GTK+ 2.0 Tutorial using Ocaml

Tony Gale

Ian Main

& the GTK team

Ocaml Adaptation: SooHyoung Oh

GTK+ 2.0 Tutorial using Ocaml

by Tony Gale, Ian Main, & the GTK team, and Ocaml Adaptation: SooHyoung Oh

This is a tutorial on how to use GTK (the GIMP Toolkit) through its Ocaml interface.

Table of Contents

Chapter 1. Tutorial Availability

A copy of this tutorial in SGML and HTML is distributed with each source code release of GTK+. For binary distributions, please check with you vendor.

A copy is available online for reference at www.gtk.org/tutorial.

A packaged verion of this tutorial is available from ftp.gtk.org/pub/gtk/tutorial which contains the tutorial in various different formats. This package is primary for those people wanting to have the tutorial available for offline reference and for printing.

1.1. Ocaml Version Tutorial Availability

This tutorial is available from http://compiler.kaist.ac.kr/~shoh/ocaml/lablgtk2/lablgtk2_tutorial.tar.gz

And the source code used in this tutorial is available at http://compiler.kaist.ac.kr/~shoh/ocaml/lablgtk2/lablgtk2_tutorial_

Chapter 2. Introduction

GTK (GIMP Toolkit) is a library for creating graphical user interfaces. It is licensed using the LGPL license, so you can develop open software, free software, or even commercial non-free software using GTK without having to spend anything for licenses or royalties.

It's called the GIMP toolkit because it was originally written for developing the GNU Image Manipulation Program (GIMP), but GTK has now been used in a large number of software projects, including the GNU Network Object Model Environment (GNOME) project. GTK is built on top of GDK (GIMP Drawing Kit) which is basically a wrapper around the low-level functions for accessing the underlying windowing functions (Xlib in the case of the X windows system), and gdk-pixbuf, a library for client-side image manipulation.

The primary authors of GTK are:

- Peter Mattis petm@xcf.berkeley.edu
- Spencer Kimball spencer@xcf.berkeley.edu
- Josh MacDonald jmacd@xcf.berkeley.edu

GTK is currently maintained by:

- Owen Taylor otaylor@redhat.com
- Tim Janik timj@gtk.org

GTK is essentially an object oriented application programmers interface (API). Although written completely in C, it is implemented using the idea of classes and callback functions (pointers to functions).

There is also a third component called GLib which contains a few replacements for some standard calls, as well as some additional functions for handling linked lists, etc. The replacement functions are used to increase GTK's portability, as some of the functions implemented here are not available or are nonstandard on other Unixes such as g_strerror(). Some also contain enhancements to the libc versions, such as g_malloc() that has enhanced debugging utilities.

In version 2.0, GLib has picked up the type system which forms the foundation for GTK's class hierarchy, the signal system which is used throughout GTK, a thread API which abstracts the different native thread APIs of the various platforms and a facility for loading modules.

As the last component, GTK uses the Pango library for internationalized text output.

The original tutorial describes the C interface to GTK. There are GTK bindings for many other languages including Ocaml, C++, Guile, Perl, Python, TOM, Ada95, Objective C, Free Pascal, Eiffel, Java and C#. If you intend to use another language's bindings to GTK, look at that binding's documentation first. In some cases that documentation may describe some important conventions (which you should know first) and then refer you back to the original tutorial. There are also some cross-platform APIs (such as wxWindows and V) which use GTK as one of their target platforms; again, consult their documentation first.

If you're developing your GTK application in C++, a few extra notes are in order. There's a C++ binding to GTK called GTK--, which provides a more C++-like interface to GTK; you should probably look into this instead. If you don't like that approach for whatever reason, there are two alternatives for using GTK. First, you can use only the C subset of C++ when interfacing with GTK and then use the C interface as described in the original tutorial. Second, you can use GTK and C++ together by declaring all callbacks as static functions in C++ classes, and again calling GTK using its C interface. If you choose this last approach, you can include as the callback's data value a pointer to the object to be manipulated (the so-called "this" value). Selecting between these options is simply a matter of preference, since in all three approaches you get C++ and GTK. None of these approaches requires the use of a specialized preprocessor, so no matter what you choose you can use standard C++ with GTK.

The original tutorial is an attempt to document as much as possible of GTK, but it is by no means complete. The tutorial assumes a good understanding of C, and how to create C programs. It would be a great benefit for the reader to have previous X programming experience, but it shouldn't be necessary. If you are learning GTK as your first widget set, please comment on how you found this tutorial, and what you had trouble with. There are also Ocaml, C++, Objective C, ADA, Guile and other language bindings available, but I don't follow these.

The original document is a "work in progress". Please look for updates on http://www.gtk.org/.

I would very much like to hear of any problems you have learning GTK from the original document, and would appreciate input as to how it may be improved. Please see the section on Contributing for further information.

2.1. GTK+ 2.0 in Ocaml

`Ocaml` is a fantastic programming language which has the various modern features. Some of them are:

- Functional / Imperative features: It is a functional language, since the basic units of programs are functions. And it provides full imperative capabilities, including updatable arrays, imperative variables and records with mutable fields.
- Strongly-typed: It is a strongly-typed language; it means that the objects that you use belong to a set that has a name, called its type. In Caml, types are managed by the computer, the user has nothing to do about types (types are synthesized). And caml features “polymorphic typing”: some types may be left unspecified, standing for “any type”.
- Compiler / Interpreter: Caml provides an interpreter as well as compiler. An interactive top-level executes “read-eval-print” loop, that is convenient for both learning, or rapid testing and debugging of programs.
- Object oriented: Objective Caml features objects that give a fully fledged object oriented programming style in Caml programs.

`LablGTK` is an Objective Caml interface to gtk+. It comes in two flavors: LablGTK1 for gtk+-1.2 and LablGTK2 for gtk+-2.0 to gtk+-2.4.

It is still under development, but already fully functional. All widgets (but one) are available, with almost all their methods. The GLArea widget is also supported in combination with LablGL. LibGlade and GdkPixbuf support is also included for both versions. LablGTK2 adds support for gnomecanvas, librsvg and libpanel Many examples are provided.

You can use LablGTK in win32 systems as well as in the unix like system.

Note: Some chapters of the original document have been omitted in ocaml version:

- GLib
- Tips For Writing GTK Applications

The chapters or sections with "(?)" at the end of the name is not translated part to ocaml language. It is waiting contributors.

The following chapters have only titles while the corresponding one in the C version tutorial has large contents:

- Writing Your Own Widgets
- GTK Signals

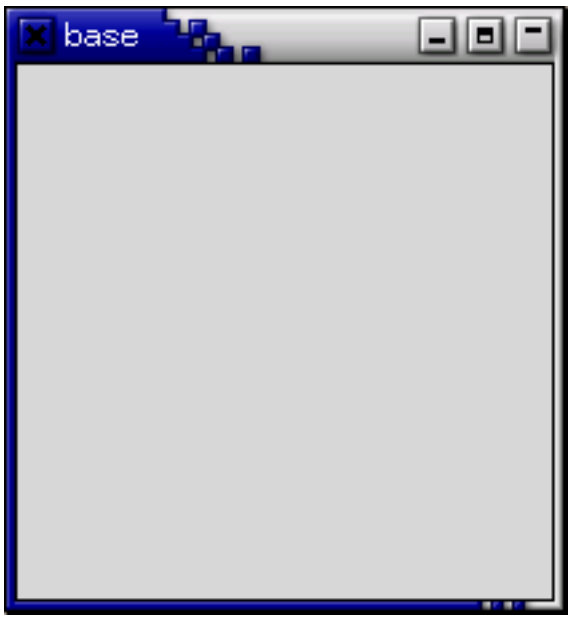
So, you have to read the `original` document to refer these chapters.

Note: There are good example sources in lablgtk distribution and I borrowed some code from that.

Note: Some part of this document came from LablGTK README files.

Chapter 3. Getting Started

To begin our introduction to GTK, we'll start with the simplest program possible. This program will create a 200x200 pixel window and has no way of exiting except to be killed by using the shell.




```
(* file: base.ml *)

let main () =
  let window = GWindow.window () in
  window#show ();
  GMain.Main.main ()

let _ = main ()
```

You can compile the above program with ocamlc using:

```
ocamlc -I +lablgtk2 -o base lablgtk.cma gtkInit.cmo base.ml
```

The meaning of the unusual compilation options is explained below in Compiling Hello World.

The first two lines of code create and display a window.

```
  let window = GWindow.window () in
  window#show ();
```

Rather than create a window of 0x0 size, a window without children is set to 200x200 by default so you can still manipulate it.

The `GWindow.window#show` method lets GTK know that we are done setting the attributes of this widget, and that it can display it.

The last line enters the GTK main processing loop.

```
  GMain.Main.main ()
```

GMain.Main.main `GMain.Main.main` is another call you will see in every GTK application. When control reaches this point, GTK will sleep waiting for events (such as button or key presses), timeouts, or file IO notifications to occur. In our simple example, however, events are ignored.

3.1. Hello World in GTK

Now for a program with a widget (a button). It's the classic hello world a la GTK.


```
(* file: hello.ml *)

(* This is a callback function. *)
let hello () =
  print_endline "Hello World";
  flush stdout

(* Another callback function.
 * If you return [false] in the "delete_event" signal handler,
 * GTK will emit the "destroy" signal. Returning [true] means
 * you don't want the window to be destroyed.
 * This is useful for popping up 'are you sure you want to quit?'
 * type dialogs. *)
let delete_event ev =
  print_endline "Delete event occurred";
  flush stdout;

  (* Change [true] to [false] and the main window will be destroyed with
   * a "delete event" *)
  true

let destroy () = GMain.Main.quit ()

let main () =
  (* Create a new window and sets the border width of the window. *)
  let window = GWindow.window ~border_width:10 () in

  (* When the window is given the "delete_event" signal (this is given
   * by the window manager, usually by the "close" option, or on the
   * titlebar), we ask it to call the delete_event () function
   * as defined above. *)
  window#event#connect#delete ~callback:delete_event;

  (* Here we connect the "destroy" event to a signal handler.
   * This event occurs when we call window#destroy method
   * or if we return [false] in the "delete_event" callback. *)
  window#connect#destroy ~callback:destroy;

  (* Creates a new button with the label "Hello World".
   * and packs the button into the window (a gtk container). *)
  let button = GButton.button ~label:"Hello World" ~packing:window#add () in

  (* When the button receives the "clicked" signal, it will call the
   * function hello().  The hello() function is defined above. *)
  button#connect#clicked ~callback:hello;

  (* This will cause the window to be destroyed by calling
   * window#destroy () when "clicked".  Again, the destroy
   * signal could come from here, or the window manager. *)
  button#connect#clicked ~callback:window#destroy;

  (* The final step is to display the window. *)
  window#show ();

  (* All GTK applications must have a GMain.Main.main (). Control ends here
   * and waits for an event to occur (like a key press or
   * mouse event). *)
  GMain.Main.main ()

let _ = main ()
```

3.2. Compiling Hello World

To compile use:

```
ocamlc -I +lablgtk2 -o helloworld lablgtk.cma gtkInit.cmo helloworld.ml
```

The options are:

- `-I +lablgtk2`: adds the subdirectory `lablgtk2` of the standard library to the search path. In that directory, there are compiled interface files (.cmi), compiled object code files (.cmo), libraries (.cma) related with lablgtk2.
- `-o hello`: specify the name of the output file produced by the linker.

The library and object that are usually linked in are:

- The LablGtk library (`lablgtk.cma`), the GTK+ widget library.
- gtkInit object(`gtkInit.cmo`), containing gtkInit function. If you use this object code, you don't have to call `GtkMain.Main.init ()` function before any lablgtk functions.

There are many other options and libraries very useful, please refer the ocaml manual.

3.3. Theory of Signals and Callbacks

Before we look in detail at `helloworld`, we'll discuss signals and callbacks. GTK is an event driven toolkit, which means it will sleep in `GMain.Main.main` until an event occurs and control is passed to the appropriate function.

This passing of control is done using the idea of "signals". (Note that these signals are not the same as the Unix system signals, and are not implemented using them, although the terminology is almost identical.) When an event occurs, such as the press of a mouse button, the appropriate signal will be "emitted" by the widget that was pressed. This is how GTK does most of its useful work. There are signals that all widgets inherit, such as "destroy", and there are signals that are widget specific, such as "toggled" on a toggle button.

To make a button perform an action, we set up a signal handler to catch these signals and call the appropriate function. This is done by using a function such as:

```
widget#connect#signal_name ~callback:(unit -> unit) -> GtkSignal.id
```

where the `widget` is the one which will be emitting the signal, and the `signal_name` is the name of the signal you wish to catch. The `callback` is the function you wish to be called when it is caught.

3.4. Events

In addition to the signal mechanism described above, there is a set of `events` that reflect the Window event mechanism. Callbacks may also be attached to these events. These events are: (see `event_signals`)

- any
- button_press
- button_release
- scroll
- motion_notify
- delete
- destroy
- expose
- key_press
- key_release
- enter_notify
- leave_notify
- configure
- focus_in
- focus_out
- map
- unmap

- property_notify
- selection_clear
- selection_request
- selection_notify
- proximity_in
- proximity_out
- visibility_notify
- client
- no_expose
- window_state

In order to connect a callback function to one of these events you use the `#event#connect` method, using one of the above event names as the `event_signal_name` like this:

```
widget#event#connect#event_signal_name ~callback:(event -> bool) -> GtkSignal.id
```

The callback function for events has a slightly different form than that for signals as you can see.

The argument of the callback function `event` may have various type which will depend upon which of the above events has occurred. The components of the event structure will depend upon the type of the event. The possible types of event are: (see `event_signals`)

```
Gtk.Tags.event_type Gdk.event
[‘DELETE] Gdk.event
[‘DESTROY] Gdk.event
[‘MAP] Gdk.event
[‘UNMAP] Gdk.event
GdkEvent.Button.t
GdkEvent.Crossing.t
GdkEvent.Configure.t
GdkEvent.Expose.t
GdkEvent.Focus.t
GdkEvent.Key.t
GdkEvent.Motion.t
GdkEvent.Proximity.t
GdkEvent.Selection.t
...
```

So, to connect a callback function to one of these events we would use something like:

```
button#event#connect#button_press ~callback:button_pressed;
```

This assumes that `button` is a Button widget. Now, when the mouse is over the button and a mouse button is pressed, the function `button_pressed` will be called. This function may be declared as:

```
button_pressed (ev:GdkEvent.Button.t) =
 ...
 true (* or false *)
```

Note that the argument has a type `GdkEvent.Button.t` as we know what type of event will occur for this function to be called.

The value returned from this function indicates whether the event should be propagated further by the GTK event handling mechanism. Returning `true` indicates that the event has been handled, and that it should not propagate further. Returning `false` continues the normal event handling. See the section on Advanced Event and Signal Handling for more details on this propagation process.

For details on the GdkEvent data types, see the appendix entitled GDK Event Types.

The GDK selection and drag-and-drop APIs also emit a number of events which are reflected in GTK by the signals. See Signals on the source widget and Signals on the destination widget for details on the signatures of the callback functions for these signals:

- #drag#connect#beginning
- #drag#connect#ending
- #drag#connect#data_delete
- #drag#connect#motion
- #drag#connect#drop
- #drag#connect#data_get

- #drag#connect#data_received

3.5. Stepping Through Hello World

Now that we know the theory behind this, let's clarify by walking through the example *helloworld* program.

Here is the callback function that will be called when the button is "clicked". We ignore both the widget and the data in this example, but it is not hard to do things with them. The next example will use the data argument to tell us which button was pressed.

```
let hello () =
  print_endline "Hello World";
  flush stdout
```

The next callback is a bit special. The "delete_event" occurs when the window manager sends this event to the application. We have a choice here as to what to do about these events. We can ignore them, make some sort of response, or simply quit the application.

The value you return in this callback lets GTK know what action to take. By returning `true`, we let it know that *we don't want* to have the "destroy" signal emitted, keeping our application running. By returning `false`, we ask that "destroy" be emitted, which in turn will call our "destroy" signal handler.

```
let delete_event ev =
  print_endline "Delete event occurred";
  flush stdout;
  true
```

Here is another callback function which causes the program to quit by calling `GMain.Main.quit ()`. This function tells GTK that it is to exit from `GMain.Main.main` when control is returned to it.

```
let destroy () = GMain.Main.quit ()
```

In "C" language, you *have to* use "main" for starting function name, but in "Ocaml", you don't have to. You may use any name for a fuction, and we'll tell the system the name of the starting point. In this example, we'll use "main" for the name of ther starting function.

```
let main () =
```

Create a new window using `GWindow.window`. This is fairly straightforward. Memory is allocated for the GtkWidget *window structure so it now points to a valid structure. It sets up a new window, but it is not displayed until we call `window#show ()` near the end of our program.

```
  let window = GWindow.window ~border_width:10 () in
```

This `~border_width` option is used to set an attribute of a container object. This just sets the window so it has a blank area along the inside of it 10 pixels wide where no widgets will go. There are other similar functions which we will look at in the section on Setting Widget Attributes

Here are two examples of connecting a signal handler to an object, in this case, the window. Here, the "delete_event" and "destroy" signals are caught. The first is emitted when we use the window manager to kill the window, or when we use the `window#destroy ()` call passing in the window widget as the object to destroy. The second is emitted when, in the "delete_event" handler, we return `false`. The `G_OBJECT` and `G_CALLBACK` are macros that perform type casting and checking for us, as well as aid the readability of the code.

```
  window#event#connect#delete ~callback:delete_event;
  window#connect#destroy ~callback:destroy;
```

This call creates a new button. It allocates space for a new GtkWidget structure in memory, initializes it, and makes the button pointer point to it. It will have the label "Hello World" on it when displayed.

```
  let button = GButton.button ~label:"Hello World" ~packing:window#add () in
```

There is a `~packing` option, which will be explained in depth later on in Packing Widgets. But it is fairly easy to understand. It simply tells GTK that the button is to be placed in the window where it will be displayed. Note that a GTK container can only contain one widget. There are other widgets, that are described later, which are designed to layout multiple widgets in various ways.

Here, we take this button, and make it do something useful. We attach a signal handler to it so when it emits the "clicked" signal, our hello() function is called. Obviously, the "clicked" signal is emitted when we click the button with our mouse pointer.

```
button#connect#clicked ~callback:hello;
```

We are also going to use this button to exit our program. This will illustrate how the "destroy" signal may come from either the window manager, or our program. When the button is "clicked", same as above, it calls the first hello() callback function, and then this one in the order they are set up. You may have as many callback functions as you need, and all will be executed in the order you connected them.

```
button#connect#clicked ~callback:window#destroy;
```

Now we have everything set up the way we want it to be. With all the signal handlers in place, and the button placed in the window where it should be, we ask GTK to "show" the widgets on the screen. All widgets are "shown" by default except window widget. The window widget is shown last so the whole window will pop up at once rather than seeing the window pop up, and then the button form inside of it. Although with such a simple example, you'd never notice.

```
window#show ();
GMain.Main.main ()
```

And of course, we call `GMain.Main.main ()` which waits for events to come from the X server/Window and will call on the widgets to emit signals when these events come.

```
GMain.Main.main ()
```

And for the final thing, we should tell the system how to do this application. On executing the following line, the function named "main" will be called.

```
let _ = main ()
```

or, if you want to see error message which is raised on evaluating `main ()` functon:

```
let _ = Printexc.print main ()
```

Now, when we click the mouse button on a GTK button, the widget emits a "clicked" signal. In order for us to use this information, our program sets up a signal handler to catch that signal, which dispatches the function of our choice. In our example, when the button we created is "clicked", the hello() function is called , and then the next handler for this signal is called. This calls the `window#destroy ()` function, destroying the window widget. This causes the window to emit the "destroy" signal, which is caught, and calls our destroy() callback function, which simply exits GTK.

Another course of events is to use the window manager to kill the window, which will cause the "delete_event" to be emitted. This will call our "delete_event" handler. If we return `true` here, the window will be left as is and nothing will happen. Returning `false` will cause GTK to emit the "destroy" signal which of course calls the "destroy" callback, exiting GTK.

Chapter 4. Moving On

4.1. More on Signal Handlers

Lets take another look at the #connect#*signal_name* declaration.

```
#connect#signal_name ~callback:(unit -> unit) -> GtkSignal.id
```

Notice the GtkSignal.id return value? This is a tag that identifies your callback function. As stated above, you may have as many callbacks per signal and per object as you need, and each will be executed in turn, in the order they were attached.

This tag allows you to remove this callback from the list by using `#misc#disconnect` method:

```
#misc#disconnect: GtkSignal.id -> unit
```

So, by passing the tag returned by one of the connect functions to the `#misc#disconnect` method, you can disconnect a signal handler.

