UI definition. The function that removes a UI definition is

```
void            gtk_ui_manager_remove_ui          ( GtkUIManager *self,
                                                    guint merge_id);
```

This is given the merge-id of the UI definition to be removed. For example, if I create a UI with the call

```
merge_id = gtk_ui_manager_add_ui_from_string(
                                  ui_manager, ui_toolbar, -1, &error);
```

and I later want to remove the toolbar from the window, I would call

```
gtk_ui_manager_remove_ui(ui_manager, merge_id);
```

In order to add an element such as a toolbar in one part of the code, and later remove it in a callback function, you would need to make the merge-id either a shared variable, or attach it as a property to a widget that the callback is passed.

There is a third function for adding a new element to the user interface:

```
void            gtk_ui_manager_add_ui             ( GtkUIManager *self,
                                                    guint merge_id,
                                                    const gchar *path,
                                                    const gchar *name,
                                                    const gchar *action,
                                                    GtkUIManagerItemType type,
                                                    gboolean top);
```

The parameters have the following meanings:

self	a `GtkUIManager`
merge_id	the merge-id for the merged UI
path	a path
name	the name for the added UI element
action	the name of the action to be proxied, or `NULL` to add a separator. [allow-none]
type	the type of UI element to add.
top	if `TRUE`, the UI element is added before its siblings, otherwise it is added after its siblings.

This function can add a single element to the UI, such as a menu item, a toolbar item, a menu, or a menubar. It cannot add an entire UI definition such as the ones contained in the strings defined above. Furthermore, it cannot be used to insert an element in a place where such an element cannot be inserted. For example, you cannot insert a toolbar inside a menu, or a menu inside a menu, but you can insert a menu item in a menu, or a menu in a menubar.

In order to use this function, you need a merge-id to give to it. It will assign associate the new UI element to this merge-id so that it can be removed at a later time. New merge-ids are created with

```
guint               gtk_ui_manager_new_merge_id         (GtkUIManager *self);
```

The third parameter is the absolute path name to the position at which you want to add the new UI element. For example, if you want to insert a new menu item at the top of the `File` menu, the path would be "`/MainMenu/FileMenu`". The fourth parameter is a name that you want this item to have for future access and the fifth is the name fo the action, which must exist already, that should be connected to this element.

The type must be a member of the `GtkUIManagerItemType`, which has the following values:

```
GTK_UI_MANAGER_AUTO
GTK_UI_MANAGER_MENUBAR
GTK_UI_MANAGER_MENU
GTK_UI_MANAGER_TOOLBAR
GTK_UI_MANAGER_PLACEHOLDER
GTK_UI_MANAGER_POPUP
GTK_UI_MANAGER_MENUITEM
GTK_UI_MANAGER_TOOLITEM
GTK_UI_MANAGER_SEPARATOR
GTK_UI_MANAGER_ACCELERATOR
```

Their meanings should be self-explanatory, except for the first. You can use `GTK_UI_MANAGER_AUTO` as the type to let GTK decide the type of the element that can be inserted at the indicated path. Lastly, if you want the element to be above the element that is currently in that position, you set top to `TRUE`, otherwise `FALSE`.

As an example, suppose that I want to add a `Print` menu item in my `File` menu just below the `Open` menu item. I could use the following code fragment, assuming that I have already defined an action named `Print`:

```
merge_id =  gtk_ui_manager_new_merge_id ( ui_manager );
gtk_ui_manager_add_ui ( ui_manager,  merge_id,
                        "/MainMenu/FileMenu/Open",
                        "Print", "Print",
                        GTK_UI_MANAGER_MENUITEM,
                        FALSE);
```

This will insert the `Print` menu item into the proper position.

Assuming that your menu is to be changed dynamically, these steps will not be enough to make the menu elements appear dynamically. The UI manager does not handle the task of packing new toolbars or menubars into their places in the window. However, it does emit the "add-widget" signal for each generated menubar and toolbar. Your application can respond to this signal with a callback function that can pack the UI element into the appropriate position. Therefore, two additional steps are needed by a program that adds and removes menubars or toolbars:

- Create a callback function to pack these widgets into the parent container, and
- Connect the "add-widget" signal emitted by the UI manager to this callback.