You can also temporarily disable signal handlers with the `#misc#handler_block` and `#misc#handler_unblock` family of functions.

```
#misc#handler_block : GtkSignal.id -> unit
#misc#handler_unblock : GtkSignal.id -> unit
```

4.2. An Upgraded Hello World

Let's take a look at a slightly improved *helloworld* with better examples of callbacks. This will also introduce us to our next topic, packing widgets.


```
(* file: hello2.ml *)

let clicked msg () =
  print_endline msg;
  flush stdout

let delete_event ev =
 GMain.Main.quit ();
 false

let main () =
  (* Create a new window and sets the border width and title of the window. *)
  let window = GWindow.window ~title:"Hello Buttons!" ~border_width:10 () in

  (* Here we just set a handler for delete_event that immediately
   * exits GTK. *)
  window#event#connect#delete ~callback:delete_event;

  (* We create a box to pack widgets into.  This is described in detail
   * in the "packing" section. The box is not really visible, it
   * is just used as a tool to arrange widgets.
   * And put the box into the main window. *)
  let box1 = GPack.hbox ~packing:window#add () in

  (* Creates a new button with the label "Button 1".
   * Instead of box1#add, we pack this button into the invisible
   * box, which has been packed into the window. *)
  let button = GButton.button ~label:"Button 1" ~packing:box1#pack () in
```

```
  (* Now when the button is clicked, we call the "clicked" function
   * with "button 1" as its argument *)
  button#connect#clicked ~callback:(clicked "button 1");

  (* Do these same steps again to create a second button *)
  let button = GButton.button ~label:"Button 2" ~packing:box1#pack () in

  (* Call the same callback function with a different argument,
   * passing "button 2" instead. *)
  button#connect#clicked ~callback:(clicked "button 2");

  (* Display the window. *)
  window#show ();

  (* Rest in GMain.Main.main and wait for the fun to begin! *)
  GMain.Main.main ()

let _ = main ()
```

Compile this program using the same linking arguments as our first example. You'll notice this time there is no easy way to exit the program, you have to use your window manager or command line to kill it. A good exercise for the reader would be to insert a third "Quit" button that will exit the program. You may also wish to play with the options to `box#pack ()` while reading the next section. Try resizing the window, and observe the behavior.

Chapter 5. Packing Widgets

When creating an application, you'll want to put more than one widget inside a window. Our first *helloworld* example only used one widget so we could simply use a #add (or #pack) method call to "pack" the widget into the window. But when you want to put more than one widget into a window, how do you control where that widget is positioned? This is where packing comes in.

5.1. Theory of Packing Boxes

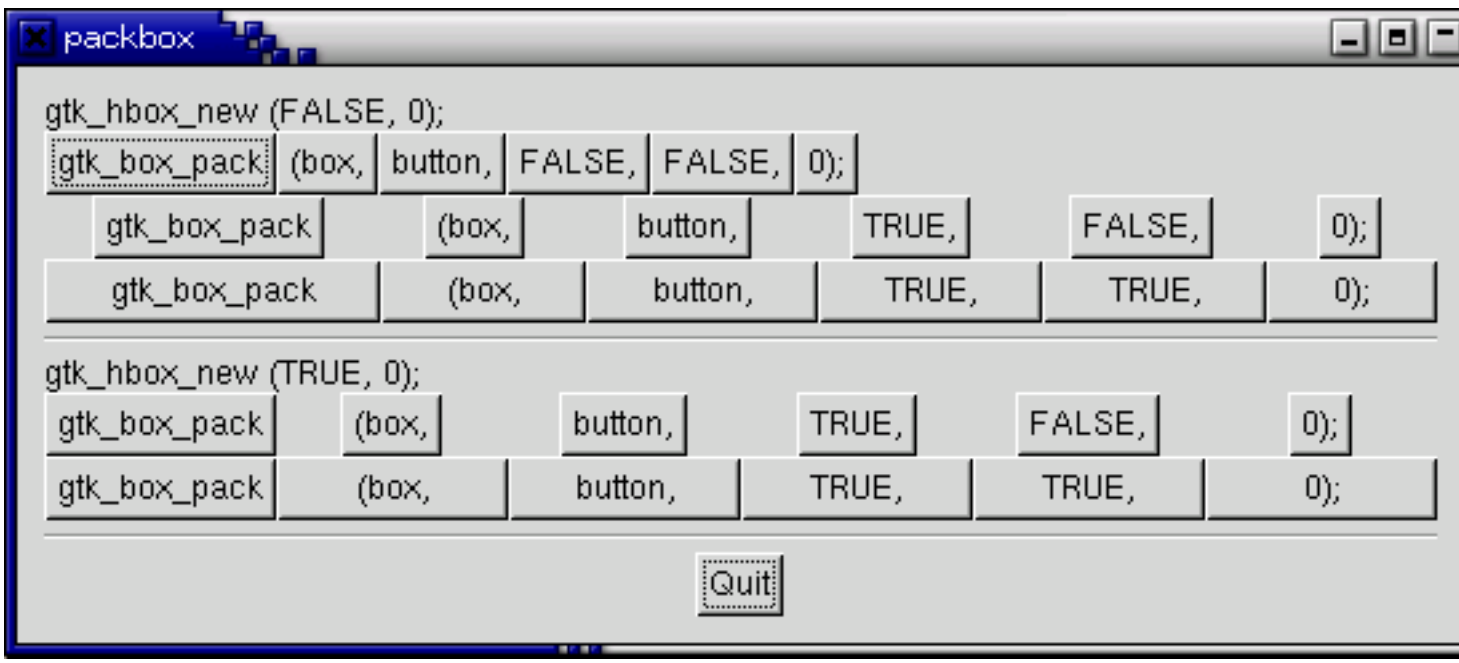
Most packing is done by creating boxes. These are invisible widget containers that we can pack our widgets into which come in two forms, a horizontal box, and a vertical box. When packing widgets into a horizontal box, the objects are inserted horizontally from left to right or right to left depending on the call used. In a vertical box, widgets are packed from top to bottom or vice versa. You may use any combination of boxes inside or beside other boxes to create the desired effect.

To create a new horizontal box, we use a call to `GPack.hbox ()`, and for vertical boxes, `GPack.vbox ()`. The `box#pack` or `box#add` method is used to place objects inside of these containers. An object may be another container or a widget. In fact, many widgets are actually containers themselves, including the button, but we usually only use a label inside a button.

By using this method, GTK knows where you want to place your widgets so it can do automatic resizing and other nifty things. There are also a number of options as to how your widgets should be packed. As you can imagine, this method gives us a quite a bit of flexibility when placing and creating widgets.

5.2. Details of Boxes

Because of this flexibility, packing boxes in GTK can be confusing at first. There are a lot of options, and it's not immediately obvious how they all fit together. In the end, however, there are basically five different styles.



Each line contains one horizontal box (hbox) with several buttons. The call to `#pack` method is shorthand for the call to pack each of the buttons into the hbox. Each of the buttons is packed into the hbox the same way.

This is the declaration of the `#pack` method.

```
method pack :
 ?from:Gtk.Tags.pack_type ->
 ?expand:bool ->
 ?fill:bool ->
 ?padding:int ->
 GObj.widget -> unit

from : default value is 'START
```

```
expand : default vaue is false
fill : default value is true, ignored if expand is false
```

The first optional argument `from` can have `'START` or `'END`. The `~from:'START` will start at the top and work its way down in a vbox, and pack left to right in an hbox. The `~from:'END` will do the opposite, packing from bottom to top in a vbox, and right to left in an hbox. Using `from` option allows us to right justify or left justify our widgets and may be mixed in any way to achieve the desired effect.

The `expand` argument controls whether the widgets are laid out in the box to fill in all the extra space in the box so the box is expanded to fill the area allotted to it (`true`); or the box is shrunk to just fit the widgets (`false`). Setting expand to `false` will allow you to do right and left justification of your widgets. Otherwise, they will all expand to fit into the box.

The `fill` argument to the `#pack` method control whether the extra space is allocated to the objects themselves (`true`), or as extra padding in the box around these objects (`false`). It only has an effect if the `expand` argument is also `true`.

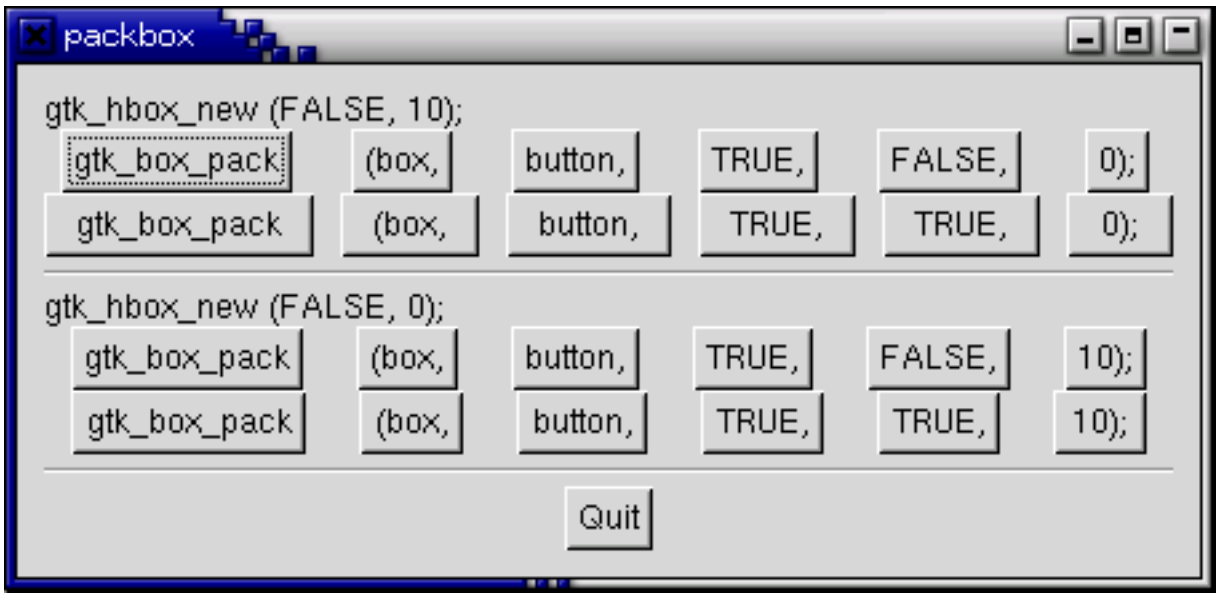
The last argument is the object you are packing into the box. The objects will all be buttons for now, so we'll be packing buttons into boxes.

When creating a new box, the function looks like this (`GPack.hbox ()`):

```
GPack.hbox :
 ?homogeneous:bool ->
 ?spacing:int ->
 ?border_width:int ->
 ?width:int ->
 ?height:int ->
 ?packing:(GObj.widget -> unit) ->
 ?show:bool ->
 unit -> box

spacing: 0 by default
homogeneous: false by default
```

The `homogeneous` argument to `GPack.hbox ()` (and the same for `GPack.vbox ()`) controls whether each object in the box has the same size (i.e., the same width in an hbox, or the same height in a vbox). If it is set, the #pack method essentially as if the `expand` argument was always turned on.

What's the difference between `spacing` (set when the box is created) and `padding` (set when elements are packed)? `Spacing` is added between objects, and `padding` is added on either side of an object. The following figure should make it clearer:



Here is the code used to create the above images. I've commented it fairly heavily so I hope you won't have any problems following it. Compile it yourself and play with it.

5.3. Packing Demonstration Program

```
(* file: packbox.ml *)

(* Make a new hbox filled with button-labels. Arguments of the
 * variables we're interested are passed in to this fuction.
 * We do not show the box, butdo show everything inside. *)
let make_box ~homogeneous ~spacing ~expand ~fill ~padding ?packing () =
  (* Create a new hbox with the appropriate homogeneous
   * and spacing settings *)
  let box = GPack.box 'HORIZONTAL ~homogeneous ~spacing ?packing () in

  (* Create a series of buttons with the appropriate settings *)
  let button = GButton.button ~label:"box_pack"
    ~packing:(box#pack ~expand ~fill ~padding) () in

  let button = GButton.button ~label:"(box"
    ~packing:(box#pack ~expand ~fill ~padding) () in

  let button = GButton.button ~label:"button,"
    ~packing:(box#pack ~expand ~fill ~padding) () in

  (* Create a button with the label depending on the value of
   * expand. *)
  let button = GButton.button ~label:(if expand then "TRUE," else "FALSE,")
    ~packing:(box#pack ~expand ~fill ~padding) () in

  (* This is the same as the button creation for "expand". *)
  let button = GButton.button ~label:(if fill then "TRUE," else "FALSE,")
    ~packing:(box#pack ~expand ~fill ~padding) () in

  let button = GButton.button ~label:(Printf.sprintf "%d)" padding)
    ~packing:(box#pack ~expand ~fill ~padding) () in

  box

let main () =
  if Array.length Sys.argv < 2 then (
    prerr_endline "usage: packbox num, where num is 1, 2, or 3.";
    exit 1
  );
  let which = int_of_string Sys.argv.(1) in

  (* Create our window *)
  let window = GWindow.window ~title:"Packing" ~border_width:10 () in

  (* You should always rememeber to connect the destroy signal
   * to the main window. This is very important for proper intutive
   * behavior *)
  window #connect#destroy ~callback:GMain.Main.quit;

  let box1 = GPack.vbox ~packing:window#add () in

  (* which example to show. These correspond to the pictures above. *)
  begin
  match which with
  | 1 ->
      (* Create a new label. *)
      let label = GMisc.label ~text:"hbox_new (false, 0);" () in

      (* Align the label to the left side. We'll discuss this function and
       * others in the section on Widget Attributes. *)
      label#set_xalign 0.0;
      label#set_yalign 0.0;

      (* Pack the label into the vertical box (vobx box1). Remember that
       * widget added to a vbox will be packed one on top of the otehr in
       * order. *)
      box1#pack ~expand:false ~fill:false ~padding:0 label#coerce;

      (* Call our make box function *)
```

```
      let box2 = make_box ~homogeneous:false ~spacing:0
        ~expand:false ~fill:false ~padding:0 () in
      box1#pack ~expand:false ~fill:false ~padding:0 box2#coerce;

      (* Call our make box function *)
      let box2 = make_box ~homogeneous:false ~spacing:0
        ~expand:true ~fill:false ~padding:0 () in
      box1#pack ~expand:false ~fill:false ~padding:0 box2#coerce;

      (* Call our make box function *)
      let box2 = make_box ~homogeneous:false ~spacing:0
        ~expand:true ~fill:true ~padding:0 () in
      box1#pack ~expand:false ~fill:false ~padding:0 box2#coerce;

      (* Creates a separator, we'll learn more about these later,
       * but they are quite simple. *)
      let separator = GMisc.separator `HORIZONTAL () in
      box1#pack ~expand:false ~fill:true ~padding:0 separator#coerce;

      (* Create another new label, and show it. *)
      let label = GMisc.label ~text:"hbox_new (true, 0);" () in
      label#set_xalign 0.0;
      label#set_yalign 0.0;
      box1#pack ~expand:false ~fill:false ~padding:0 label#coerce;

      let box2 = make_box ~homogeneous:true ~spacing:0
        ~expand:true ~fill:false ~padding:0 () in
      box1#pack ~expand:false ~fill:false ~padding:0 box2#coerce;

      let box2 = make_box ~homogeneous:true ~spacing:0
        ~expand:true ~fill:true ~padding:0 () in
      box1#pack ~expand:false ~fill:false ~padding:0 box2#coerce;

      (* Another new separator. *)
      let separator = GMisc.separator `HORIZONTAL () in
      box1#pack ~expand:false ~fill:true ~padding:5 separator#coerce;

  | 2 ->
      (* Create a new label, remember box1 is a vbox as created
       * near the beginning of main().
       * You can give widget attributes on widget creation.
       * See: ~xalign, ~yalign, ~packing *)
      let label = GMisc.label ~text:"hbox_new (false, 10);"
        ~xalign:0.0 ~yalign:0.0
     ~packing:(box1#pack ~expand:false ~fill:false ~padding:0) () in

      (* Packing is done in make_box () *)
      make_box ~homogeneous:false ~spacing:10
        ~expand:true ~fill:false ~padding:0
        ~packing:(box1#pack ~expand:false ~fill:false ~padding:0) ();

      (* Same: packing is done in make_box () *)
      make_box ~homogeneous:false ~spacing:10
        ~expand:true ~fill:true ~padding:0
        ~packing:(box1#pack ~expand:false ~fill:false ~padding:0) ();

      (* Packing is done in GMisc.separator () *)
      GMisc.separator `HORIZONTAL
        ~packing:(box1#pack ~expand:false ~fill:true ~padding:5) ();

      (* Alignment and packing is done in GMisc.label () *)
      GMisc.label ~text:"hbox_new (false, 0);"
        ~xalign:0.0 ~yalign:0.0
        ~packing:(box1#pack ~expand:false ~fill:false ~padding:0) ();

      make_box ~homogeneous:false ~spacing:0
        ~expand:true ~fill:false ~padding:10
        ~packing:(box1#pack ~expand:false ~fill:false ~padding:0) ();

      make_box ~homogeneous:false ~spacing:0
        ~expand:true ~fill:true ~padding:10
```

```
        ~packing:(box1#pack ~expand:false ~fill:false ~padding:0) ();

      GMisc.separator `HORIZONTAL
        ~packing:(box1#pack ~expand:false ~fill:true ~padding:5) ();

      ()

  | 3 ->
      (* This demonstrates the ability to use "pack ~from:`END" to
       * right justify widgets. First, we create a new box as before. *)
      let box2 = make_box ~homogeneous:false ~spacing:0
        ~expand:false ~fill:false ~padding:0 () in

      (* Create the label that will be put at the end.
       * and pack it using "pack ~from:`END", so it is put on the right
       * side of the hbox created in the make_box () call. *)
      let label = GMisc.label ~text:"end"
     ~packing:(box2#pack ~from:`END ~expand:false ~fill:false ~padding:0) () in
      (* Pack box2 into box1 (the vbox remember ? :) *)
      box1#pack ~expand:false ~fill:false ~padding:0 box2#coerce;

      (* A separator for the bottom.
       * And pack the separator into the vbox (box1) created near the start
       * of main () *)
      let separator = GMisc.separator `HORIZONTAL
        ~packing:(box1#pack ~expand:false ~fill:true ~padding:5) () in

      (* This explicitly set the separstor to 400 pixels wide by 5 pixels
       * high. This is so the hbox we created will also be 400 pixels wide,
       * and the "end" label will be separated from the other labels in the
       * hbox. Otherwise, all the widgets in the hbox would be packed as
       * close together as possible. *)
      separator#misc#set_size_request ~width:400 ~height:5 ()

  | _ -> ()
  end;

  (* Create another new hbox. remember we can use as many as we need! *)
  let quitbox = GPack.hbox ~homogeneous:false ~spacing:0
    ~packing:(box1#pack ~expand:false ~fill:false ~padding:0) () in

  (* Our quit button. *)
  let button = GButton.button ~label:"Quit"
    ~packing:(box1#pack ~expand:false ~fill:false ~padding:0) () in
  button#connect#clicked ~callback:GMain.Main.quit;

  (* Showing the window last so everything pops up at once. *)
  window#show ();

  (* And of course, our main function. *)
  GMain.Main.main ()

let _ = Printexc.print main ()
```

5.4. Packing Using Tables

Let's take a look at another way of packing - Tables. These can be extremely useful in certain situations.

Using tables, we create a grid that we can place widgets in. The widgets may take up as many spaces as we specify.

The first thing to look at, of course, is the `GPack.table` function:

```
val GPack.table :
 ?columns:int ->
 ?rows:int ->
 ?homogeneous:bool ->
 ?row_spacings:int ->
 ?col_spacings:int ->
 ?border_width:int ->
 ?width:int ->
```

```
 ?height:int ->
 ?packing:(GObj.widget -> unit) ->
 ?show:bool -> unit -> table
```

The `?rows` specifies the number of rows to make in the table, while the `?columns`, obviously, is the number of columns.

The `?homogeneous` argument has to do with how the table's boxes are sized. If homogeneous is true, the table boxes are resized to the size of the largest widget in the table. If homogeneous is false, the size of a table boxes is dictated by the tallest widget in its same row, and the widest widget in its column.

The rows and columns are laid out from 0 to n, where n was the number specified in the call to GPack.table. So, if you specify rows = 2 and columns = 2, the layout would look something like this:

```
 0          1          2
0+----------+----------+
 |          |          |
1+----------+----------+
 |          |          |
2+----------+----------+
```

Note that the coordinate system starts in the upper left hand corner. To place a widget into a box, use the following function:

```
method attach :
 left:int ->
 top:int ->
 ?right:int ->
 ?bottom:int ->
 ?expand:Gtk.Tags.expand_type ->
 ?fill:Gtk.Tags.expand_type ->
 ?shrink:Gtk.Tags.expand_type ->
 ?xpadding:int ->
 ?ypadding:int -> GObj.widget -> unit

left : column number to attach the left side of the widget to
top : row number to attach the top of the widget to
right : default value is left+1
bottom : default value is top+1
expand : default value is 'NONE
fill : default value is 'BOTH
shrink : default value is 'NONE
```

The left and right attach arguments specify where to place the widget, and how many boxes to use. If you want a button in the lower right table entry of our 2x2 table, and want it to fill that entry *only*, left would be = 1, right = 2, top = 1, bottom = 2.

Now, if you wanted a widget to take up the whole top row of our 2x2 table, you'd use left = 0, right = 2, top = 0, bottom = 1.

The other options are:

`?fill`

: If the table box is larger than the widget, and `?fill` is specified, the widget will expand to use all the room available.

`?shrink`

: If the table widget was allocated less space then was requested (usually by the user resizing the window), then the widgets would normally just be pushed off the bottom of the window and disappear. If `?shrink` is specified, the widgets will shrink with the table.

`?expand`

: This will cause the table to expand to use up any remaining space in the window.

Padding is just like in boxes, creating a clear area around the widget specified in pixels.

We also have set_row_spacing and set_col_spacing meethods. These places spacing between the rows at the specified row or column.

```
method set_row_spacing : int -> int -> unit
```

and

```
method set_col_spacing : int -> int -> unit
```

Note that for columns, the space goes to the right of the column, and for rows, the space goes below the row.

You can also set a consistent spacing of all rows and/or columns with:

```
method set_row_spacings : int -> unit
```

And,

```
method set_col_spacings : int -> unit
```

Note that with these calls, the last row and last column do not get any spacing.

5.5. Table Packing Example

Here we make a window with three buttons in a 2x2 table. The first two buttons will be placed in the upper row. A third, quit button, is placed in the lower row, spanning both columns. Which means it should look something like this:

Here's the source code:

```
(* file: table.ml *)

(* Our callback. *)
let hello msg () =
  Printf.printf "Hello again - %s was pressed\n" msg;
  flush stdout

let main () =
  (* Create a new window; set title and border width *)
  let window = GWindow.window ~title:"Table" ~border_width:20 () in

  (* Set a handler for destroy event that immediately exits GTK. *)
  window#connect#destroy ~callback:GMain.Main.quit;

  (* Create a 2x2 table and put it in the main window *)
  let table = GPack.table ~rows:2 ~columns:2 ~homogeneous:true
      ~packing:window#add () in

  (* Create first button *)
  let button = GButton.button ~label:"button 1" () in

  (* Insert button 1 into the upper left quadrant of the table *)
  table#attach ~left:0 ~top:0 (button#coerce);

  (* When the button is clicked, we call the "callback" function
   * with "button 1" as its argument *)
  button#connect#clicked ~callback:(hello "button 1");

  (* Create second button *)
  let button2 = GButton.button ~label:"button 2" () in
```

```
  (* Insert button 2 into the upper right quadrant of the table *)
  table#attach ~left:1 ~top:0 (button2#coerce);

  (* When the button is clicked, we call the "callback" function
   * with "button 2" as its argument *)
  button2#connect#clicked ~callback:(hello "button 2");

  (* Create "Quit" button *)
  let quit = GButton.button ~label:"Quit" () in

  (* Insert the quit button into the both
   * lower quadrants of the table *)
  table#attach ~left:0 ~right:2 ~top:1 (quit#coerce);

  (* When the "Quit" button is clicked, the program exits *)
  quit#connect#clicked ~callback:GMain.Main.quit;

  window#show ();
  GMain.Main.main ()

let _ = Printexc.print main ()
```

Chapter 6. Widget Overview

The general steps to creating a widget in GTK are:

1. Creation - one of various functions to create a new widget. The returned widget is an object and you can use it to get/set values or do something. These are all detailed in this section.
2. Set the attributes of the widget. This can be done on widget creation.
3. Pack the widget into a container using the appropriate call such as add or pack method. This can be done on widget creation.
4. Connect all signals and events we wish to use to the appropriate handlers.
5. If it is toplevel window, show() the widget. Except toplevel window, show is default.

show method lets GTK know that we are done setting the attributes of the widget, and it is ready to be displayed. You may also use hide metehod to make it disappear again. The order in which you show the widgets is not important, but I suggest showing the window last so the whole window pops up at once rather than seeing the individual widgets come up on the screen as they're formed. The children of a widget (a window is a widget too) will not be displayed until the window itself is shown using the show() method.

6.1. Type Conversion

You'll notice as you go on that you need a type conversion. What you will see are:

```
method as_widget : Gtk.widget Gtk.obj
method coerce : widget
```

These are all used to cast arguments in functions. You'll see them in the examples, and can usually tell when to use them simply by looking at the function's declaration.

For example:

```
box#pack button#corece;
```

This casts the button into a widget.

6.2. Widget Hierarchy

> **Note:** This section shows the hierarchy of widgets in C version. I'm not sure this is correct or not in ocaml version. And the names of widgets used in this section are C version, not ocaml version.

For your reference, here is the class hierarchy tree used to implement widgets. (Deprecated widgets and auxiliary classes have been omitted.)

```
GObject
 |
 GtkObject
  +GtkWidget
  | +GtkMisc
  | | +GtkLabel
  | | | 'GtkAccelLabel
  | | +GtkArrow
  | | 'GtkImage
  | +GtkContainer
  | | +GtkBin
  | | | +GtkAlignment
  | | | +GtkFrame
  | | | | 'GtkAspectFrame
  | | | +GtkButton
  | | | | +GtkToggleButton
  | | | | | 'GtkCheckButton
  | | | | |   'GtkRadioButton
  | | | | 'GtkOptionMenu
  | | | +GtkItem
```

```
  | | | | +GtkMenuItem
  | | | |   +GtkCheckMenuItem
  | | | |   | ‘GtkRadioMenuItem
  | | | |   +GtkImageMenuItem
  | | | |   +GtkSeparatorMenuItem
  | | | |   ‘GtkTearoffMenuItem
  | | | +GtkWindow
  | | | | +GtkDialog
  | | | | | +GtkColorSelectionDialog
  | | | | | +GtkFileSelection
  | | | | | +GtkFontSelectionDialog
  | | | | | +GtkInputDialog
  | | | | | ‘GtkMessageDialog
  | | | | ‘GtkPlug
  | | | +GtkEventBox
  | | | +GtkHandleBox
  | | | +GtkScrolledWindow
  | | | ‘GtkViewport
  | | +GtkBox
  | | | +GtkButtonBox
  | | | | +GtkHButtonBox
  | | | | ‘GtkVButtonBox
  | | | +GtkVBox
  | | | | +GtkColorSelection
  | | | | +GtkFontSelection
  | | | | ‘GtkGammaCurve
  | | | ‘GtkHBox
  | | |   +GtkCombo
  | | |   ‘GtkStatusbar
  | | +GtkFixed
  | | +GtkPaned
  | | | +GtkHPaned
  | | | ‘GtkVPaned
  | | +GtkLayout
  | | +GtkMenuShell
  | | | +GtkMenuBar
  | | | ‘GtkMenu
  | | +GtkNotebook
  | | +GtkSocket
  | | +GtkTable
  | | +GtkTextView
  | | +GtkToolbar
  | | ‘GtkTreeView
  | +GtkCalendar
  | +GtkDrawingArea
  | | ‘GtkCurve
  | +GtkEditable
  | | +GtkEntry
  | |   ‘GtkSpinButton
  | +GtkRuler
  | | +GtkHRuler
  | | ‘GtkVRuler
  | +GtkRange
  | | +GtkScale
  | | | +GtkHScale
  | | | ‘GtkVScale
  | | ‘GtkScrollbar
  | |   +GtkHScrollbar
  | |   ‘GtkVScrollbar
  | +GtkSeparator
  | | +GtkHSeparator
  | | ‘GtkVSeparator
  | +GtkInvisible
  | +GtkPreview
  | ‘GtkProgressBar
  +GtkAdjustment
  +GtkCellRenderer
  | +GtkCellRendererPixbuf
  | +GtkCellRendererText
  | +GtkCellRendererToggle
  +GtkItemFactory
```

```
+GtkTooltips
‘GtkTreeViewColumn
```

6.3. Widgets Without Windows

Note: The names of widgets used in this section are C version, not ocaml version.

The following widgets do not have an associated window. If you want to capture events, you’ll have to use the EventBox. See the section on the EventBox widget.

```
GtkAlignment
GtkArrow
GtkBin
GtkBox
GtkButton
GtkCheckButton
GtkFixed
GtkImage
GtkLabel
GtkMenuItem
GtkNotebook
GtkPaned
GtkRadioButton
GtkRange
GtkScrolledWindow
GtkSeparator
GtkTable
GtkToolbar
GtkAspectFrame
GtkFrame
GtkVBox
GtkHBox
GtkVSeparator
GtkHSeparator
```

We’ll further our exploration of GTK by examining each widget in turn, creating a few simple functions to display them. Another good source is the `testgtk` program that comes with LablGtk. It can be found in `examples/testgtk.ml`.

6.4. Structure of Widgets

The following modules provide classes to wrap the raw Gtk+ function calls. Here are the widget classes contained in each module:

- `GPango`: Pango font handling
- `Gdk`: pixmaps, etc...
- `GObj`: gtkobj, widget, style
- `GData`: data, adjustment, tooltips
- `GContainer`: container, item_container
- `GWindow`: window, dialog, color_selection_dialog, file_selection, plug
- `GPack`: box, button_box, table, fixed, layout, packer, paned, notebook
- `GBin`: scrolled_window, event_box, handle_box, frame, aspect_frame, viewport, socket
- `GButton`: button, toggle_button, check_button, radio_button, toolbar
- `GMenu`: menu_item, tearoff_item, check_menu_item, radio_menu_item, menu_shell, menu, option_menu, menu_bar, factory
- `GMisc`: separator, statusbar, calendar, drawing_area, misc, arrow, image, pixmap, label, tips_query, color_selection, font_selection

- `GTree`: tree_item, tree, view (also tree/list_store, model)
- `GList`: list_item, liste, clist
- `GEdit`: editable, entry, spin_button, combo
- `GRange`: progress, progress_bar, range, scale, scrollbar
- `GText`: view (also buffer, iter, mark, tag, tagtable)

Practically, each widget class is composed of:

- `coerce` method, returning the object coerced to the type widget.
- `as_widget` method, returning the raw Gtk widget used for packing, etc...
- `destroy` method, sending the destroy signal to the object.
- `get_oid` method, the equivalent of Oo.id for Gtk objects.
- `connect` sub-object, allowing one to widget specific signals (this is what prevents width subtyping in sub-classes.)
- `misc` sub-object, giving access to miscellanous functionality of the basic gtkwidget class, and a misc#connect sub-object.
- `event` sub-object, for Xevent related functions (only if the widget has an Xwindow), and an event#connect sub-object.
- `drag` sub-object, containing drag and drop functions, and a drag#connect sub-object.
- widget specific methods.

Here is a diagram of the structure (- for methods, + for sub-objects)

```
- coerce : widget
- as_widget : Gtk.widget obj
- destroy : unit -> unit
- get_oid : int
- ...
+ connect : mywidget_signals
|   - after
|   - signal_name : callback:(... -> ...) -> GtkSignal.id
+ misc : misc_ops
|   - show, hide, disconnect, ...
|   + connect : misc_signals
+ drag : drag_ops
|   - ...
|   + connect : drag_signals
+ event : event_ops
|   - add, ...
|   + connect : event_signals
```

Chapter 7. Creating Widgets

You create a widget by

```
[Module].[widget name] options ... ()
```

Many optional arguments are admitted. The last two of them, packing: and show:, allow you respectively to call a function on your newly created widget, and to decide wether to show it immediately or not. By default all widgets except toplevel windows (GWindow module) are shown immediately.

7.1. Default Arguments

For many constructor or method arguments, default values are provided. Generally, this default value is defined by GTK, and you must refer to GTK's documentation.

For ML defined defaults, usually default values are either false, 0, None or 'NONE, according to the expected type.

Important exceptions are ~show, which default to true in all widgets except those in GWindow, and ~fill, which defaults to true or 'BOTH.

7.2. Memory Management

Important efforts have been dedicated to cooperate with Gtk's reference counting mechanism. As a result you should generally be able to use Gdk/Gtk data structures without caring about memory management. They will be freed when nobody points to them any more. This also means that you do not need to pay too much attention to whether a data structure is still alive or not. If it is not, you should get an error rather than a core dump. The case of Gtk objects deserves special care. Since they are interactive, we cannot just destroy them when they are no longer referenced. They have to be explicitely destroyed. If a widget was added to a container widget, it will automatically be destroyed when its last container is destroyed. For this reason you need only destroy toplevel widgets.

IMPORTANT: Some Gtk data structures are allocated in the Caml heap, and there use in signals (Gtk functions internally cally callbacks) relies on their address being stable during a function call. For this reason automatic compaction is disabled in GtkMain. If you need it, you may use compaction through Gc.compact where it is safe (timeouts, other threads...), but do not enable automatic compaction.

Chapter 8. The Button Widget

8.1. Normal Buttons

We've almost seen all there is to see of the button widget. It's pretty simple. There is however more than one way to create a button. You can use the `GButton.button` function with `~label` or `~mnemonic` option to create a button with a label, use `~stock` option to create a button containing the image and text from a stock item or use it without these options to create a blank button. It's then up to you to pack a label or pixmap into this new button. To do this, create a new box which is packed into button using `#add` method, and then pack your objects into this box using the usual `#pack` method.

Here's an example of using `GButton.button` to create a button with a image and a label in it. I've broken up the code to create a box from the rest so you can use it in your programs. There are further examples of using images later in the tutorial.


```
(* file: button.ml *)

open GMain

(* Create a new hbox with an image and a label packed into it
 * and pack the box *)
let xpm_label_box ~file ~text ~packing () =
  if not (Sys.file_exists file) then failwith (file ^ " does not exist");

  (* Create box for image and label and pack *)
  let box = GPack.hbox ~border_width:2 ~packing () in

  (* Now on to the image stuff and pack into box *)
  let pixmap = GDraw.pixmap_from_xpm ~file () in
  GMisc.pixmap pixmap ~packing:(box#pack ~padding:3) ();

  (* Create a label for the button and pack into box *)
  GMisc.label ~text ~packing:(box#pack ~padding:3) ()

let main () =
  (* Create a new window; set title and border_width *)
  let window = GWindow.window ~title:"Pixmap'd Buttons!" ~border_width:10 () in

  (* It's a good idea to do this for all windows. *)
  window#connect#destroy ~callback:Main.quit;
  window#event#connect#delete ~callback:(fun _ -> Main.quit (); true);

  (* Create a new button and pack *)
  let button = GButton.button ~packing:window#add () in

  (* Connect the "clicked" signal of the button to callback *)
  button#connect#clicked ~callback:
    (fun () -> print_endline "Hello again - cool button was pressed");

  (* Create box with xpm and label and pack into button *)
  xpm_label_box ~file:"info.xpm" ~text:"cool button" ~packing:button#add ();

  (* Show the window and wait for the fun to begin! *)
  window#show ();
  Main.main ()

let _ = main ()
```

The xpm_label_box() function could be used to pack images and labels into any widget that can be a container.

The Button widget has the following signals; see `GButton.button_signals`:

- `pressed` - emitted when pointer button is pressed within Button widget
- `released` - emitted when pointer button is released within Button widget
- `clicked` - emitted when pointer button is pressed and then released within Button widget
- `enter` - emitted when pointer enters Button widget
- `leave` - emitted when pointer leaves Button widget

8.2. Toggle Buttons

Toggle buttons are derived from normal buttons and are very similar, except they will always be in one of two states, alternated by a click. They may be depressed, and when you click again, they will pop back up. Click again, and they will pop back down.

Toggle buttons are the basis for check buttons and radio buttons, as such, many of the calls used for toggle buttons are inherited by radio and check buttons. I will point these out when we come to them.

Creating a new toggle button; see `GButton.toggle_button`:

```
val GButton.toggle_button :
 ?label:string ->
 ?use_mnemonic:bool ->
 ?stock:GtkStock.id ->
 ?relief:Gtk.Tags.relief_style ->
 ?active:bool ->
 ?draw_indicator:bool ->
 ?packing:(GObj.widget -> unit) ->
 ?show:bool -> unit -> toggle_button
```

As you can imagine, these work identically to the normal button widget calls. You can create a blank toggle button, and with `~lablel` or `~use_mnemonic`, a button with a label widget already packed into it. The `use_mnemonic` variant additionally parses the label for '_'-prefixed mnemonic characters.

To retrieve the state of the toggle widget, including radio and check buttons, we use a construct as shown in our example below. This tests the state of the toggle button, by calling the `active` method of the toggle widget's structure. The signal of interest to us emitted by toggle buttons (the toggle button, check button, and radio button widgets) is the "toggled" signal. To check the state of these buttons, set up a signal handler to catch the toggled signal, and get its state. The callback will look something like:

```
method toggled : callback:(unit -> unit) -> GtkSignal.id
```

To force the state of a toggle button, and its children, the radio and check buttons, use this method:

```
method set_active : bool -> unit
```

The above call can be used to set the state of the toggle button, and its children the radio and check buttons. Passing a `true` or `false` as the argument to specify whether it should be down (depressed) or up (released). Default is up, or `false`.

Note that when you use the `set_active` method, and the state is actually changed, it causes the "clicked" and "toggled" signals to be emitted from the button.

```
method active : bool
```

This returns the current state of the toggle button.

8.3. Check Buttons

Check buttons inherit many properties and functions from the the toggle buttons above, but look a little different. Rather than being buttons with text inside them, they are small squares with the text to the right of them. These are often used for toggling options on and off in applications.

The creation functions are similar to those of the normal button. See `GButton.check_button`.

```
val GButton.check_button :
 ?label:string ->
 ?use_mnemonic:bool ->
 ?stock:GtkStock.id ->
 ?relief:Gtk.Tags.relief_style ->
 ?active:bool ->
 ?draw_indicator:bool ->
 ?packing:(GObj.widget -> unit) ->
 ?show:bool -> unit -> toggle_button
```

This function call with `~lablel` option creates a check button with a label beside it.

Checking the state of the check button is identical to that of the toggle button.

8.4. Radio Buttons

Radio buttons are similar to check buttons except they are grouped so that only one may be selected/depressed at a time. This is good for places in your application where you need to select from a short list of options.

Creating a new radio button is done with one of these calls; see `GButton.radio_button`:

```
val GButton.radio_button :
 ?group:Gtk.radio_button Gtk.group ->
 ?label:string ->
 ?use_mnemonic:bool ->
 ?stock:GtkStock.id ->
 ?relief:Gtk.Tags.relief_style ->
 ?active:bool ->
 ?draw_indicator:bool ->
 ?packing:(GObj.widget -> unit) ->
 ?show:bool -> unit -> radio_button
```

You'll notice the extra argument to these calls. They require a group to perform their duty properly. The first call to radio_button should not pass the `~group` argument. Then create a group using:

```
method group : Gtk.radio_button Gtk.group
```

The important thing to remember is that `group` method must be called for each new button added to the group. The result is then passed into the next call to radio_button. This allows a chain of buttons to be established. The example below should make this clear.

You can shorten this slightly by using the following syntax, which removes the need for a variable to hold the list of buttons:

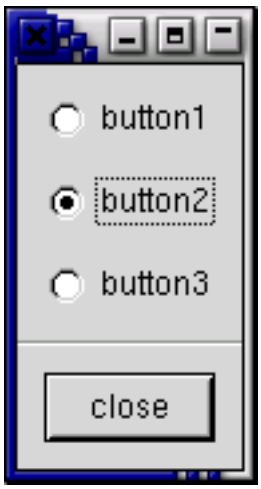
```
let button2 = GButton.radio_button ~label:"button2" ~group:button1#group () in
```

It is also a good idea to explicitly set which button should be the default depressed button with:

```
method set_active : bool -> unit
```

This is described in the section on toggle buttons, and works in exactly the same way. Once the radio buttons are grouped together, only one of the group may be active at a time. If the user clicks on one radio button, and then on another, the first radio button will first emit a "toggled" signal (to report becoming inactive), and then the second will emit its "toggled" signal (to report becoming active).