The callback for this signal has the prototype

```
void           user_function                    ( GtkUIManager *merge,
                                                  GtkWidget    *widget,
                                                  gpointer      user_data);
```

The first parameter is the UI manager emitting the signal, the second is the widget that has been added, and the third is optional user data. We can use the third parameter to pass the parent container to the callback, so that it can pack the widget into it. For example:

```
void menu_add_widget       ( GtkUIManager *ui_manager,
                             GtkWidget *widget,
                             GtkContainer *container)
{
    gtk_box_pack_start (GTK_BOX (container), widget, FALSE, FALSE, 0);
    gtk_widget_show (widget);
}
```

This will pack the menubar or toolbar after any other widgets in the parent. It must show the widget to realize it. This callback can be connected to the "add-widget" signal in the main program with the call

```
g_signal_connect (ui_manager, "add_widget",
                  G_CALLBACK (menu_add_widget),
                  menu_box);
```

assuming that `menu_box` is a `GtkBox` of some kind that the menu or toolbars should be packed into.

4.5.1 Example

An example is in order. Not all of the pieces will be presented, because the program can become quite large. In fact at this point, the code should be decomposed into separate files with well-defined tasks. Suppose that we want to do two things dynamically in our application:

When the program starts up, there is an `Open` menu item in the `File` menu, but no `Close` item. The `Close` item is present only when something has been opened. Conversely, there is no `Open` menu item when something is open. Therefore, these two menu items will be swapped alternately as they are activated.

When nothing is open in the application, there is no toolbar. When the `Open` item is activated, a special toolbar will be placed below the menubar. When the `Close` item is activated, the toolbar is removed.

In order to do this, we create a general UI definition that will always be present, with a placeholder for the `Open`/`Close` menu items. Just before showing all of the widgets in the main program, we create an `Open` menu item and insert it into the UI, saving its merge-id. The UI definition that never changes is:

```
const gchar *ui_constant =
"<ui>"
"  <menubar name='MainMenu'>"
"      <menu action='FileMenu'>"
"      <placeholder name='OpenClose'/>"
"      <separator/>"
"      <menuitem action='Exit'/>"
"    </menu>"
"    <menu action='ViewMenu'>"
"      <menuitem action='ZoomIn'/>"
"      <menuitem action='ZoomOut'/>"
"      <separator/>"
"      <menuitem action='FullScreen'/>"
"      <separator/>"
"      <menuitem action='JustifyLeft'/>"
"      <menuitem action='JustifyCenter'/>"
"      <menuitem action='JustifyRight'/>"
"    </menu>"
"  </menubar>"
"</ui>";
```

Notice that there is a placeholder where the `Open` and `Close` menu items should be. This UI definition is loaded in the main program and error-checked:

```
// Add the UI defined from the strings in uim02_defs.h  to the manager
// If a 0 is returned, something went wrong, so bail out. If in addition
// error is set, then display the message dialog to the user.
merge_no_openclose_id = gtk_ui_manager_add_ui_from_string(
                             ui_manager, ui_constant, -1, &error);
if ( 0 == merge_no_openclose_id ) {
    if ( error != NULL)
    {
        error_notification(GTK_WINDOW(window), error);
        g_error_free(error);
        error = NULL;
    }
    return 1;
}
```

Then we create a new merge-id for the `Open` item, inserting it into the proper place:

```
// Get a merge id to use for the Open/Close menu item
appState.merge_id =  gtk_ui_manager_new_merge_id ( ui_manager );

// Add the Open menu item directly above the Exit menu item in the
// File menu and make appState.merge_id represent this state of things.
gtk_ui_manager_add_ui ( ui_manager,  appState.merge_id,
                                 "/MainMenu/FileMenu/OpenClose",
                                 "Open", "Open",
                                 GTK_UI_MANAGER_MENUITEM,
                                 TRUE);
```

Finally, we set up the callback for the “add-widget” signal, using the callback shown above:

```
g_signal_connect (ui_manager, "add_widget",
                          G_CALLBACK (menu_add_widget),
                          menu_box);
```

Of course prior to doing all of this we created the action groups and added these to the UI manager.