The following example creates a radio button group with three buttons.




```
(* file: radiobutton.ml *)

open GMain

let main () =
  let window = GWindow.window ~title:"radio buttons" ~border_width:0 () in
  window#connect#destroy ~callback:Main.quit;

  let box1 = GPack.vbox ~packing:window#add () in

  let box2 = GPack.vbox ~spacing:10 ~border_width:10 ~packing:box1#add () in

  let button1 = GButton.radio_button ~label:"button1" ~packing:box2#add () in

  let button2 = GButton.radio_button ~group:button1#group ~label:"button2"
      ~active:true ~packing:box2#add () in

  let button3 = GButton.radio_button
      ~group:button1#group ~label:"button3" ~packing:box2#add () in

  let separator = GMisc.separator 'HORIZONTAL ~packing: box1#pack () in

  let box3 = GPack.vbox ~spacing:10 ~border_width:10 ~packing:box1#pack () in

  let button = GButton.button ~label:"close" ~packing:box3#add () in
  button#connect#clicked ~callback:Main.quit;
  button#grab_default ();

  window#show ();
  Main.main ()

let _ = main ()
```

Chapter 9. Adjustments

GTK has various widgets that can be visually adjusted by the user using the mouse or the keyboard, such as the range widgets, described in the Range Widgets section. There are also a few widgets that display some adjustable portion of a larger area of data, such as the text widget and the viewport widget.

Obviously, an application needs to be able to react to changes the user makes in range widgets. One way to do this would be to have each widget emit its own type of signal when its adjustment changes, and either pass the new value to the signal handler, or require it to look inside the widget's data structure in order to ascertain the value. But you may also want to connect the adjustments of several widgets together, so that adjusting one adjusts the others. The most obvious example of this is connecting a scrollbar to a panning viewport or a scrolling text area. If each widget has its own way of setting or getting the adjustment value, then the programmer may have to write their own signal handlers to translate between the output of one widget's signal and the "input" of another's adjustment setting function.

GTK solves this problem using the Adjustment object, which is not a widget but a way for widgets to store and pass adjustment information in an abstract and flexible form. The most obvious use of Adjustment is to store the configuration parameters and values of range widgets, such as scrollbars and scale controls. However, since Adjustments are derived from Object, they have some special powers beyond those of normal data structures. Most importantly, they can emit signals, just like widgets, and these signals can be used not only to allow your program to react to user input on adjustable widgets, but also to propagate adjustment values transparently between adjustable widgets.

You will see how adjustments fit in when you see the other widgets that incorporate them: Progress Bars, Viewports, Scrolled Windows, and others.

9.1. Creating an Adjustment

Many of the widgets which use adjustment objects do so automatically, but some cases will be shown in later examples where you may need to create one yourself. You create an adjustment using `GData.adjustment`:

```
val GData.adjustment :
 ?value:float ->
 ?lower:float ->
 ?upper:float ->
 ?step_incr:float ->
 ?page_incr:float ->
 ?page_size:float -> unit -> adjustment

lower : default value is 0.
upper : default value is 100.
step_incr : default value is 1.
page_incr : default value is 10.
page_size : default value is 10.
```

The `value` argument is the initial value you want to give to the adjustment, usually corresponding to the topmost or leftmost position of an adjustable widget. The `lower` argument specifies the lowest value which the adjustment can hold. The `step_increment` argument specifies the "smaller" of the two increments by which the user can change the value, while the `page_increment` is the "larger" one. The `page_size` argument usually corresponds somehow to the visible area of a panning widget. The `upper` argument is used to represent the bottom most or right most coordinate in a panning widget's child. Therefore it is *not* always the largest number that `value` can take, since the `page_size` of such widgets is usually non-zero.

9.2. Using Adjustments the Easy Way

The adjustable widgets can be roughly divided into those which use and require specific units for these values and those which treat them as arbitrary numbers. The group which treats the values as arbitrary numbers includes the range widgets (scrollbars and scales, the progress bar widget, and the spin button widget). These widgets are all the widgets which are typically "adjusted" directly by the user with the mouse or keyboard. They will treat the `lower` and `upper` values of an adjustment as a range within which the user can manipulate the adjustment's `value`. By default, they will only modify the `value` of an adjustment.

The other group includes the text widget, the viewport widget, the compound list widget, and the scrolled window widget. All of these widgets use pixel values for their adjustments. These are also all widgets which are typically "adjusted" indirectly using scrollbars. While all widgets which use adjustments can either create their own adjustments or use ones you supply, you'll generally want to let this particular category of widgets create its

own adjustments. Usually, they will eventually override all the values except the `value` itself in whatever adjustments you give them, but the results are, in general, undefined (meaning, you'll have to read the source code to find out, and it may be different from widget to widget).

Now, you're probably thinking, since text widgets and viewports insist on setting everything except the `value` of their adjustments, while scrollbars will *only* touch the adjustment's `value`, if you *share* an adjustment object between a scrollbar and a text widget, manipulating the scrollbar will automagically adjust the viewport widget? Of course it will! Just like this:

```
...

(* creates its own adjustments *)
let viewport = GBin.viewport () in

(* uses the newly-created adjustment for the scrollbar as well *)
let vscrollbar = GRange.scrollbar 'VERTICAL ~adjustment:viewport#vadjustment ()
in

...
```

9.3. Adjustment Internals

Ok, you say, that's nice, but what if I want to create my own handlers to respond when the user adjusts a range widget or a spin button, and how do I get at the value of the adjustment in these handlers?

You can use the following accessor to inspect the `value` of an adjustment:

```
method value : float
```

Since, when you set the `value` of an Adjustment, you generally want the change to be reflected by every widget that uses this adjustment, GTK provides this convenience function to do this:

```
method set_value : float -> unit
```

As mentioned earlier, Adjustment is a subclass of Object just like all the various widgets, and thus it is able to emit signals. This is, of course, why updates happen automagically when you share an adjustment object between a scrollbar and another adjustable widget; all adjustable widgets connect signal handlers to their adjustment's `value_changed` signal, as can your program. Here's the definition of this signal:

```
method value_changed : callback:(unit -> unit) -> GtkSignal.id
```

The various widgets that use the Adjustment object will emit this signal on an adjustment whenever they change its value. This happens both when user input causes the slider to move on a range widget, as well as when the program explicitly changes the value with set_value method. So, for example, if you have a scale widget, and you want to change the rotation of a picture whenever its value changes, you would create a callback like this:

```
let cb_rotate_picture adj picture () =
  picture#set_rotation adj#value;
  ...
```

and connect it to the scale widget's adjustment like this:

```
adj#connect#value_changed ~callback:(cb_rotate_picture adj picture);
```

What about when a widget needs to reconfigure the `upper` or `lower` fields of its adjustment, such as when a user adds more text to a text widget? In this case, you can use

```
method set_bounds :
 ?lower:float ->
 ?upper:float ->
 ?step_incr:float ->
 ?page_incr:float ->
 ?page_size:float -> unit -> unit
```

When an adjustment is reconfigured, it emits the `changed` signal, which looks like this:

```
method changed : callback:(unit -> unit) -> GtkSignal.id
```

Range widgets typically connect a handler to this signal, which changes their appearance to reflect the change - for example, the size of the slider in a scrollbar will grow or shrink in inverse proportion to the difference between the `lower` and `upper` values of its adjustment.

Now go forth and adjust!

Chapter 10. Range Widgets

The category of range widgets includes the ubiquitous scrollbar widget and the less common scale widget. Though these two types of widgets are generally used for different purposes, they are quite similar in function and implementation. All range widgets share a set of common graphic elements, each of which has its own X window and receives events. They all contain a "trough" and a "slider" (what is sometimes called a "thumbwheel" in other GUI environments). Dragging the slider with the pointer moves it back and forth within the trough, while clicking in the trough advances the slider towards the location of the click, either completely, or by a designated amount, depending on which mouse button is used.

As mentioned in Adjustments above, all range widgets are associated with an adjustment object, from which they calculate the length of the slider and its position within the trough. When the user manipulates the slider, the range widget will change the value of the adjustment.

10.1. Scrollbar Widgets

These are your standard, run-of-the-mill scrollbars. These should be used only for scrolling some other widget, such as a list, a text box, or a viewport (and it's generally easier to use the scrolled window widget in most cases). For other purposes, you should use scale widgets, as they are friendlier and more featureful.

There are separate types for horizontal and vertical scrollbars. There really isn't much to say about these. You create them with the `GRange.scrollbar` function:

```
val GRange.scrollbar :
 Gtk.Tags.orientation ->
 ?adjustment:GData.adjustment ->
 ?inverted:bool ->
 ?update_policy:Gtk.Tags.update_type ->
 ?packing:(GObj.widget -> unit) ->
 ?show:bool -> unit -> range

inverted : default value is false
update_policy : default value is 'CONTINUOUS
```

and that's about it.

The `'HORIZONTAL` or `'VERTICAL` should be given as an argument to specify the orientataion of the scrollbar.

The `adjustment` argument may not be given, in which case one will be created for you. Not specifying might actually be useful in this case, if you wish to pass the newly-created adjustment to the constructor function of some other widget which will configure it for you, such as a text widget.

10.2. Scale Widgets

Scale widgets are used to allow the user to visually select and manipulate a value within a specific range. You might want to use a scale widget, for example, to adjust the magnification level on a zoomed preview of a picture, or to control the brightness of a color, or to specify the number of minutes of inactivity before a screensaver takes over the screen.

10.2.1. Creating a Scale Widget

As with scrollbars, there is a widget type for horizontal and vertical scale widgets. (Most programmers seem to favour horizontal scale widgets.) The following function `GRange.scale` creates vertical or horizontal scale widgets according to the argument;`'VERTICAL` or `'HORIZONTAL`:

```
val GRange.scale :
 Gtk.Tags.orientation ->
 ?adjustment:GData.adjustment ->
 ?digits:int ->
 ?draw_value:bool ->
 ?value_pos:Gtk.Tags.position ->
 ?inverted:bool ->
 ?update_policy:Gtk.Tags.update_type ->
 ?packing:(GObj.widget -> unit) ->
 ?show:bool -> unit -> scale
```

```
digits : default value is 1
draw_value : default value is false
value_pos : default value is ‘LEFT
inverted : default value is false
update_policy : default value is ‘CONTINUOUS
```

The `adjustment` argument may be given which has already been created with `GData.adjustment`, or may not, in which case, an anonymous Adjustment is created with all of its values set to `0.0` (which isn't very useful in this case). In order to avoid confusing yourself, you probably want to create your adjustment with a `page_size` of `0.0` so that its `upper` value actually corresponds to the highest value the user can select. (If you're *already* thoroughly confused, read the section on Adjustments again for an explanation of what exactly adjustments do and how to create and manipulate them.)

10.2.2. Functions and Signals (well, functions, at least)

Scale widgets can display their current value as a number beside the trough. The default behaviour is to show the value, but you can change this with this function:

```
method set_draw_value : bool -> unit
```

The value displayed by a scale widget is rounded to one decimal point by default, as is the `value` field in its Adjustment. You can change this with:

```
method set_digits : int -> unit
```

where `digits` is the number of decimal places you want. You can set `digits` to anything you like, but no more than 13 decimal places will actually be drawn on screen.

Finally, the value can be drawn in different positions relative to the trough:

```
method set_value_pos : Gtk.Tags.position -> unit
```

The argument can take one of the following values:

```
‘LEFT
‘RIGHT
‘TOP
‘BOTTOM
```

If you position the value on the "side" of the trough (e.g., on the top or bottom of a horizontal scale widget), then it will follow the slider up and down the trough.

10.3. Common Range Functions

The Range widget class is fairly complicated internally, but, like all the "base class" widgets, most of its complexity is only interesting if you want to hack on it. Also, almost all of the functions and signals it defines are only really used in writing derived widgets. There are, however, a few useful functions that will work on all range widgets.

10.3.1. Setting the Update Policy

The "update policy" of a range widget defines at what points during user interaction it will change the `value` field of its Adjustment and emit the "value_changed" signal on this Adjustment. The update policies, defined as type `Gtk.Tags.update_type`, are:

`‘CONTINUOUS`

This is the default. The "value_changed" signal is emitted continuously, i.e., whenever the slider is moved by even the tiniest amount.

`‘DISCONTINUOUS`

The "value_changed" signal is only emitted once the slider has stopped moving and the user has released the mouse button.

`'DELAYED`

The "value_changed" signal is emitted when the user releases the mouse button, or if the slider stops moving for a short period of time.

The update policy of a range widget can be set by calling this function:

```
method set_update_policy : Gtk.Tags.update_type -> unit
```

10.3.2. Getting and Setting Adjustments

Getting and setting the adjustment for a range widget "on the fly" is done, predictably, with:

```
method adjustment : GData.adjustment
method set_adjustment : GData.adjustment -> unit
```

`adjustment` method returns the adjustment to which `range` is connected.

`set_adjustment` method does absolutely nothing if you pass it the adjustment that `range` is already using, regardless of whether you changed any of its fields or not. If you pass it a new Adjustment, it will disconnect the old one if it exists, connect the appropriate signals to the new one, and call the private function `gtk_range_adjustment_changed()`, which will (or at least, is supposed to...) recalculate the size and/or position of the slider and redraw if necessary.

10.4. Key and Mouse bindings

All of the GTK range widgets react to mouse clicks in more or less the same way. Clicking button-1 in the trough will cause its adjustment's `page_increment` to be added or subtracted from its `value`, and the slider to be moved accordingly. Clicking mouse button-2 in the trough will jump the slider to the point at which the button was clicked. Clicking button-3 in the trough of a range or any button on a scrollbar's arrows will cause its adjustment's value to change by `step_increment` at a time.

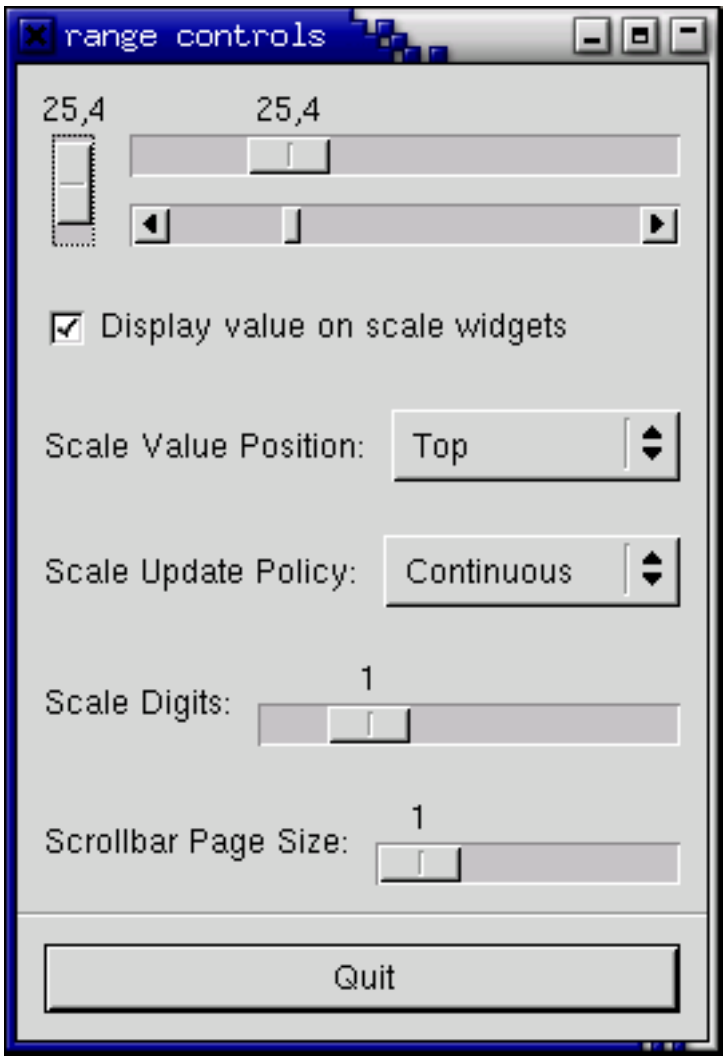
Scrollbars are not focusable, thus have no key bindings. The key bindings for the other range widgets (which are, of course, only active when the widget has focus) are do *not* differentiate between horizontal and vertical range widgets.

All range widgets can be operated with the left, right, up and down arrow keys, as well as with the `Page Up` and `Page Down` keys. The arrows move the slider up and down by `step_increment`, while `Page Up` and `Page Down` move it by `page_increment`.

The user can also move the slider all the way to one end or the other of the trough using the keyboard. This is done with the `Home` and `End` keys.

10.5. Example

It basically puts up a window with three range widgets all connected to the same adjustment, and a couple of controls for adjusting some of the parameters mentioned above and in the section on adjustments, so you can see how they affect the way these widgets work for the user.




```
(* file: range.ml *)

let cb_pos_menu_select hscale vscale pos () =
  hscale#set_value_pos pos;
  vscale#set_value_pos pos

let cb_update_menu_select hscale vscale policy () =
  hscale#set_update_policy policy;
  vscale#set_update_policy policy

let cb_digits_scale hscale vscale adj () =
  hscale#set_digits (int_of_float adj#value);
  vscale#set_digits (int_of_float adj#value)

let clamp x low high =
  if x > high then high
  else if x < low then low
  else x

let cb_page_size get (set: GData.adjustment) () =
  (* Set the page size and page increment size of the sample
   * adjustment to the value specified by the "Page Size" scale *)
  let v = get#value in
  set#set_bounds ~page_incr:v ~page_size:v ();

  (* This sets the adjustment and makes it emit the "chagnged" signal to
   * reconfigure all the widgets that are attatched to this signal. *)
   set#set_value (clamp set#value set#lower (set#upper -. set#page_size))

let cb_draw_value button hscale vscale () =
```

```
  hscale#set_draw_value button#active;
  vscale#set_draw_value button#active

let make_menu_item ~label ~packing ~callback =
  let item = GMenu.menu_item ~label ~packing () in
  ignore (item#connect#activate ~callback)

let create_range_controls () =
  (* Standard window-creating stuff *)
  let window = GWindow.window ~title:"Range controls" ~border_width:20 () in
  window#connect#destroy ~callback:GMain.Main.quit;

  let box1 = GPack.vbox ~packing:window#add () in
  let box2 = GPack.hbox ~border_width:10 ~packing:box1#add () in

  (* Note that the page_size value only makes a difference for
   * scrollbar widgets, and the highest value you'll get is actually
   * (upper - page_size). *)
  let adj1 = GData.adjustment ~value:0.0 ~lower:0.0 ~upper:101.0
    ~step_incr:0.1 ~page_incr:1.0 ~page_size:1.0 ()
  in
  let vscale = GRange.scale `VERTICAL ~adjustment:adj1 ~packing:box2#add () in

  let box3 = GPack.vbox ~packing:box2#add () in

  (* Reuse the same adjustment *)
  let hscale = GRange.scale `HORIZONTAL ~adjustment:adj1 ~packing:box3#add () in

  (* Reuse the same adjustment again.
   * Default update_policy is `CONTINUOUS.
   * Notice the scales always be updated continuously
   * when the scrollbar is moved *)
  let scrollbar = GRange.scrollbar `HORIZONTAL ~adjustment:adj1
    ~packing:box3#add ()
  in

  let box2 = GPack.hbox ~border_width:10 ~packing:box1#add () in

  (* A checkbutton to control whether the value is displayed or not *)
  let button = GButton.check_button ~label:"Display value on scale widgets"
    ~active:true ~packing:box2#add ()
  in
  button#connect#toggled ~callback:(cb_draw_value button hscale vscale);

  let box2 = GPack.hbox ~border_width:10 ~packing:box1#add () in
  let label = GMisc.label ~text:"Scale Value Position:" ~packing:box2#add () in

  let opt = GMenu.option_menu ~packing:box2#add () in
  let menu = GMenu.menu ~packing:opt#set_menu () in
  let f (label, pos) =
    make_menu_item ~label ~packing:menu#append
   ~callback:(cb_pos_menu_select hscale vscale pos)
  in
  List.iter f [("Top", `TOP); ("Bottom", `BOTTOM); ("Left", `LEFT);
    ("Right", `RIGHT)];

  let box2 = GPack.hbox ~border_width:10 ~packing:box1#add () in
  (* Yet another option menu, this time for the update policy of the
   * scale widgets *)
  let label = GMisc.label ~text:"Scale Update Policy:" ~packing:box2#add () in
  let opt = GMenu.option_menu ~packing:box2#add () in
  let menu = GMenu.menu ~packing:opt#set_menu () in
  let f (label, policy) =
    make_menu_item ~label ~packing:menu#append
   ~callback:(cb_update_menu_select hscale vscale policy)
  in
  List.iter f [("Continuous", `CONTINUOUS); ("Discontinuous", `DISCONTINUOUS);
    ("Delayed", `DELAYED)];

  let box2 = GPack.hbox ~border_width:10 ~packing:box1#add () in
  let label = GMisc.label ~text:"Scale Digits:" ~packing:box2#add () in
```

```
  let adj2 = GData.adjustment ~value:1.0 ~lower:0.0 ~upper:5.0
    ~step_incr:1.0 ~page_incr:1.0 ~page_size:0.0 ()
  in
  adj2#connect#value_changed ~callback:(cb_digits_scale hscale vscale adj2);

  let scale = GRange.scale 'HORIZONTAL ~adjustment:adj2
    ~digits:0 ~packing:box2#add ()
  in

  let box2 = GPack.hbox ~border_width:10 ~packing:box1#add () in
  (* And, one last Horizontal Scale widget for adjusting the page size of the
   * scrollbar *)
  let label = GMisc.label ~text:"Scrollbar Page Size:" ~packing:box2#add () in

  let adj2 = GData.adjustment ~value:1.0 ~lower:0.0 ~upper:101.0
    ~step_incr:1.0 ~page_incr:1.0 ~page_size:0.0 ()
  in
  adj2#connect#value_changed ~callback:(cb_page_size adj2 adj1);
  let scale = GRange.scale 'HORIZONTAL ~adjustment:adj2
    ~digits:0 ~packing:box2#add ()
  in

  let separator = GMisc.separator 'HORIZONTAL ~packing:box1#add () in

  let box2 = GPack.hbox ~border_width:10 ~packing:box1#add () in
  let button = GButton.button ~label:"Quit" ~packing:box2#add () in
  button#connect#clicked ~callback:GMain.quit;
  button#misc#set_can_default true;
  button#misc#grab_default ();
  window#show ()

let main () =
  create_range_controls ();
  GMain.Main.main ()

let _ = Printexc.print main ()
```

You will notice that the program does not call connect() for the "delete_event", but only for the "destroy" signal. This will still perform the desired function, because an unhandled "delete_event" will result in a "destroy" signal being given to the window.

Chapter 11. Miscellaneous Widgets

11.1. Labels

Labels are used a lot in GTK, and are relatively simple. Labels emit no signals as they do not have an associated X window. If you need to catch signals, or do clipping, place it inside a EventBox widget or a Button widget.

To create a new label, use `GMisc.label`:

```
val GMisc.label : ?text:string ->
 ?markup:string ->
 ?use_underline:bool ->
 ?mnemonic_widget:#GObj.widget ->
 ?justify:Gtk.Tags.justification ->
 ?line_wrap:bool ->
 ?pattern:string ->
 ?selectable:bool ->
 ?xalign:float ->
 ?yalign:float ->
 ?xpad:int ->
 ?ypad:int ->
 ?width:int ->
 ?height:int ->
 ?packing:(GObj.widget -> unit) ->
 ?show:bool ->
 unit -> label

markup : overrides text if both are present
use_underline : default value is false
justify : default value is 'LEFT
line_wrap : default values is false
selectable : default value is false
```

To change the label's text after creation, use the function `GMisc.label#set_text`:

```
method set_text : string -> unit
```

The space needed for the new string will be automatically adjusted if needed. You can produce multi-line labels by putting line breaks in the label string.

To retrieve the current string, use `GMisc.label#text`:

```
method text : string
```

The label text can be justified using `GMisc.label#set_justify`:

```
method set_justify : Gtk.Tags.justification -> unit
```

Values for `Gtk.Tags.justification` are:

```
  'LEFT
  'RIGHT
  'CENTER (the default)
  'FILL
```

The label widget is also capable of line wrapping the text automatically. This can be activated using `GMisc.label#set_line_wrap`:

```
method set_line_wrap : bool -> unit
```

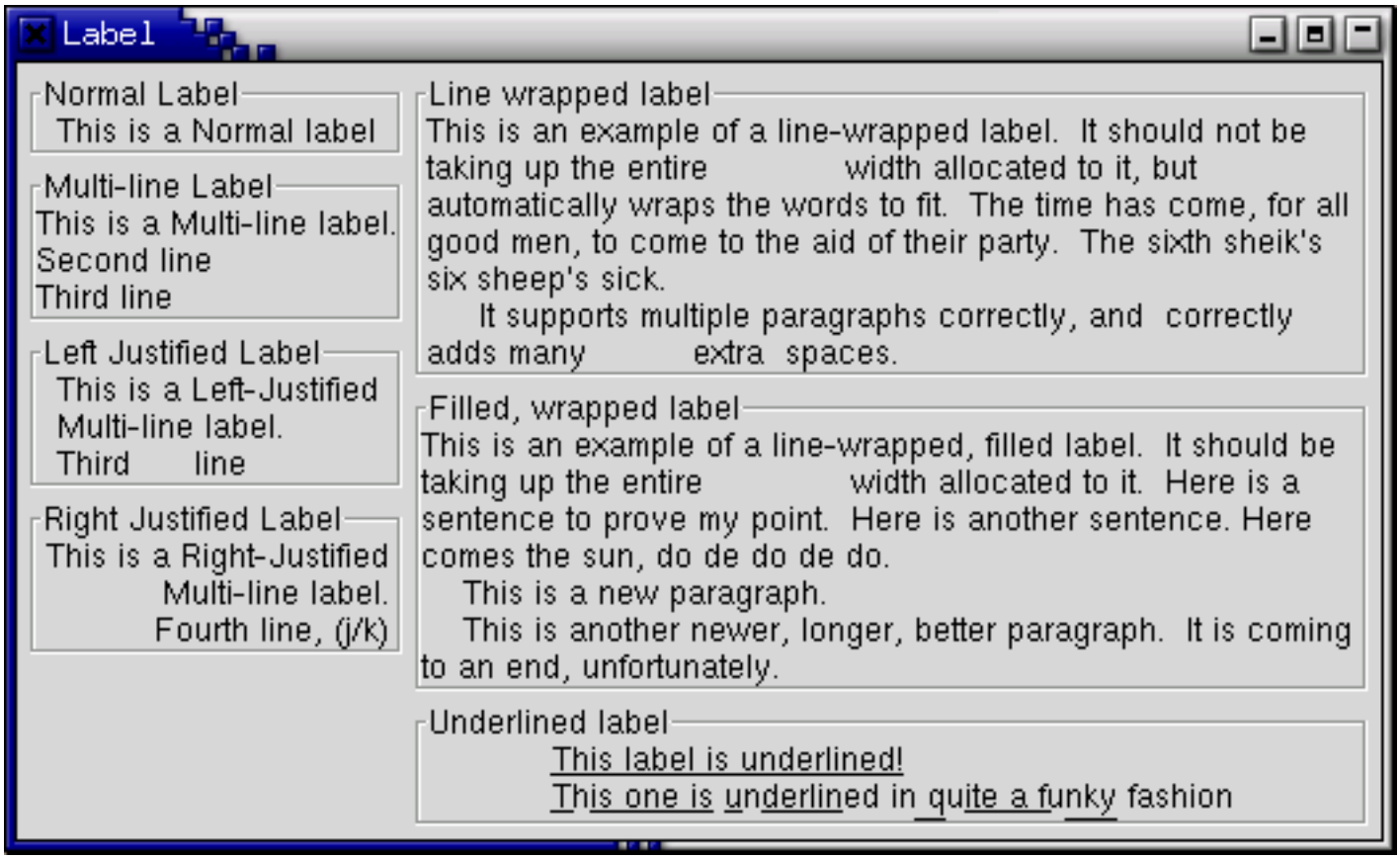
If you want your label underlined, then you can set a pattern on the label `GMisc.label#set_pattern`:

```
method set_pattern : string -> unit
```

The pattern argument indicates how the underlining should look. It consists of a string of underscore and space characters. An underscore indicates that the corresponding character in the label should be underlined. For example, the string "__ __" would underline the first two characters and eight and ninth characters.

Below is a short example to illustrate these functions. This example makes use of the Frame widget to better demonstrate the label styles. You can ignore this for now as the Frame widget is explained later on.

In GTK+ 2.0, label texts can contain markup for font and other text attribute changes, and labels may be selectable (for copy-and-paste). These advanced features won't be explained here.




```
(* file: label.ml *)

let main () =
  let window = GWindow.window ~title:"Labels" ~border_width:5 () in
  window #connect#destroy ~callback:GMain.Main.quit;

  let hbox = GPack.hbox ~spacing:5 ~packing:window#add () in
  let vbox = GPack.vbox ~spacing:5 ~packing:hbox#add () in

  let frame = GBin.frame ~label:"Normal Label" ~packing:vbox#pack () in
  GMisc.label ~text:"This is a normal label" ~packing:frame#add ();

  let frame = GBin.frame ~label:"Multi_line Label" ~packing:vbox#pack () in
  GMisc.label
    ~text:"This is a multi-line label.\nSecond line\nThird line"
    ~packing:frame#add ();

  let frame = GBin.frame ~label:"Left Justified Label" ~packing:vbox#pack () in
  GMisc.label
    ~text:"This is a left justified\nmulti_line label\nThird      line"
    ~justify:`LEFT ~packing:frame#add ();

  let frame = GBin.frame ~label:"Right Justified Label" ~packing:vbox#pack () in
  GMisc.label
    ~text:"This is a Right-Justified\nMulti-line label.\nThird line, (j/k)"
    ~justify:`RIGHT ~packing:frame#add ();

  let vbox = GPack.vbox ~spacing:5 ~packing:hbox#add () in

  let frame = GBin.frame ~label:"Line wrapped Label" ~packing:vbox#pack () in
  GMisc.label
    ~text:"This is an example of a line-wrapped label.  It should not be taking up the entire
```

```
     It supports multiple paragraphs correctly, and  correctly   adds many          extra  spaces. "
    ~packing:frame#add ~line_wrap:true ();

  let frame = GBin.frame ~label:"Filled, wrapped label" ~packing:vbox#pack () in
  GMisc.label
    ~text:"This is an example of a line-wrapped, filled label.  It should be taking up the entire
    This is a new paragraph.
    This is another newer, longer, better paragraph.  It is coming to an end, unfortunately."
    ~line_wrap:true ~justify:`FILL ~packing:frame#add ();

  let frame = GBin.frame ~label:"Underlined Label" ~packing:vbox#pack () in
  GMisc.label
    ~text:"This label is underlined!\nThis one is underlined in quite a funky fashion"
    ~pattern:"_____________________________ _ _________ _ ______     __ _______ ___"
    ~justify:`LEFT ~packing:frame#add ();

  window #show ();
  GMain.Main.main ()

let _ = Printexc.print main ()
```

11.2. Arrows

The Arrow widget draws an arrowhead, facing in a number of possible directions and having a number of possible styles. It can be very useful when placed on a button in many applications. Like the Label widget, it emits no signals.

`GMisc.arrrow` is the function for creating an Arrow widget:

```
val GMisc.arrow :
 ?kind:Gtk.Tags.arrow_type ->
 ?shadow:Gtk.Tags.shadow_type ->
 ?xalign:float ->
 ?yalign:float ->
 ?xpad:int ->
 ?ypad:int ->
 ?width:int ->
 ?height:int ->
 ?packing:(GObj.widget -> unit) ->
 ?show:bool -> unit -> arrow

 kind : default value is `RIGHT
 shadow : default value is `OUT
```

This creates a new arrow widget with the indicated type and appearance. The second allows these values to be altered retrospectively. The `?kind` argument may take one of the following values:

```
  `UP
  `DOWN
  `LEFT
  `RIGHT
```

These values obviously indicate the direction in which the arrow will point. The `?shadow` argument may take one of these values:

```
  `IN
  `OUT (the default)
  `ETCHED_IN
  `ETCHED_OUT
  `NONE
```

Here's a brief example to illustrate their use.

```
(* file: arrow.ml *)

(* Create an Arrow widget with the specified parameters
 * and pack in into a button *)
let create_arrow_button ~kind ~shadow ~packing () =
  let button = GButton.button ~packing () in
  let arrow = GMisc.arrow ~kind ~shadow ~packing:button#add () in
  button

let main () =
  (* Create a new window; set title and border width *)
  let window = GWindow.window ~title:"Arrow Buttons" ~border_width:10 () in

  (* Set a handler for destroy event that immediately exits GTK. *)
  window#connect#destroy ~callback:GMain.Main.quit;

  (* Create a box to hold the arrow/buttons *)
  let box = GPack.hbox ~border_width:2 ~packing:window#add () in

  let f (kind, shadow) =
    create_arrow_button ~kind ~shadow ~packing:box#add ();
    ()
  in
  List.iter f [(‘UP, ‘IN); (‘DOWN, ‘OUT); (‘LEFT, ‘ETCHED_IN);
    (‘RIGHT, ‘ETCHED_OUT) ];

  window#show ();
  (* Rest in main and wait for the fun to begin! *)
  GMain.Main.main ()

let _ = Printexc.print main ()
```

11.3. The Tooltips Object

These are the little text strings that pop up when you leave your pointer over a button or other widget for a few seconds. They are easy to use, so I will just explain them without giving an example. If you want to see some code, take a look at the `testgtk.ml` program distributed with LablGTK.

Widgets that do not receive events (widgets that do not have their own window) will not work with tooltips.

The first call you will use creates a new tooltip using `GData.tooltips` function. You only need to do this once for a set of tooltips as the `GData.tooltips object` this function returns can be used to create multiple tooltips.

```
val GData.tooltips : ?delay:int -> unit -> tooltips
```

Once you have created a new tooltip, and the widget you wish to use it on, simply use this call to set it:

```
method set_tip : ?text:string -> ?privat:string -> GObj.widget -> unit
```

The `?text` argument is the text you with to say, which is followed by the widget you wish to have this tooltip pop up for wish it to say. The `?privat` argument is a text string that can be used as an identifier when using GtkTipsQuery to implement context sensitive help. For now, you don’t have to give it.

Here’s a short example:

```
let tooltips = GData.tooltips () in
let button = GButton.button ~label:"This is button1" () in
.
.
.
tooltips#set_tip button#coerce ~text:"This is button1";
```

There are other calls that can be used with tooltips. I will just list them with a brief description of what they do.

```
method enable : unit -> unit
```

Enable a disabled set of tooltips.

```
method disable : unit -> unit
```

Disable an enabled set of tooltips.

11.4. Progress Bars

Progress bars are used to show the status of an operation. They are pretty easy to use, as you will see with the code below. But first lets start out with the calls to create a new progress bar using `GRange.progress_bar`.

```
val GRange.progress_bar:
 ?orientation:Gtk.Tags.progress_bar_orientation ->
 ?pulse_step:float ->
 ?packing:(GObj.widget -> unit) ->
 ?show:bool -> unit -> progress_bar

orientation : default value is 'LEFT_TO_RIGHT
pulse_step : default value is 0.1
```

Now that the progress bar has been created we can use it.

```
method set_fraction : float -> unit
```

The argument is the amount "completed", meaning the amount the progress bar has been filled from 0-100%. This is passed to the method as a float number ranging from 0.0 to 1.0

GTK v1.2 has added new functionality to the progress bar that enables it to display its value in different ways, and to inform the user of its current value and its range.

A progress bar may be set to one of a number of orientations using the function

```
method set_orientation : Gtk.Tags.progress_bar_orientation -> unit
```

The `orientation` argument may take one of the following values to indicate the direction in which the progress bar moves:

```
'LEFT_TO_RIGHT
'RIGHT_TO_LEFT
'BOTTOM_TO_TOP
'TOP_TO_BOTTOM
```

As well as indicating the amount of progress that has occured, the progress bar may be set to just indicate that there is some activity. This can be useful in situations where progress cannot be measured against a value range. The following function indicates that some progress has been made.

```
method pulse : unit -> unit
```

The step size of the activity indicator is set using the following function.

```
method set_pulse_step : float -> unit
```

When not in activity mode, the progress bar can also display a configurable text string within its trough, using the following function.

```
method set_text : string -> unit
```

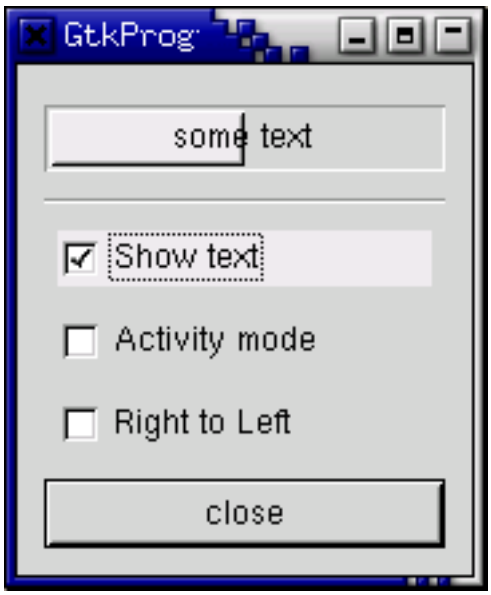
You can turn off the display of the string by calling set_text method again with empty string as an argument.

The current text setting of a progressbar can be retrieved with the following function.

```
method text : string
```

Progress Bars are usually used with timeouts or other such functions (see section on Timeouts and Idle Functions) to give the illusion of multitasking. All will employ the set_fraction or pulse methods in the same manner.

Here is an example of the progress bar, updated using timeouts. This code also shows you how to reset the Progress Bar.




```
(* file: progressbar.ml *)

let pulse_mode = ref false

(* Update the value of the progress bar so that we get
 * some movement *)
let progress_timeout pbar () =
  if !pulse_mode
  then pbar#pulse ()
  else (
    (* Calculate the value of the progress bar using the
     * value range set in the adjustment object *)
    let new_val =
      let v = pbar#fraction +. 0.01 in
      if v > 1.0 then 0.0 else v
    in
      pbar#set_fraction new_val
  );

  (* As this is a timeout function, return true so that it
   * continues to get called *)
  true

(* Callback that toggles the text display with the progress bar trough *)
let toggle_show_text pbar () =
  let text = pbar#text in
  if text = ""
  then pbar#set_text "some text"
  else pbar#set_text ""

(* Callback that toggles the activity mode of the progress bar *)
let toggle_activity_mode pbar () =
  pulse_mode := not !pulse_mode;
  if !pulse_mode
  then pbar#pulse ()
  else pbar#set_fraction 0.0

(* Callback that toggles the orientation of the progress bar *)
let toggle_orientation pbar () =
  match pbar#orientation with
  | `LEFT_TO_RIGHT -> pbar#set_orientation `RIGHT_TO_LEFT
  | `RIGHT_TO_LEFT -> pbar#set_orientation `LEFT_TO_RIGHT
  | _ -> ()
```

```
(* Remove timer and quit *)
let destroy_progress timer () =
  GMain.Timeout.remove timer;
  GMain.Main.quit ()

let main () =
  let window = GWindow.window ~title:"ProgressBar" () in

  let vbox = GPack.vbox ~border_width:10 ~packing:window#add () in

  (* Create a centering alignment object *)
  let align = GBin.alignment ~xalign:0.5 ~yalign:0.5
    ~xscale:0.0 ~yscale:0.0 ~packing:vbox#add ()
  in

  (* Create the progressbar *)
  let pbar = GRange.progress_bar ~packing:align#add () in

  (* Add a timer callback to update the value of the progress bar *)
  let timer = GMain.Timeout.add ~ms:100 ~callback:(progress_timeout pbar) in
  GMisc.separator 'HORIZONTAL ~packing:vbox#add ();

  let table = GPack.table ~rows:3 ~columns:1 ~homogeneous:false
    ~packing:vbox#add ()
  in

  (* Add a check button to select displaying of trough text *)
  let check = GButton.check_button ~label:"Show text"
    ~packing:(table#attach ~left:0 ~top:0) ()
  in
  check#connect#clicked ~callback:(toggle_show_text pbar);

  (* Add a check button to toggle activity mode *)
  let check = GButton.check_button ~label:"Activity mode"
    ~packing:(table#attach ~left:0 ~top:1) ()
  in
  check#connect#clicked ~callback:(toggle_activity_mode pbar);

  (* Add a check button to toggle orientation *)
  let check = GButton.check_button ~label:"Right to Left"
    ~packing:(table#attach ~left:0 ~top:2) ()
  in
  check#connect#clicked ~callback:(toggle_orientation pbar);

  (* Add a buton to exit the program *)
  let button = GButton.button ~label:"Close" ~packing:vbox#add () in
  button#connect#clicked ~callback:(destroy_progress timer);

  window#connect#destroy ~callback:(destroy_progress timer);
  window#show ();
  GMain.Main.main ()

let _ = main ()
```

11.5. Dialogs

The Dialog widget is very simple, and is actually just a window with a few things pre-packed into it for you.