The next step is to define the callbacks. In the callback for the `Open` action, we remove the UI for the `Open` action, add a UI in the same position for the `Close` action, and add a UI for the toolbar from a string in the program. This follows:

```
void on_open_file          ( GtkAction* action, gpointer pdata)
{
    state * appstate = (state*) pdata;
    GError *error;

    appstate->isFileOpen = TRUE;
    gtk_ui_manager_remove_ui( appstate->uim,
                              appstate->merge_id );
    gtk_ui_manager_add_ui ( appstate->uim,
                               appstate->merge_id,
                               "/MainMenu/FileMenu/OpenClose",
                               "Close", "Close",
                               GTK_UI_MANAGER_MENUITEM,
                               TRUE);
    appstate->toolbar_id = gtk_ui_manager_add_ui_from_string (
                              appstate->uim,
                              ui_toolbar,
                              -1,
                              &error);
}
```

Notice that the application state keeps track of whether or not a file is open with a flag named `isFileOpen`. This callback begins by removing the UI associated with the merge-id `appstate->merge_id`, which is the one assigned when we added the `Open` menu item to the UI. It then adds a menu item for the `Close` action at the placeholder where the `Open` item was, using the same merge-id that was used for the `Open` item. Finally, it reads a UI definition for a toolbar (not shown here) from a UI definition string named `ui_toolbar`, and assigns the returned merge-id to a member variable of the application state named `toolbar_id`. This can be used in the callback for the `Close` action.

The callback for the `Close` action must do the reverse of these steps. It must remove the `Close` menu item and the toolbar, and add the `Open` item where the `Close` item was. That code is:

```
void on_close_file (  GtkAction* action, gpointer pdata)
{
    state * appstate = (state*) pdata;
    appstate->isFileOpen = FALSE;
    gtk_ui_manager_remove_ui(appstate->uim,
                                 appstate->merge_id );
    gtk_ui_manager_remove_ui(appstate->uim,
                                 appstate->toolbar_id );
    gtk_ui_manager_add_ui ( appstate->uim,
                               appstate->merge_id,
                               "/MainMenu/FileMenu/OpenClose",
                               "Open", "Open",
                               GTK_UI_MANAGER_MENUITEM,
                               TRUE);
}
```

Neither of these functions error-checks the calls to read the UI definition, which they should. Neither does any real work of course. They just show how to change the interface.

4.5.2 Example

We can add several elements at once or add a submenu using the same method as above. Suppose the existing UI is defined by the following XML:

```
const gchar *ui_constant =
"<ui>"
"  <menubar name='MainMenu'>"
"    <menu action='ViewMenu'>"
"      <menuitem action='ZoomIn'/>"
"      <menuitem action='ZoomOut'/>"
"      <separator/>"
"      <menuitem action='FullScreen'/>"
"    </menu>"
"  </menubar>"
"</ui>";
```

and that we define the following subtree:

```
const char *ui_justify =
"<ui>"
"  <menubar name='MainMenu'>"
"      <menu action='ViewMenu'>"
"          <placeholder action='Justify' >"
"              <menuitem action='JustifyLeft'/>"
"              <menuitem action='JustifyCenter'/>"
"              <menuitem action='JustifyRight'/>"
"          </placeholder>"
"      </menu>"
"  </menubar>"
"</ui>";
```

If we merge this UI definition into our existing tree of UI elements using

```
appstate->justifymerge_id = gtk_ui_manager_add_ui_from_string (
                                      appstate->uim,
                                      ui_justify,
                                      -1,
                                      &error);
```

then the three items `JustifyLeft`, `JustifyCenter`, and `JustifyRight`, will be inserted into the `View` menu below all other items that are currently there, saving the merge-id of those three items in the variable `appstate->justifymerge_id`. If we change the placeholder tag to a menu tag, then a submenu will be inserted instead.

4.6 Controlling Positions of Merged Elements

When you use the `gtk_ui_manager_add_ui()` function to add a UI element, you can specify the exact position where you want that element to be placed in the UI through two parameters: the absolute pathname to the position in the XML description of the UI, and the top parameter, which when `TRUE`, forces it to be before the element currently there, and when `FALSE`, after it.