It simply creates a window, and then packs a vbox into the top, which contains a separator and then an hbox called the "action_area". After creating a dialog, you can reference the vbox and the action_area using the following methods:

```
method vbox : GPack.box
method action_area : GPack.button_box
```

The Dialog widget can be used for pop-up messages to the user, and other similar tasks. `GWindow.dialog` is the function which creates a new Dialog.

```
val GWindow.dialog:
 ?no_separator:bool ->
 ?parent:#window_skel ->
 ?destroy_with_parent:bool ->
 ?title:string ->
 ?allow_grow:bool ->
 ?allow_shrink:bool ->
 ?icon:GdkPixbuf.pixbuf ->
 ?modal:bool ->
 ?resizable:bool ->
 ?screen:Gdk.screen ->
 ?type_hint:Gdk.Tags.window_type_hint ->
 ?position:Gtk.Tags.window_position ->
 ?wm_name:string ->
 ?wm_class:string ->
 ?border_width:int ->
 ?width:int ->
 ?height:int ->
 ?show:bool -> unit -> [> 'DELETE_EVENT ] dialog

no_separator : default value is false
destroy_with_parent : default value is false
```

This function will create an empty dialog, and it is now up to you to use it. You could pack a button in the action_area by doing something like this:

```
let w = GWindow.dialog ... () in
let button = GButton.button ... ~packing:w#action_area#add () in
```

And you could add to the vbox area by packing, for instance, a label in it, try something like this:

```
let w = GWindow.dialog ... () in
let label = GMisc.label ~text:"Dialogs are groovy" ~packing:w#vbox#add () in
```

As an example in using the dialog box, you could put two buttons in the action_area, a Cancel button and an Ok button, and a label in the vbox area, asking the user a question or giving an error etc. Then you could attach a different signal to each of the buttons and perform the operation the user selects.

If the simple functionality provided by the default vertical and horizontal boxes in the two areas doesn't give you enough control for your application, then you can simply pack another layout widget into the boxes provided. For example, you could pack a table into the vertical box.

The optional arguments allows to set one or more of the following flags.

`?modal`

make the dialog modal.

`?destroy_with_parent`

ensures that the dialog window is destroyed together with the parent.

`?no_separator`

omits the separator between the vbox and the action_area.

11.6. Rulers

Ruler widgets are used to indicate the location of the mouse pointer in a given window. A window can have a vertical ruler spanning across the width and a horizontal ruler spanning down the height. A small triangular indicator on the ruler shows the exact location of the pointer relative to the ruler.

A ruler must first be created. Horizontal and vertical rulers are created using `GRange.ruler`

```
val GRange.ruler:
 Gtk.Tags.orientation ->
 ?metric:Gtk.Tags.metric_type ->
 ?lower:float ->
 ?upper:float ->
 ?max_size:float ->
```

```
 ?position:float ->
 ?packing:(GObj.widget -> unit) ->
 ?show:bool -> unit -> ruler

metric : default value is 'PIXELS
```

Orientation should be given as an argument. It can be `'HORIZONTAL` for horizontal ruler and `'VERTICAL` for vertical ruler.

Units of measure for rulers can be `'PIXELS`, `'INCHES` or `'CENTIMETERS`. This can be set using

```
method set_metric : Gtk.Tags.metric_type -> unit
```

The default measure is `'PIXELS`.

Other important characteristics of a ruler are how to mark the units of scale and where the position indicator is initially placed. These are set for a ruler using

```
method set_lower : float -> unit
method set_upper : float -> unit
method set_position : float -> unit
method set_max_size : float -> unit
```

The lower and upper define the extent of the ruler, and max_size is the largest possible number that will be displayed. Position defines the initial position of the pointer indicator within the ruler.

A vertical ruler can span an 800 pixel wide window thus

```
vruler#set_lower 0;
vruler#set_upper 800;
vruler#set_position 0;
vruler#set_max_size 800;
```

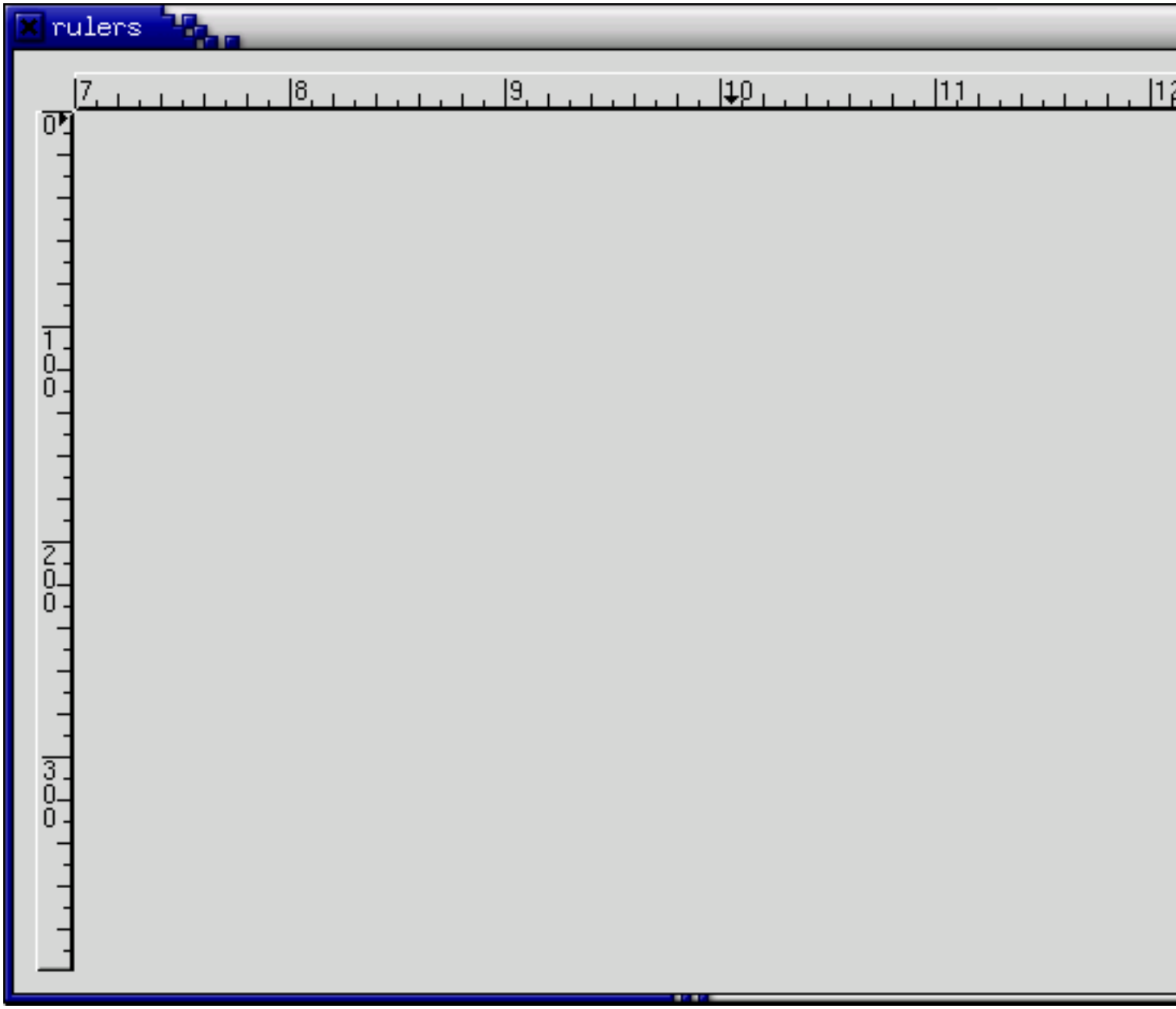
The markings displayed on the ruler will be from 0 to 800, with a number for every 100 pixels. If instead we wanted the ruler to range from 7 to 16, we would code

```
vruler#set_lower 7;
vruler#set_upper 16;
vruler#set_position 0;
vruler#set_max_size 20;
```

The indicator on the ruler is a small triangular mark that indicates the position of the pointer relative to the ruler. If the ruler is used to follow the mouse pointer, the motion_notify_event signal should be connected to the motion_notify_event method of the ruler. To follow all mouse movements within a window area, we would use

```
area#event#connect#motion_notify ~callback:(fun ev -> hruler#event#send (ev :> GdkEvent.any));
```

The following example creates a drawing area with a horizontal ruler above it and a vertical ruler to the left of it. The size of the drawing area is 600 pixels wide by 400 pixels high. The horizontal ruler spans from 7 to 13 with a mark every 100 pixels, while the vertical ruler spans from 0 to 400 with a mark every 100 pixels. Placement of the drawing area and the rulers is done using a table.




```
(* file: ruler.ml *)

let xsize = 600
let ysize = 400

let main () =
  let window = GWindow.window ~title:"Ruler" ~border_width:10 () in
  window#connect#destroy ~callback:GMain.Main.quit;

  (* Create a table for placing the ruler and the drawing area *)
  let table = GPack.table ~rows:3 ~columns:2 ~packing:window#add () in

  let area = GMisc.drawing_area ~width:xsize ~height:ysize
    ~packing:(table#attach ~left:1 ~top:1) () in
  area#event#add [‘POINTER_MOTION; ‘POINTER_MOTION_HINT];

  (* The horizontal ruler goes on the top. As the mouse moves across
   * the drawing area, a motion_notify_event is passed to the
   * approprite event handler for the ruler. *)
  let hruler = GRange.ruler ‘HORIZONTAL ~metric:‘PIXELS
    ~lower:7.0 ~upper:13.0 ~position:0.0 ~max_size:20.0
    ~packing:(table#attach ~left:1 ~top:0) () in
  area#event#connect#motion_notify
```

```
    ~callback:(fun ev -> hruler#event#send (ev :> GdkEvent.any));

  (* The vertical ruler goes on the left. As the mouse moves across
   * the drawing area, a motion_notify_event is passed to the
   * approprite event handler for the ruler. *)
  let vruler = GRange.ruler 'VERTICAL ~metric:'PIXELS
    ~lower:0.0 ~upper:(float ysize) ~position:0.0 ~max_size:(float ysize)
    ~packing:(table#attach ~left:0 ~top:1) () in
  area#event#connect#motion_notify
    ~callback:(fun ev -> vruler#event#send (ev :> GdkEvent.any));

  window#show ();
  GMain.Main.main ()

let _ = main ()
```

11.7. Statusbars

Statusbars are simple widgets used to display a text message. They keep a stack of the messages pushed onto them, so that popping the current message will re-display the previous text message.

In order to allow different parts of an application to use the same statusbar to display messages, the statusbar widget issues Context Identifiers which are used to identify different "users". The message on top of the stack is the one displayed, no matter what context it is in. Messages are stacked in last-in-first-out order, not context identifier order.

A statusbar is created with a call to `GMisc.status_bar` :

```
val GMisc.statusbar :
 ?border_width:int ->
 ?width:int ->
 ?height:int ->
 ?packing:(GObj.widget -> unit) ->
 ?show:bool -> unit -> statusbar
```

A new Context Identifier is requested using a call to the following function with a short textual description of the context:

```
method new_context : name:string -> statusbar_context
```

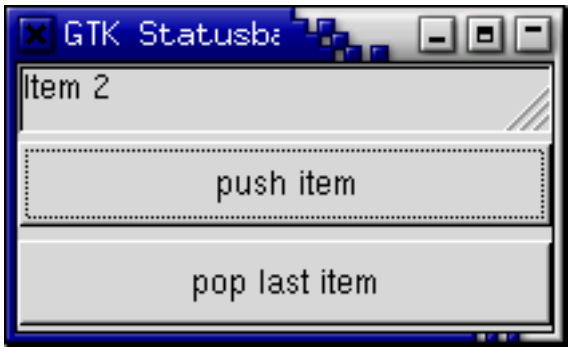
There are three functions that can operate on statusbar_contexts:

```
method push : string -> Gtk.statusbar_message
method pop : unit -> unit
method remove : Gtk.statusbar_message -> unit
```

The first, push method, is used to add a new message to the statusbar. It returns a Message Identifier, which can be passed later to the remove method to remove the message with the given Message Identifiers and Context from the statusbar's stack.

The method pop removes the message highest in the stack with the given Context.

The following example creates a statusbar and two buttons, one for pushing items onto the statusbar, and one for popping the last item back off.



```
(* file: statusbar.ml *)

let count = ref 0

let push_item context () =
  incr count;
  context#push (Printf.sprintf "item %d" !count);
  ()

let pop_item context () =
  context#pop ();
  ()

let main () =
  (* Create a new window; set title and border width *)
  let window = GWindow.window ~title:"Statusbar" () in

  (* Set a handler for destroy event that immediately exits GTK. *)
  window#connect#destroy ~callback:GMain.Main.quit;

  let vbox = GPack.vbox ~packing:window#add () in

  let statusbar = GMisc.statusbar ~packing:vbox#add () in
  let context = statusbar#new_context ~name:"Statusbar example" in

  let button = GButton.button ~label:"push item" ~packing:vbox#add () in
  button#connect#clicked ~callback:(push_item context);

  let button = GButton.button ~label:"pop last item" ~packing:vbox#add () in
  button#connect#clicked ~callback:(pop_item context);

  (* always display the window as the last step so it all splashes on
   * the screen at once. *)
  window#show ();
  GMain.Main.main ()

let _ = Printexc.print main ()
```

11.8. Text Entries

The Entry widget allows text to be typed and displayed in a single line text box. The text may be set with function calls that allow new text to replace, prepend or append the current contents of the Entry widget.

Create a new Entry widget with the function `GEdit.entry`.

```
val GEdit.entry :
 ?text:string ->
 ?visibility:bool ->
 ?max_length:int ->
 ?activates_default:bool ->
 ?editable:bool ->
 ?has_frame:bool ->
 ?width_chars:int ->
 ?width:int ->
 ?height:int ->
 ?packing:(GObj.widget -> unit) ->
 ?show:bool -> unit -> entry
```

The next method alters the text which is currently within the Entry widget.

```
method set_text : string -> unit
```

The set_text method sets the contents of the Entry widget, replacing the current contents. Note that the class Entry implements the Editable interface (yes, gobject supports Java-like interfaces) which contains some more functions for manipulating the contents.

The contents of the Entry can be retrieved by using a call to the following method. This is useful in the callback functions described below.

```
method text : string
```

If we don't want the contents of the Entry to be changed by someone typing into it, we can change its editable state.

```
method set_editable : bool -> unit
```

The method above allows us to toggle the editable state of the Entry widget by passing in a true or false value as argument.

If we are using the Entry where we don't want the text entered to be visible, for example when a password is being entered, we can use the following method, which also takes a boolean flag.

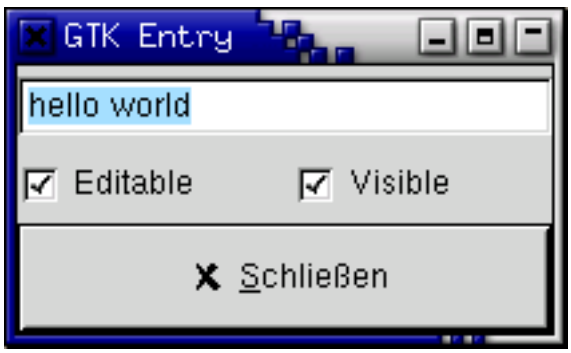
```
method set_visibility : bool -> unit
```

A region of the text may be set as selected by using the following method. This would most often be used after setting some default text in an Entry, making it easy for the user to remove it.

```
method select_region : start:int -> stop:int -> unit
```

If we want to catch when the user has entered text, we can connect to the `activate` or `changed` signal. Activate is raised when the user hits the enter key within the Entry widget. Changed is raised when the text changes at all, e.g., for every character entered or removed.

The following code is an example of using an Entry widget.




```
(* file: entry.ml *)

let enter_cb entry () =
  let text = entry#text in
  Printf.printf "Entry contents: %s\n" text;
  flush stdout

let toggle checkbutton f () = f checkbutton#active

let main () =
  (* Create a new window; set title and border width *)
  let window = GWindow.window ~title:"Entry"
    ~width:200 ~height:100 ~border_width:10 () in

  (* Set a handler for destroy event that immediately exits GTK. *)
  window#connect#destroy ~callback:GMain.Main.quit;

  let vbox = GPack.vbox ~packing:window#add () in

  let entry = GEdit.entry ~text:"hello" ~max_length:500 ~packing:vbox#add () in
  entry#connect#activate ~callback:(enter_cb entry);
  let tmp_pos = entry#text_length in
  entry#insert_text " world" tmp_pos;
  entry#select_region ~start:0 ~stop:entry#text_length;

  let hbox = GPack.hbox ~packing:vbox#add () in
  let check = GButton.check_button ~label:"Editable"
    ~active:true ~packing:hbox#add () in
  check#connect#toggled ~callback:(toggle check entry#set_editable);

  let check = GButton.check_button ~label:"Visible"
```

```
    ~active:true ~packing:hbox#add () in
  check#connect#toggled ~callback:(toggle check entry#set_visibility);

  let button = GButton.button ~stock:'CLOSE ~packing:vbox#add () in
  button#connect#clicked ~callback:window#destroy;

  button#misc#set_can_default true;
  button#misc#grab_default ();

  window#show ();
  GMain.Main.main ()

let _ = Printexc.print main ()
```

11.9. Spin Buttons

The Spin Button widget is generally used to allow the user to select a value from a range of numeric values. It consists of a text entry box with up and down arrow buttons attached to the side. Selecting one of the buttons causes the value to "spin" up and down the range of possible values. The entry box may also be edited directly to enter a specific value.

The Spin Button allows the value to have zero or a number of decimal places and to be incremented/decremented in configurable steps. The action of holding down one of the buttons optionally results in an acceleration of change in the value according to how long it is depressed.

The Spin Button uses an Adjustment object to hold information about the range of values that the spin button can take. This makes for a powerful Spin Button widget.

Recall that an adjustment widget is created with the function `GData.adjustment`, which illustrates the information that it holds:

```
val GData.adjustment :
 ?value:float ->
 ?lower:float ->
 ?upper:float ->
 ?step_incr:float ->
 ?page_incr:float ->
 ?page_size:float -> unit -> adjustment

lower : default value is 0.
upper : default value is 100.
step_incr : default value is 1.
page_incr : default value is 10.
page_size : default value is 10.
```

These attributes of an Adjustment are used by the Spin Button in the following way:

- `value`: initial value for the Spin Button
- `lower`: lower range value
- `upper`: upper range value
- `step_increment`: value to increment/decrement when pressing mouse button 1 on a button
- `page_increment`: value to increment/decrement when pressing mouse button 2 on a button
- `page_size`: unused

Additionally, mouse button 3 can be used to jump directly to the `upper` or `lower` values when used to select one of the buttons. Lets look at how to create a Spin Button using `GEdit.spin_button`:

```
val GEdit.spin_button :
 ?adjustment:GData.adjustment ->
 ?rate:float ->
 ?digits:int ->
 ?numeric:bool ->
 ?snap_to_ticks:bool ->
 ?update_policy:[ 'ALWAYS | 'IF_VALID ] ->
 ?value:float ->
 ?wrap:bool ->
```

```
 ?width:int ->
 ?height:int ->
 ?packing:(GObj.widget -> unit) ->
 ?show:bool -> unit -> spin_button
```

The `rate` argument take a value between 0.0 and 1.0 and indicates the amount of acceleration that the Spin Button has. The `digits` argument specifies the number of decimal places to which the value will be displayed.

A Spin Button can be reconfigured after creation using the following methods:

```
method set_adjustment : GData.adjustment -> unit
method set_rate : float -> unit
method set_digits : int -> unit
```

The adjustment can be retrieved using the following function:

```
method adjustment : GData.adjustment
```

The value that a Spin Button is currently displaying can be changed using the following function:

```
method set_value : float -> unit
```

The current value of a Spin Button can be retrieved as either a floating point or integer value with the following functions:

```
method value : float
method value_as_int : int
```

If you want to alter the value of a Spin Button relative to its current value, then the following function can be used:

```
method spin : Gtk.Tags.spin_type -> unit
```

The argument can take one of the following values:

```
  'STEP_FORWARD
  'STEP_BACKWARD
  'PAGE_FORWARD
  'PAGE_BACKWARD
  'HOME
  'END
  'USER_DEFINED of float
```

This function packs in quite a bit of functionality, which I will attempt to clearly explain. Many of these settings use values from the Adjustment object that is associated with a Spin Button.

`'STEP_FORWARD` and `'STEP_BACKWARD` change the value of the Spin Button by the amount specified by `increment`, unless `increment` is equal to 0, in which case the value is changed by the value of `step_increment` in theAdjustment.

`'PAGE_FORWARD` and `'PAGE_BACKWARD` simply alter the value of the Spin Button by `increment`.