But when you use the either `gtk_ui_manager_add_ui_from_string()` or `gtk_ui_manager_add_ui_from_file()`, you do not specify the position at which the inserted subtree should be placed, nor is there a parameter to indicate whether it should precede or follow what is currently there. The way to control the exact placement is by using the `position` attribute in the UI definition of the element to be merged. For example, if you want to add a submenu to an existing menu, by default it will be appended to its siblings in the tree. If you do not want it to be at the bottom of the menu to which it is added, then you have to use position parameters to control this.

Suppose that we want to add a `Justify` menu to the top of the `View` menu. We can define the `Justify` submenu with the XML

```
"<ui>"
"  <menubar name='MainMenu'>"
"      <menu action='ViewMenu'>"
"          <menu action='Justify' position='top'>"
"              <menuitem action='JustifyLeft'/>"
"              <menuitem action='JustifyCenter'/>"
"              <menuitem action='JustifyRight'/>"
"              <menu action='IndentMenu'>"
"                  <menuitem action='Indent'/>"
"                  <menuitem action='Unindent'/>"
"              </menu>"
"          </menu>"
"      </menu>"
"  </menubar>"
```

Note that the `Justify` menu element has `position='top'`. If the existing UI definition was loaded from the string

```
"<ui>"
"  <menubar name='MainMenu'>"
"      <menu name='ViewMenu' action='ViewMenu'>"
"          <placeholder name='JMPlace' />"
"          <menuitem action='ZoomIn'     always-show-image='true'/>"
"          <menuitem action='ZoomOut'    always-show-image='true'/>"
"          <separator/>"
"          <menuitem action='FullScreen' always-show-image='true'/>"
"          <separator/>"
"      </menu>"
"  </menubar>"
```

then if we call

```
    justifymerge_id = gtk_ui_manager_add_ui_from_string (
                                 appstate->uim,
                                 ui_justify,
                                 -1,
                                 &error);
```

the `Justify` menu will be inserted above its siblings, the first of which is the `ZoomIn` menu item. The placeholder has no use in this context. It just serves to remind us where it will be placed.

4.7 What Else?

The preceding example adds a toolbar and a single menu item dynamically. You can use these same ideas for more complex changes, such as adding submenus, or adding a menu and toolbar simultaneously as the application's state changes. The demo program `uimanager03` in the demos/menus directory shows how to do these things. It also shows how you can use the "connect-proxy" signal to monitor the changes in the UI. The "connect-proxy" signal is emitted by the UI manager every time a proxy is connected to an action. The callback for it must be of the form

```
void            user_function                    ( GtkUIManager *uimanager,
                                                   GtkAction    *action,
                                                   GtkWidget    *proxy,
                                                   gpointer      user_data);
```

The action is the action to which the proxy was connected, and the proxy is the menu or tool item that was connected. A callback that shows what the UI manager is doing "behind the scenes" follows:

```
void on_proxy_connect( GtkUIManager *ui,
                                GtkAction    *action,
                                GtkWidget    *proxy,
                                gpointer     *data)
{
    const gchar*  action_name = gtk_action_get_name(action);
    const gchar*  proxy_name  = gtk_widget_get_name(proxy);
    g_print ("%s connected to %s \n", action_name, proxy_name );
}
```

This shows that you can retrieve the action name and the proxy name using the `gtk_action_get_name()` method and the `gtk_widget_get_name()` method respectively. The function uses these strings to print a message on the console.

A Appendix

A.1 Listing: menu_by_hand.c

```
Listing: menu_by_hand.c
#include <stdio.h>
#include <gtk/gtk.h>

/*****************************************************************************
                                  Function Prototypes
*****************************************************************************/

static gboolean on_button_press     ( GtkWidget *widget,
                                      GdkEvent *event );
static void on_window_destroy       ( GtkWidget *widget,
                                      gpointer data);
static void on_menu_activate        ( GtkMenuItem* menu_item,
                                      GtkWindow * parent);


/*****************************************************************************
                                  Main Program
*****************************************************************************/

int main( int   argc,
          char *argv[] )
{
    GtkWidget *window;
    /*
       The textual layout of the declarations below model the hierarchical
       relationship of the menus and menu items that will be displayed by
       this application.
    */
    GtkWidget *menu_bar;
    GtkWidget *file_menu,
              *file_item,
                   *open_item,
                   *close_item,
                   *separator1,
                   *exit_item,
              *view_menu,
              *view_item,
              *tools_menu,
              *tools_item,
              *help_menu,
              *help_item,
                   *query_item,
                   *separator2,
                   *about_help_item,
                   *about_help_menu,
                       *about_tool_item,
                       *about_stuff_item
              ;
    GtkWidget *popupmenu,
                   *makebig_item,
                   *makesmall_item
              ;
    GtkWidget *vbox;
    GtkWidget *button;

    gtk_init (&argc, &argv);

    // create a new window
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_widget_set_size_request (GTK_WIDGET (window), 400, 300);
    gtk_window_set_title (GTK_WINDOW (window), "GTK Menu Demonstration 1");
    g_signal_connect (G_OBJECT (window), "delete_event",
                      G_CALLBACK (gtk_main_quit), NULL);


    // A vbox to put a menu and a button in:
    vbox = gtk_vbox_new (FALSE, 0);
    gtk_container_add (GTK_CONTAINER (window), vbox);
    gtk_widget_show (vbox);

    // Create a menu-bar to hold the menus and add it to our main window
    menu_bar = gtk_menu_bar_new ();


    // Create the menu items to put into the menubar first
    file_item = gtk_menu_item_new_with_label( "File" );
    view_item = gtk_menu_item_new_with_label( "View" );
    tools_item = gtk_menu_item_new_with_label( "Tools" );
    help_item = gtk_menu_item_new_with_label( "Help" );

    // Add the menu items to the menu_bar, left to right order
    gtk_menu_shell_append( GTK_MENU_SHELL( menu_bar ), file_item );
    gtk_menu_shell_append( GTK_MENU_SHELL( menu_bar ), view_item );
    gtk_menu_shell_append( GTK_MENU_SHELL( menu_bar ), tools_item );
    gtk_menu_shell_append( GTK_MENU_SHELL( menu_bar ), help_item );


    // Create the menus for the menu items in the menubar
    file_menu = gtk_menu_new();
    view_menu = gtk_menu_new();
    tools_menu = gtk_menu_new();
    help_menu = gtk_menu_new();

    // Attach each submenu to the menu items just added to the menu_bar
    gtk_menu_item_set_submenu( GTK_MENU_ITEM( file_item ), file_menu );
    gtk_menu_item_set_submenu( GTK_MENU_ITEM( view_item ), view_menu );
    gtk_menu_item_set_submenu( GTK_MENU_ITEM( tools_item ), tools_menu );
    gtk_menu_item_set_submenu( GTK_MENU_ITEM( help_item ), help_menu );

    // For each menu, create the items that will go into it and pack them
    // Create three items to put into the File menu
    open_item  = gtk_menu_item_new_with_label( "Open");
    close_item = gtk_menu_item_new_with_label( "Close");
    separator1 = gtk_separator_menu_item_new();
    exit_item  = gtk_menu_item_new_with_label( "Exit");

    // Append the items to the File menu
    gtk_menu_shell_append( GTK_MENU_SHELL(file_menu), open_item );
    gtk_menu_shell_append( GTK_MENU_SHELL(file_menu), close_item );
    gtk_menu_shell_append( GTK_MENU_SHELL(file_menu), separator1 );
    gtk_menu_shell_append( GTK_MENU_SHELL(file_menu), exit_item );
    // The View and Tools menu will be empty for now

    // Create the items for the Help menu
    query_item = gtk_menu_item_new_with_label( "What's This?");
    separator2 = gtk_separator_menu_item_new();
    about_help_item = gtk_menu_item_new_with_label( "About this program" );

    // and append them to the Help menu
    gtk_menu_shell_append( GTK_MENU_SHELL(help_menu), query_item );
    gtk_menu_shell_append( GTK_MENU_SHELL(help_menu), separator2 );
    gtk_menu_shell_append( GTK_MENU_SHELL(help_menu), about_help_item );

    // create the submenu and attach to the about_help_item
    about_help_menu = gtk_menu_new();
    gtk_menu_item_set_submenu ( GTK_MENU_ITEM( about_help_item ),
                                about_help_menu );

    // finally, create two menu items for this submenu and pack them into it
    about_tool_item = gtk_menu_item_new_with_label( "About This" );
    about_stuff_item = gtk_menu_item_new_with_label( "About That" );
    gtk_menu_shell_append( GTK_MENU_SHELL(about_help_menu), about_tool_item );
    gtk_menu_shell_append( GTK_MENU_SHELL(about_help_menu), about_stuff_item );


    // Connect the activate signal to each menu item.
    // This must be done using g_signal_connect_swapped