`'HOME` sets the value of the Spin Button to the bottom of the Adjustments range.

`'END` sets the value of the Spin Button to the top of the Adjustments range.

`'USER_DEFINED` simply alters the value of the Spin Button by the specified amount.

We move away from functions for setting and retreving the range attributes of the Spin Button now, and move onto functions that effect the appearance and behaviour of the Spin Button widget itself.

The first of these functions is used to constrain the text box of the Spin Button such that it may only contain a numeric value. This prevents a user from typing anything other than numeric values into the text box of a Spin Button:

```
method set_numeric : bool -> unit
```

You can set whether a Spin Button will wrap around between the upper and lower range values with the following function:

```
method set_wrap : bool -> unit
```

You can set a Spin Button to round the value to the nearest `step_increment`, which is set within the Adjustment object used with the Spin Button. This is accomplished with the following function:

```
method set_snap_to_ticks : bool -> unit
```

The update policy of a Spin Button can be changed with the following function:

```
method set_update_policy : [ 'ALWAYS | 'IF_VALID ] -> unit
```

The possible values of `policy` are either `'ALWAYS` or `'IF_VALID`.

These policies affect the behavior of a Spin Button when parsing inserted text and syncing its value with the values of the Adjustment.

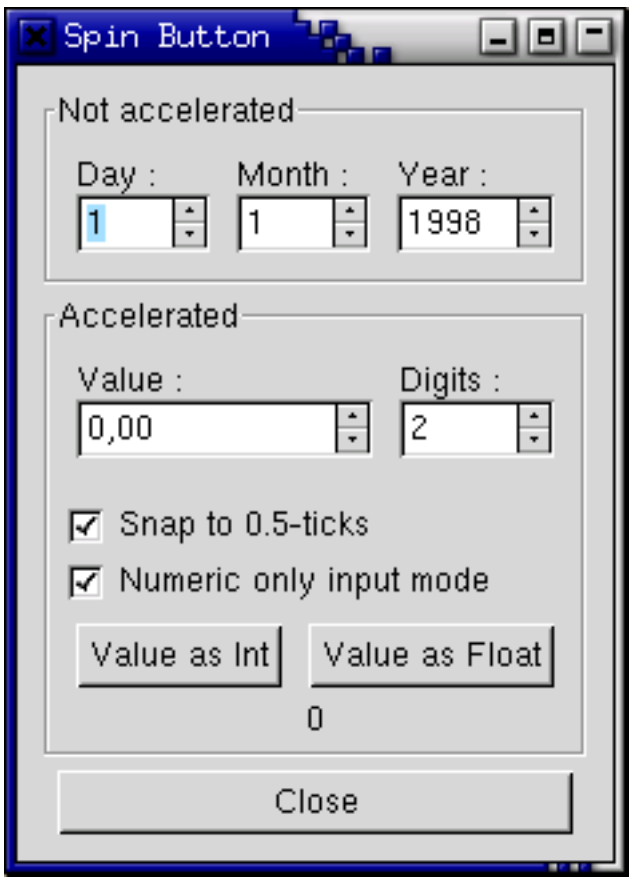
In the case of `'IF_VALID` the Spin Button only value gets changed if the text input is a numeric value that is within the range specified by the Adjustment. Otherwise the text is reset to the current value.

In case of `'ALWAYS` we ignore errors while converting text into a numeric value.

Finally, you can explicitly request that a Spin Button update itself:

```
method update : unit
```

It's example time again.




```
(* file: spinbutton.ml *)

let toggle checkbutton f () = f checkbutton#active

let get_value spinner label show_type () =
  let text =
    match show_type with
    | 'INT -> Printf.sprintf "%d" spinner#value_as_int
    | _ -> Printf.sprintf "%0.*f" spinner#digits spinner#value
  in
  label#set_text text

let main () =
```

```
  (* Create a new window; set title and border width *)
  let window = GWindow.window ~title:"Spin Button" ~border_width:10 () in

  (* Set a handler for destroy event that immediately exits GTK. *)
  window#connect#destroy ~callback:GMain.Main.quit;

  let main_vbox = GPack.vbox ~border_width:10 ~packing:window#add () in

  let frame = GBin.frame ~label:"Not accelerated" ~packing:main_vbox#add () in
  let vbox = GPack.vbox ~border_width:5 ~packing:frame#add () in

  (* Day, month, year spinners *)
  let hbox = GPack.hbox ~packing:vbox#add () in

  let vbox2 = GPack.vbox ~packing:hbox#add () in
  let label = GMisc.label ~text:"Day :" ~xalign:0.0 ~yalign:0.5 ~packing:vbox2#add () in
  let adj = GData.adjustment ~value:1.0 ~lower:1.0 ~upper:31.0 ~step_incr:1.0 ~page_incr:5.0 ~page_siz
  let spinner = GEdit.spin_button ~adjustment:adj ~rate:0.0 ~digits:0 ~wrap:true ~packing:vbox2#add (

  let vbox2 = GPack.vbox ~packing:hbox#add () in
  let label = GMisc.label ~text:"Month :" ~xalign:0.0 ~yalign:0.5 ~packing:vbox2#add () in
  let adj = GData.adjustment ~value:1.0 ~lower:1.0 ~upper:12.0 ~step_incr:1.0 ~page_incr:5.0 ~page_siz
  let spinner = GEdit.spin_button ~adjustment:adj ~rate:0.0 ~digits:0 ~wrap:true ~packing:vbox2#add (

  let vbox2 = GPack.vbox ~packing:hbox#add () in
  let label = GMisc.label ~text:"Year :" ~xalign:0.0 ~yalign:0.5 ~packing:vbox2#add () in
  let adj = GData.adjustment ~value:1998.0 ~lower:0.0 ~upper:2100.0 ~step_incr:1.0 ~page_incr:100.0 ~p
  let spinner = GEdit.spin_button ~adjustment:adj ~rate:0.0 ~digits:0 ~wrap:false ~width:55 ~packing:v

  let frame = GBin.frame ~label:"Accelerated" ~packing:main_vbox#add () in
  let vbox = GPack.vbox ~border_width:5 ~packing:frame#add () in

  let hbox = GPack.hbox ~packing:vbox#add () in

  let vbox2 = GPack.vbox ~packing:hbox#add () in
  let label = GMisc.label ~text:"Value :" ~xalign:0.0 ~yalign:0.5 ~packing:vbox2#add () in
  let adj = GData.adjustment ~value:0.0 ~lower:(-10000.0) ~upper:10000.0 ~step_incr:0.5 ~page_incr:100
  let spinner1 = GEdit.spin_button ~adjustment:adj ~rate:1.0 ~digits:2 ~width:100 ~packing:vbox2#add (

  let vbox2 = GPack.vbox ~packing:hbox#add () in
  let label = GMisc.label ~text:"Digits :" ~xalign:0.0 ~yalign:0.5 ~packing:vbox2#add () in
  let adj = GData.adjustment ~value:2.0 ~lower:1.0 ~upper:5.0 ~step_incr:1.0 ~page_incr:1.0 ~page_size
  let spinner2 = GEdit.spin_button ~adjustment:adj ~rate:0.0 ~digits:0 ~packing:vbox2#add () in
  adj#connect#value_changed ~callback:(fun () -> spinner1#set_digits spinner2#value_as_int);

  let button = GButton.check_button ~label:"Snap to 0.5-ticks" ~packing:vbox#add () in
  button#connect#clicked ~callback:(toggle button spinner1#set_snap_to_ticks);
  let button = GButton.check_button ~label:"Numeric only input mode" ~active:true ~packing:vbox#add (
  button#connect#clicked ~callback:(toggle button spinner1#set_numeric);

  let hbox = GPack.hbox ~packing:vbox#add () in
  let val_label = GMisc.label ~text:"0" ~packing:vbox#add () in

  let button = GButton.button ~label:"Value as Int" ~packing:hbox#add () in
  button#connect#clicked ~callback:(get_value spinner1 val_label 'INT);
  let button = GButton.button ~label:"Value as Float" ~packing:hbox#add () in
  button#connect#clicked ~callback:(get_value spinner1 val_label 'FLOAT);

  let hbox = GPack.hbox ~packing:main_vbox#add () in
  let button = GButton.button ~label:"Close" ~packing:hbox#add () in
  button#connect#clicked ~callback:window#destroy;

  window#show ();

  (* Enter the event loop *)
  GMain.Main.main ()

let _ = Printexc.print main ()
```

11.10. Combo Box

The combo box is another fairly simple widget that is really just a collection of other widgets. From the user's point of view, the widget consists of a text entry box and a pull down menu from which the user can select one of a set of predefined entries. Alternatively, the user can type a different option directly into the text box.

The following extract child widgets from a Combo Box:

```
method entry : entry
method list : GList.liste
```

As you can see, the Combo Box has two principal parts that you really care about: an entry and a list.

First off, to create a combo box, use `GEdit.combo`:

```
val GEdit.combo :
 ?popdown_strings:string list ->
 ?allow_empty:bool ->
 ?case_sensitive:bool ->
 ?enable_arrow_keys:bool ->
 ?value_in_list:bool ->
 ?border_width:int ->
 ?width:int ->
 ?height:int ->
 ?packing:(GObj.widget -> unit) ->
 ?show:bool -> unit -> combo
```

Now, if you want to set the string in the entry section of the combo box, this is done by manipulating the `entry` widget directly:

```
combo#entry#set_text "My String";
```

To set the values in the popdown list, one uses the function:

```
method set_popdown_strings : string list -> unit
```

At this point you have a working combo box that has been set up. There are a few aspects of its behavior that you can change. These are accomplished with the functions:

```
method set_enable_arrow_keys : bool -> unit
method set_case_sensitive : bool -> unit
```

set_enable_arrow_keys method lets the user change the value in the entry using the up/down arrow keys. This doesn't bring up the list, but rather replaces the current text in the entry with the next list entry (up or down, as your key choice indicates). It does this by searching in the list for the item corresponding to the current value in the entry and selecting the previous/next item accordingly. Usually in an entry the arrow keys are used to change focus (you can do that anyway using TAB). Note that when the current item is the last of the list and you press arrow-down it changes the focus (the same applies with the first item and arrow-up).

If the current value in the entry is not in the list, then the function of set_enable_arrow_keys is disabled.

set_case_sensitive method toggles whether or not GTK searches for entries in a case sensitive manner. This is used when the Combo widget is asked to find a value from the list using the current entry in the text box. This completion can be performed in either a case sensitive or insensitive manner, depending upon the use of this function. The Combo widget can also simply complete the current entry if the user presses the key combination MOD-1 and "Tab". MOD-1 is often mapped to the "Alt" key, by the `xmodmap` utility. Note, however that some window managers also use this key combination, which will override its use within GTK.

Now that we have a combo box, tailored to look and act how we want it, all that remains is being able to get data from the combo box. This is relatively straightforward. The majority of the time, all you are going to care about getting data from is the entry. The entry is accessed simply by `combo#entry`. The two principal things that you are going to want to do with it are connect to the activate signal, which indicates that the user has pressed the Return or Enter key, and read the text. The first is accomplished using something like:

```
combo#entry#connect#activate ~callback:my_callback;
```

Getting the text at any arbitrary time is accomplished by simply using the entry function such as:

```
let text = combo#entry#text in
...
```

That's about all there is to it. There is a function

```
method disable_activate : unit -> unit
```

that will disable the activate signal on the entry widget in the combo box. Personally, I can't think of why you'd want to use it, but it does exist.

11.11. Calendar

The Calendar widget is an effective way to display and retrieve monthly date related information. It is a very simple widget to create and work with.

Creating a GtkCalendar widget is a simple as: (see `GMisc.calendar`)

```
val GMisc.calendar :
 ?options:Gtk.Tags.calendar_display_options list ->
 ?packing:(GObj.widget -> unit) ->
 ?show:bool -> unit -> calendar
```

There might be times where you need to change a lot of information within this widget and the following functions allow you to make multiple change to a Calendar widget without the user seeing multiple on-screen updates.

```
method freeze : unit -> unit
method thaw : unit -> unit
```

They work just like the freeze/thaw functions of every other widget.

The Calendar widget has a few options that allow you to change the way the widget both looks and operates by using the function

```
method display_options : Gtk.Tags.calendar_display_options list -> unit
```

The `flags` argument can be formed by combining either of the following five options:

`'SHOW_HEADING`

this option specifies that the month and year should be shown when drawing the calendar.

`'SHOW_DAY_NAMES`

this option specifies that the three letter descriptions should be displayed for each day (eg Mon,Tue, etc.).

`'NO_MONTH_CHANGE`

this option states that the user should not and can not change the currently displayed month. This can be good if you only need to display a particular month such as if you are displaying 12 calendar widgets for every month in a particular year.

`'SHOW_WEEK_NUMBERS`

this option specifies that the number for each week should be displayed down the left side of the calendar. (eg. Jan 1 = Week 1,Dec 31 = Week 52).

`'WEEK_START_MONDAY`

this option states that the calander week will start on Monday instead of Sunday which is the default. This only affects the order in which days are displayed from left to right.

The following functions are used to set the the currently displayed date:

```
method select_month : month:int -> year:int -> unit
method select_day : int -> unit
```

With `select_day()` the specified day number is selected within the current month, if that is possible. A `day` value of 0 will deselect any current selection.

In addition to having a day selected, any number of days in the month may be "marked". A marked day is highlighted within the calendar display. The following functions are provided to manipulate marked days:

```
method mark_day : int -> unit
```

```
method unmark_day : int -> unit
method clear_marks : unit
```

For example:

```
let calendar = GMisc.calendar in

...

(* Is day 7 marked? *)
if calendar#mark_day 7 then (
    (* day is marked *)
) else (
    ...
)
```

Note that marks are persistent across month and year changes.

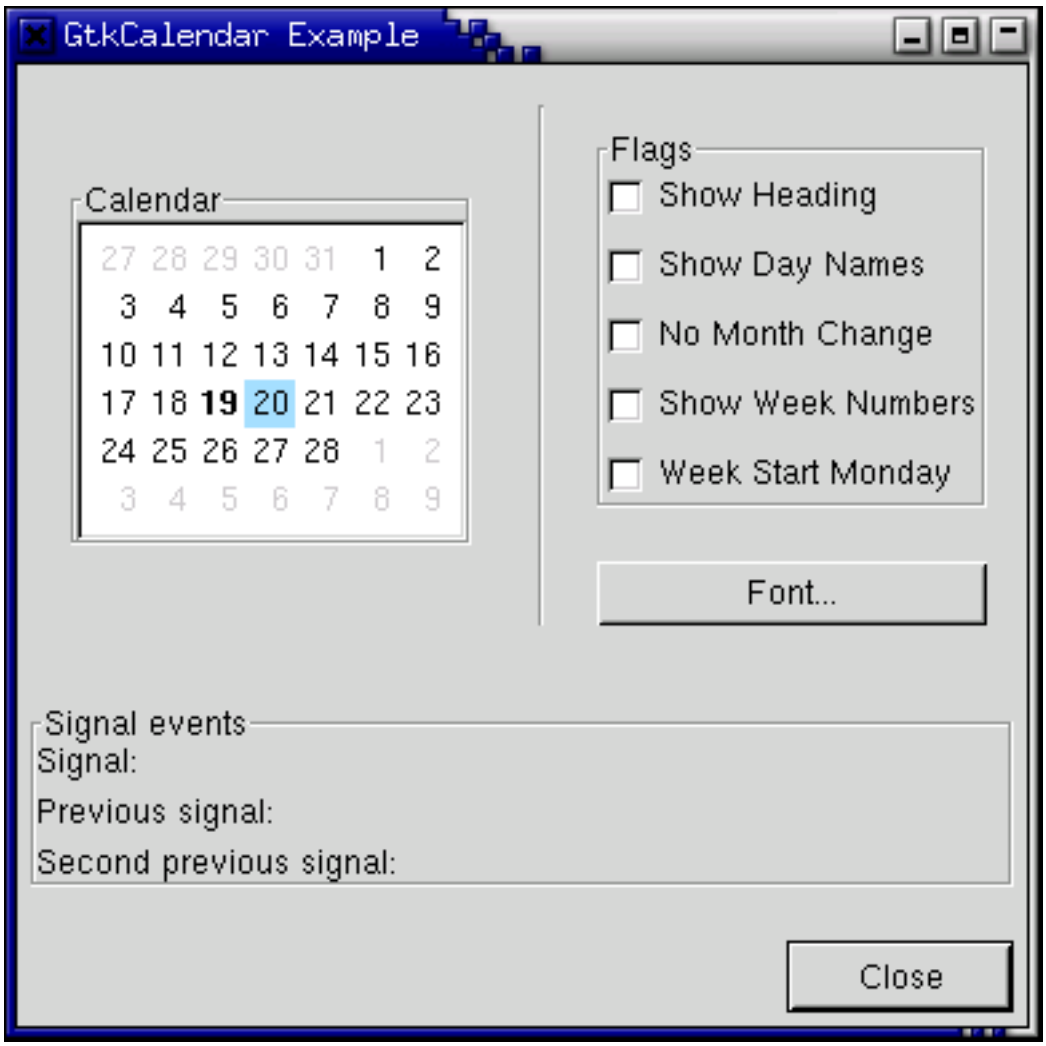
The final Calendar widget function is used to retrieve the currently selected date, month and/or year.

```
method date : int * int * int
```

The Calendar widget can generate a number of signals indicating date selection and change. The names of these signals are self explanatory, and are; see `GMisc.calendar_signals`:

- `month_changed`
- `day_selected`
- `day_selected_double_click`
- `prev_month`
- `next_month`
- `prev_year`
- `next_year`

That just leaves us with the need to put all of this together into example code.




```
(* file: calendar.ml *)

type signals =
  { mutable last_sig: GMisc.label;
    mutable prev_sig: GMisc.label;
    mutable prev2_sig: GMisc.label
  }

let signals =
  let label = GMisc.label () in
  { last_sig = label;
    prev_sig = label;
    prev2_sig = label }

let set_signal_strings string =
  signals.prev2_sig#set_text signals.prev_sig#text;
  signals.prev_sig#set_text signals.last_sig#text;
  signals.last_sig#set_text string

let show_signal calendar msg () =
  let (y, m, d) = calendar#date in
  let str = Printf.sprintf "%s: %d/%d/%d" msg y m d in
  set_signal_strings str

let toggle_flags calendar but_flags () =
  let rec loop bflags =
    match bflags with
    | [] -> []
    | (but, flag)::rest ->
      if but#active
      then flag :: loop rest
```

```
      else loop rest
  in
  let flags = loop but_flags in
  calendar#display_options flags

let font_selection_ok font_window calendar () =
  let font_name = font_window#selection#font_name in
  let font_desc = GPango.font_description font_name in
  calendar#misc#modify_font font_desc

let select_font calendar () =
  let fwin = GWindow.font_selection_dialog ~title:"Font Selection Dialog" ~modal:true ~position:‘MOUSE
  fwin#connect#destroy ~callback:fwin#destroy;
  fwin#ok_button#connect#clicked ~callback:(font_selection_ok fwin calendar);
  fwin#cancel_button#connect#clicked ~callback:fwin#destroy;
  fwin#show ()

let flags = [("Show Heading", true, ‘SHOW_HEADING);
    ("Show Day Names", true, ‘SHOW_DAY_NAMES);
    ("No Month Change", false, ‘NO_MONTH_CHANGE);
    ("Show Week Numbers", false, ‘SHOW_WEEK_NUMBERS);
    ("Week Start Monday", false, ‘WEEK_START_MONDAY)]

let create_calendar () =
  (* Create a new window; set title and border width *)
  let window = GWindow.window ~title:"Calendar Example" ~resizable:false ~border_width:10 () in

  (* Set a handler for destroy event that immediately exits GTK. *)
  window#connect#destroy ~callback:GMain.Main.quit;

  let vbox = GPack.vbox ~border_width:10 ~packing:window#add () in

  (* The top part of the window, Calendar, flags and fontsel. *)
  let hbox = GPack.hbox ~packing:vbox#add () in
  let hbbox = GPack.button_box ‘HORIZONTAL ~layout:‘SPREAD ~spacing:5 ~packing:hbox#add () in

  (* Calendar widget *)
  let frame = GBin.frame ~packing:hbbox#add () in
  let calendar = GMisc.calendar ~packing:frame#add () in
  calendar#mark_day 19;
  calendar#connect#month_changed ~callback:(show_signal calendar "month_changed");
  calendar#connect#day_selected ~callback:(show_signal calendar "day_selected");
  calendar#connect#day_selected_double_click ~callback:(show_signal calendar "day_selected_double_clic
  calendar#connect#prev_month ~callback:(show_signal calendar "prev_month");
  calendar#connect#next_month ~callback:(show_signal calendar "next_month");
  calendar#connect#prev_year ~callback:(show_signal calendar "prev_year");
  calendar#connect#next_year ~callback:(show_signal calendar "next_year");

  let separator = GMisc.separator ‘VERTICAL ~packing:hbox#add () in

  let vbox2 = GPack.vbox ~packing:hbox#add () in

  (* Build the Right frame with the flags in *)
  let frame = GBin.frame ~label:"Flags" ~packing:vbox2#add () in
  let vbox3 = GPack.vbox ~packing:frame#add () in
  let toggle_button (label, active, flag) =
    (GButton.check_button ~label ~active ~packing:vbox3#add (), flag) in
  let flag_buttons = List.map toggle_button flags in
  let set_flag_cb (but, _) =
    but#connect#toggled ~callback:(toggle_flags calendar flag_buttons);
    ()
  in
  List.iter set_flag_cb flag_buttons;

  (* Build the right font-button *)
  let button = GButton.button ~label:"Font..." ~packing:vbox2#add () in
  button#connect#clicked ~callback:(select_font calendar);

  (* Build the Signal-event part. *)
  let frame = GBin.frame ~label:"Signal events" ~packing:vbox#add () in
  let vbox2 = GPack.vbox ~packing:frame#add () in
```

```
  let hbox = GPack.hbox ~packing:vbox2#add () in
  let label = GMisc.label ~text:"Signal: " ~packing:hbox#add () in
  signals.last_sig <- GMisc.label ~packing:hbox#add ();

  let hbox = GPack.hbox ~packing:vbox2#add () in
  let label = GMisc.label ~text:"Previous signal: " ~packing:hbox#add () in
  signals.prev_sig <- GMisc.label ~packing:hbox#add ();

  let hbox = GPack.hbox ~packing:vbox2#add () in
  let label = GMisc.label ~text:"Second previous signal: " ~packing:hbox#add () in
  signals.prev2_sig <- GMisc.label ~packing:hbox#add ();

  let bbox = GPack.button_box `HORIZONTAL ~layout:`END ~packing:vbox#add () in
  let button = GButton.button ~label:"Close" ~packing:bbox#add () in
  button#connect#clicked ~callback:GMain.Main.quit;

  button#misc#set_can_default true;
  button#misc#grab_default ();

  window#show ()

let main () =
  create_calendar ();

  (* Enter the event loop *)
  GMain.Main.main ()

let _ = Printexc.print main ()
```

11.12. Color Selection

The color selection widget is, not surprisingly, a widget for interactive selection of colors. This composite widget lets the user select a color by manipulating RGB (Red, Green, Blue) and HSV (Hue, Saturation, Value) triples. This is done either by adjusting single values with sliders or entries, or by picking the desired color from a hue-saturation wheel/value bar. Optionally, the opacity of the color can also be set.