    g_signal_connect          (G_OBJECT (open_item), "activate",
                                       G_CALLBACK (on_menu_activate),
                               (GtkWindow*) window);

    g_signal_connect          (G_OBJECT (close_item), "activate",
                                       G_CALLBACK (on_menu_activate),
                               (GtkWindow*) window);

    g_signal_connect_swapped (G_OBJECT (exit_item), "activate",
                                       G_CALLBACK (on_window_destroy),
                               (gpointer) NULL);

    g_signal_connect          (G_OBJECT (query_item), "activate",
                                       G_CALLBACK (on_menu_activate),
                               (GtkWindow*) window);

    g_signal_connect          (G_OBJECT (about_tool_item), "activate",
                                       G_CALLBACK (on_menu_activate),
                               (GtkWindow*) window);

    g_signal_connect          (G_OBJECT (about_stuff_item), "activate",
                                       G_CALLBACK (on_menu_activate),
                               (GtkWindow*) window);

    gtk_widget_show (menu_bar);
    gtk_box_pack_start (GTK_BOX (vbox), menu_bar, FALSE, FALSE, 2);

    // Create pop-up menu for button
    popupmenu = gtk_menu_new();
    makebig_item = gtk_menu_item_new_with_label( "Larger");
    makesmall_item = gtk_menu_item_new_with_label( "Smaller");
    gtk_menu_shell_append (GTK_MENU_SHELL (popupmenu), makebig_item);
    gtk_menu_shell_append( GTK_MENU_SHELL (popupmenu), makesmall_item );
    g_signal_connect (G_OBJECT (makebig_item), "activate",
                                   G_CALLBACK (on_menu_activate),
                              (GtkWindow*) window);
    g_signal_connect (G_OBJECT (makesmall_item), "activate",
                                   G_CALLBACK (on_menu_activate),
                              (GtkWindow*) window);
    gtk_widget_show (makebig_item);
    gtk_widget_show (makesmall_item);

    /*
      Prior to Gtk+ 2.16 the only way to get the label from the menu item
      is to attach it as a property. It is a property in 2.16 and later
      When the menu item is clicked I want to display the label, so I have
      to do it this way.
    */

#if GTK_MINOR_VERSION < 16
    g_object_set_data(G_OBJECT(open_item), "label", "open item");
    g_object_set_data(G_OBJECT(close_item), "label", "close item");
    g_object_set_data(G_OBJECT(about_tool_item), "label", "about_tool item");
    g_object_set_data(G_OBJECT(about_stuff_item), "label", "about_stuff item");
    g_object_set_data(G_OBJECT(query_item), "label", "query_item");
    g_object_set_data(G_OBJECT(makebig_item), "label", "large_item");
    g_object_set_data(G_OBJECT(makesmall_item), "label", "small_item");
#endif

    button = gtk_button_new_with_label ( "Push me");
    g_signal_connect_swapped (G_OBJECT (button), "event",
                                   G_CALLBACK (on_button_press),
                              G_OBJECT (popupmenu));
    gtk_box_pack_start (GTK_BOX (vbox), button, FALSE, FALSE, 2);
    gtk_widget_show (button);


    gtk_widget_show_all( window );

    gtk_main ();

    return 0;
}


/*****************************************************************************
                              CALLBACK HANDLERS
*****************************************************************************/


/*****************************************************************************
                              on_window_destroy
*****************************************************************************/


static void on_window_destroy       ( GtkWidget *widget,
                                      gpointer data)
{
    gtk_main_quit ();
}
```

```
/******************************************************************************
                              on_menu_activate
******************************************************************************/


static void on_menu_activate        ( GtkMenuItem * menu_item,
                                      GtkWindow * parent)
{

// Read the comment in the main program
#if GTK_MINOR_VERSION < 16
    const gchar*  label = g_object_get_data(G_OBJECT(menu_item), "label");
#else
    const gchar*  label = gtk_menu_item_get_label(menu_item);
#endif

    GtkWidget *dialog;
    dialog = gtk_message_dialog_new (parent, GTK_DIALOG_MODAL,
                                      GTK_MESSAGE_INFO, GTK_BUTTONS_OK,
                                    "%s activated", label);
    gtk_window_set_title (GTK_WINDOW (dialog), "Menu Selection");

    gtk_dialog_run (GTK_DIALOG (dialog));
    gtk_widget_destroy (dialog);

    /* Tell calling code that we have handled this event; the buck
     * stops here. */
    return ;

}

/******************************************************************************
                              on_button_press
******************************************************************************/

static gboolean on_button_press     ( GtkWidget *widget,
                                      GdkEvent *event )
{
    if (event->type == GDK_BUTTON_PRESS) {
        GdkEventButton *bevent = (GdkEventButton *) event;
        gtk_menu_popup (GTK_MENU (widget), NULL, NULL, NULL, NULL,
                        bevent->button, bevent->time);
        /* Tell calling code that we have handled this event; the buck
         * stops here. */
        return TRUE;
    }

    /* Tell calling code that we have not handled this event; pass it on. */
    return FALSE;
}
```

A.2 Listing: toolbar_by_hand.c

```
#include <gtk/gtk.h>
#include <libgen.h>

#define WINWIDTH      400
#define WINHEIGHT     400
```

```
int main( int argc, char *argv[])
{
    GtkWidget    *window;
    GtkWidget    *vbox;

    GtkWidget    *toolbar;
    GtkToolItem *new_button;
    GtkToolItem *open_button;
    GtkToolItem *save_button;
    GtkToolItem *separator;
    GtkToolItem *exit_button;

    gtk_init(&argc, &argv);

    window = gtk_window_new           (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title              (GTK_WINDOW (window), basename(argv[0]) );
    gtk_widget_set_size_request       (GTK_WIDGET (window),
                                       WINWIDTH, WINHEIGHT);
    gtk_window_set_position           (GTK_WINDOW(window), GTK_WIN_POS_CENTER);

    g_signal_connect (window, "destroy",
                      G_CALLBACK (gtk_main_quit),
                      NULL);

    vbox = gtk_vbox_new(FALSE, 0);
    gtk_container_add(GTK_CONTAINER(window), vbox);


    toolbar = gtk_toolbar_new();

    gtk_container_set_border_width(GTK_CONTAINER(toolbar), 2);

    new_button = gtk_tool_button_new_from_stock(GTK_STOCK_NEW);
    open_button = gtk_tool_button_new_from_stock(GTK_STOCK_OPEN);
    save_button = gtk_tool_button_new_from_stock(GTK_STOCK_SAVE);

    separator = gtk_separator_tool_item_new();
    gtk_tool_item_set_expand(GTK_TOOL_ITEM(separator), TRUE);
    gtk_separator_tool_item_set_draw(GTK_SEPARATOR_TOOL_ITEM(separator), FALSE);

    exit_button = gtk_tool_button_new_from_stock(GTK_STOCK_QUIT);


    gtk_toolbar_insert(GTK_TOOLBAR(toolbar), new_button, -1);
    gtk_toolbar_insert(GTK_TOOLBAR(toolbar), open_button, -1);
    gtk_toolbar_insert(GTK_TOOLBAR(toolbar), save_button, -1);
    gtk_toolbar_insert(GTK_TOOLBAR(toolbar), separator, -1);
    gtk_toolbar_insert(GTK_TOOLBAR(toolbar), exit_button, -1);

    gtk_box_pack_start(GTK_BOX(vbox), toolbar, FALSE, FALSE, 5);

    g_signal_connect(G_OBJECT(exit_button), "clicked",
          G_CALLBACK(gtk_main_quit), NULL);
    gtk_widget_show_all(window);
    gtk_main();

    return 0;
}
```