Lets have a look at what the color selection widget has to offer us. The widget comes in two flavours: `GMisc.color_selection` and `GWindow.color_selection_dialog`.

```
val GMisc.color_selection :
 ?alpha:int ->
 ?color:Gdk.color ->
 ?has_opacity_control:bool ->
 ?has_palette:bool ->
 ?border_width:int ->
 ?width:int ->
 ?height:int ->
 ?packing:(GObj.widget -> unit) ->
 ?show:bool -> unit -> color_selection
```

You'll probably not be using this constructor directly. It creates an orphan color_selection widget which you'll have to parent yourself. The color_selection widget inherits from the VBox widget.

```
val GWindow.color_selection_dialog :
 ?title:string ->
 ?parent:#window_skel ->
 ?destroy_with_parent:bool ->
 ?allow_grow:bool ->
 ?allow_shrink:bool ->
 ?icon:GdkPixbuf.pixbuf ->
 ?modal:bool ->
 ?screen:Gdk.screen ->
 ?type_hint:Gdk.Tags.window_type_hint ->
 ?position:Gtk.Tags.window_position ->
 ?wm_name:string ->
 ?wm_class:string ->
 ?border_width:int ->
 ?width:int ->
```

```
 ?height:int ->
 ?show:bool -> unit -> color_selection_dialog
```

This is the most common color selection constructor. It creates a color_selection_dialog. It consists of a Frame containing a color_selection widget, an HSeparator and an HBox with three buttons, "Ok", "Cancel" and "Help". You can reach these buttons by accessing the "ok_button", "cancel_button" and "help_button" methods in the color_selection_dialog object, (i.e., `color_sel_dialog#ok_button`).

```
method set_has_opacity_control : bool -> unit
```

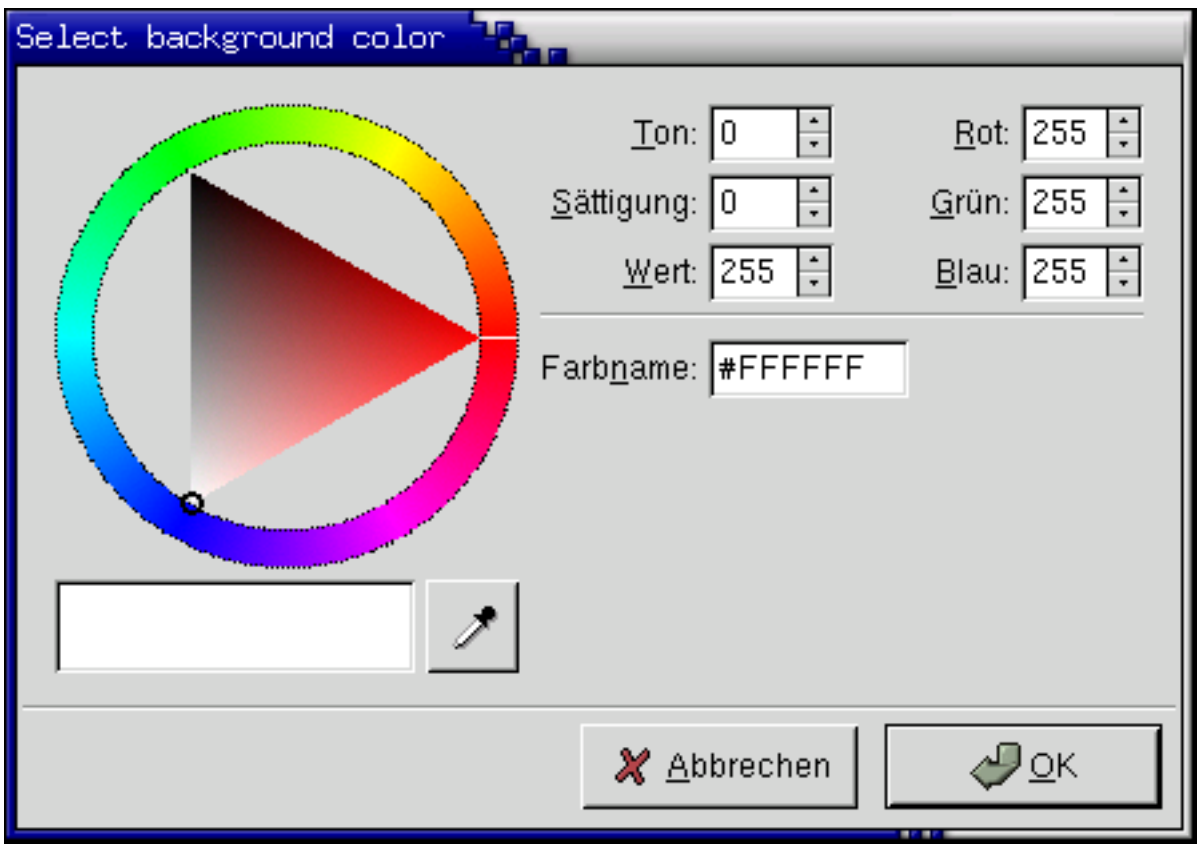
The color selection widget supports adjusting the opacity of a color (also known as the alpha channel). This is disabled by default. Calling this function with has_opacity set to true enables opacity. Likewise, has_opacity set to false will disable opacity.

```
method set_color : Gdk.color -> unit
method set_alpha : int -> unit
```

You can set the current color explicitly by calling set_color method with a GdkColor. Setting the opacity (alpha channel) is done with set_alpha method. The alpha value should be between 0 (fully transparent) and 65636 (fully opaque).

```
method color : Gdk.color
method alpha : int
```

Here's a simple example demonstrating the use of the color_selection_dialog. The program displays a window containing a drawing area. Clicking on it opens a color selection dialog, and changing the color in the color selection dialog changes the background color.




```
(* file: colorsel.ml *)

let dialog_ref = ref None
let color = ref (`RGB (0, 65535, 0))     (* GDraw.color ref type *)

(* "color_changed" event does not exist in lablgtk2!!! *)
(*
let color_changed_cb colorsel drawingarea () =
  let ncolor = colorsel#color in
```

```
  drawingarea#misc#modify_bg [(‘NORMAL, ‘COLOR ncolor)]
*)

let response dlg drawingarea resp =
  let colorsel = dlg#colorsel in
  begin
  match resp with
  | ‘OK -> color := ‘COLOR colorsel#color
  | _ -> ()
  end;
  drawingarea#misc#modify_bg [(‘NORMAL, !color)];
  dlg#misc#hide ()

(* Drawingarea button_press event handler *)
let button_pressed drawingarea ev =
  (* Create color selection dialog *)
  let colordlg =
    match !dialog_ref with
    | None ->
      let dlg = GWindow.color_selection_dialog ~title:"Select background color" () in
      dialog_ref := Some dlg;
      dlg
    | Some dlg -> dlg
  in

  (* Get the ColorSelection widget *)
  let colorsel = colordlg#colorsel in

  (* set_prev_color does not exist in lablgtk2!!! *)
  (* colorsel#set_prev_color (GDraw.color !color); *)
  colorsel#set_color (GDraw.color !color); (* requires Gdk.color type *)
  colorsel#set_has_palette true;

  (* Connect to the "color_changed" signal *)
  (* This event does not exist in lablgtk2!!! *)
  (* Need confirm to lablgtk2 team. *)
  (* colorsel#connect#color_changed ~callback:(color_changed_cb colorsel drawingarea);
  *)

  colordlg#connect#response ~callback:(response colordlg drawingarea);

  (* Show the dialog *)
  colordlg#run ();
  true

let main () =
  (* Create toplevel window, set title and policies (allow_grow, allow_shrink) *)
  let window = GWindow.window ~title:"Color selection test" ~border_width:10
    ~allow_grow:true ~allow_shrink:true () in

  (* Attach "destroy" events so we can exit *)
  window#connect#destroy ~callback:GMain.Main.quit;

  (* Create drawingarea, set size and catch button events *)
  let drawingarea = GMisc.drawing_area ~width:200 ~height:200 ~packing:window#add () in
  drawingarea#misc#modify_bg [(‘NORMAL, !color)];
  drawingarea#event#add [‘BUTTON_PRESS];
  drawingarea#event#connect#button_press ~callback:(button_pressed drawingarea);

  window#show ();
  GMain.Main.main ()

let _ = Printexc.print main ()
```

11.13. File Selections

The file selection widget is a quick and simple way to display a File dialog box. It comes complete with Ok, Cancel, and Help buttons, a great way to cut down on programming time.

To create a new file selection box use `GWindow.file_selection`:

```
val GWindow.file_selection :
 ?title:string ->
 ?show_fileops:bool ->
 ?filename:string ->
 ?select_multiple:bool ->
 ?parent:#window_skel ->
 ?destroy_with_parent:bool ->
 ?allow_grow:bool ->
 ?allow_shrink:bool ->
 ?icon:GdkPixbuf.pixbuf ->
 ?modal:bool ->
 ?resizable:bool ->
 ?screen:Gdk.screen ->
 ?type_hint:Gdk.Tags.window_type_hint ->
 ?position:Gtk.Tags.window_position ->
 ?wm_name:string ->
 ?wm_class:string ->
 ?border_width:int ->
 ?width:int ->
 ?height:int ->
 ?show:bool -> unit -> file_selection
```

To set the filename, for example to bring up a specific directory, or give a default filename, use `filename` argument or this function:

```
method set_filename : string -> unit
```

To grab the text that the user has entered or clicked on, use this function:

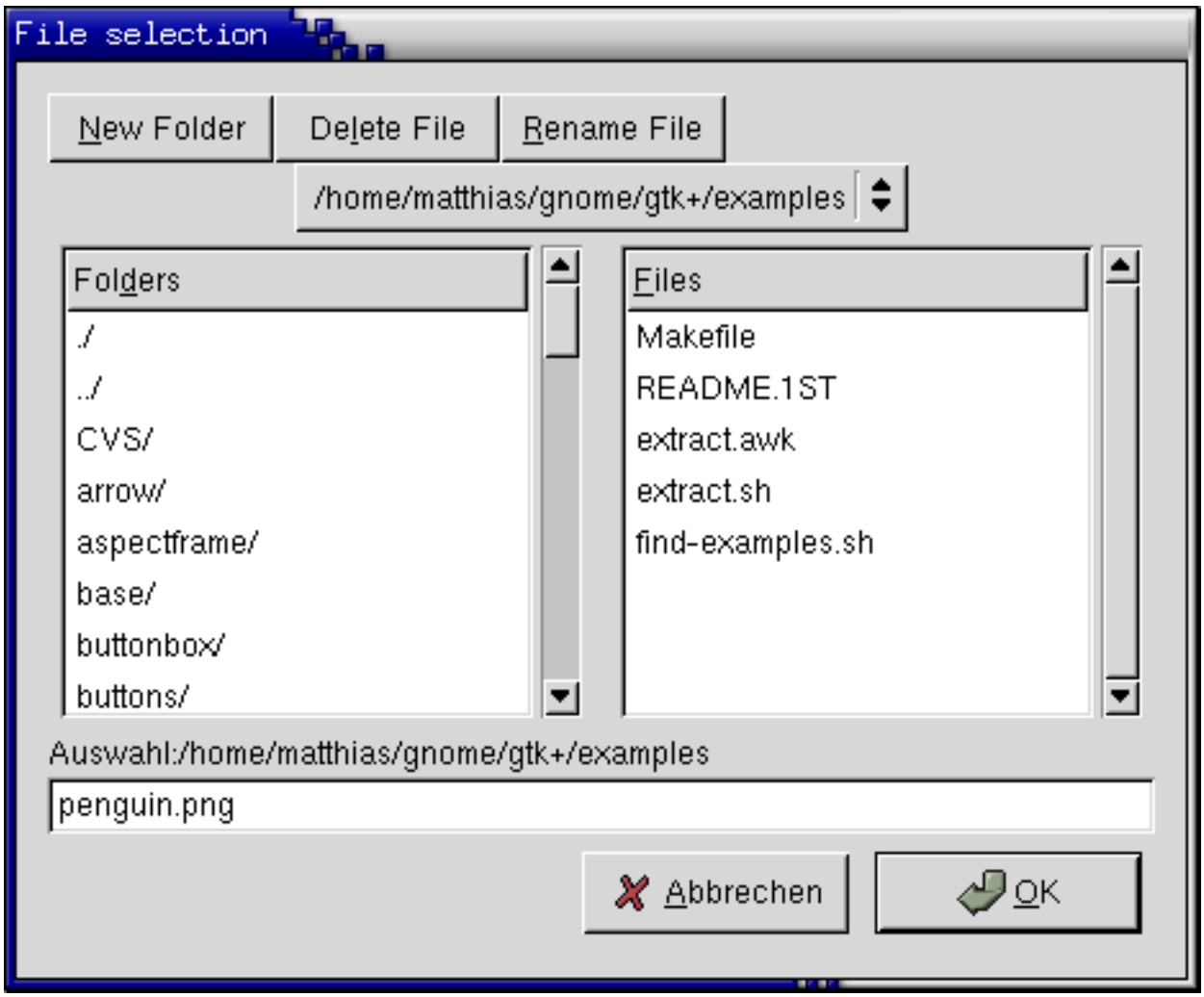
```
method filename : string
```

There are also pointers to the widgets contained within the file selection widget. These are:

```
method dir_list : string GList.clist
method file_list : string GList.clist
method get_selections : string list
method ok_button : GButton.button
method cancel_button : GButton.button
method help_button : GButton.button
```

Most likely you will want to use the ok_button, cancel_button, and help_button methods in signaling their use.

Included example is nothing much to creating a file selection widget. While in this example the Help button appears on the screen, it does nothing as there is not a signal attached to it.




```
(* file: filesel.ml *)

(* Get the selected filename and print it to the console *)
let file_ok_sel filew () =
  print_endline filew#filename;
  flush stdout

let main () =
  (* Create a new file selection widget; set default filename *)
  let filew = GWindow.file_selection ~title:"File selection" ~border_width:10
    ~filename:"penguin.png" () in

  (* Set a handler for destroy event that immediately exits GTK. *)
  filew#connect#destroy ~callback:GMain.Main.quit;

  (* Connect the ok_button to file_ok_sel function *)
  filew#ok_button#connect#clicked ~callback:(file_ok_sel filew);

  (* Connect the cancel_button to destroy the widget *)
  filew#cancel_button#connect#clicked ~callback:filew#destroy;

  filew#show ();
  (* Rest in main and wait for the fun to begin! *)
  GMain.Main.main ()

let _ = Printexc.print main ()
```

Chapter 12. Container Widgets

12.1. The EventBox

Some GTK widgets don't have associated X windows, so they just draw on their parents. Because of this, they cannot receive events and if they are incorrectly sized, they don't clip so you can get messy overwriting, etc. If you require more from these widgets, the EventBox is for you.

At first glance, the EventBox widget might appear to be totally useless. It draws nothing on the screen and responds to no events. However, it does serve a function - it provides an X window for its child widget. This is important as many GTK widgets do not have an associated X window. Not having an X window saves memory and improves performance, but also has some drawbacks. A widget without an X window cannot receive events, and does not perform any clipping on its contents. Although the name *EventBox* emphasizes the event-handling function, the widget can also be used for clipping. (and more, see the example below).

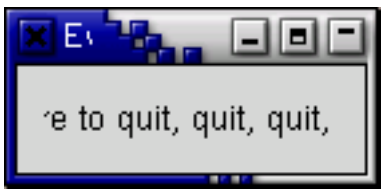
To create a new EventBox widget, use `GBin.event_box`:

```
val GBin.event_box :
 ?border_width:int ->
 ?width:int ->
 ?height:int ->
 ?packing:(GObj.widget -> unit) ->
 ?show:bool -> unit -> event_box
```

A child widget can then be added to this EventBox:

```
method add : GObj.widget -> unit
```

The following example demonstrates both uses of an EventBox - a label is created that is clipped to a small box, and set up so that a mouse-click on the label causes the program to exit. Resizing the window reveals varying amounts of the label.




```
(* file: event_box.ml *)

let main () =
  (* Create a new window; set title and border width *)
  let window = GWindow.window ~title:"Event Box" ~border_width:10 () in

  (* Set a handler for destroy event that immediately exits GTK. *)
  window#connect#destroy ~callback:GMain.Main.quit;

  (* Create an EventBox and add it to our toplevel window *)
  let event_box = GBin.event_box ~packing:window#add () in

  (* Create a long label *)
  let label = GMisc.label ~text:"Click here to quit, quit, quit, quit, quit"
    ~packing:event_box#add ()
  in

  (* Clip it short. *)
  label#misc#set_size_request ~width:110 ~height:20;

  (* And bind an action to it *)
  event_box#event#add [‘BUTTON_PRESS];
  event_box#event#connect#button_press ~callback:(fun ev -> exit 0; true);

  (* Yet one more thing you need an X window for ... *)
  event_box#misc#realize ();
  Gdk.Window.set_cursor event_box#misc#window (Gdk.Cursor.create ‘HAND1);

  window#show ();
```

```
  GMain.Main.main ()

let _ = Printexc.print main ()
```

12.2. The Alignment widget

The alignment widget allows you to place a widget within its window at a position and size relative to the size of the Alignment widget itself. For example, it can be very useful for centering a widget within the window.

`GBin.alignment` is the function associated with the Alignment widget:

```
val GBin.alignment :
 ?xalign:Gtk.clampf ->
 ?yalign:Gtk.clampf ->
 ?xscale:Gtk.clampf ->
 ?yscale:Gtk.clampf ->
 ?border_width:int ->
 ?width:int ->
 ?height:int ->
 ?packing:(GObj.widget -> unit) ->
 ?show:bool -> unit -> alignment
```

This function creates a new Alignment widget with the specified parameters.

The parameters of `Gtk.clampf` type are floating point numbers which can range from 0.0 to 1.0. The `xalign` and `yalign` arguments affect the position of the widget placed within the Alignment widget. The `xscale` and `yscale` arguments effect the amount of space allocated to the widget.

A child widget can be added to this Alignment widget using:

```
method add : GObj.widget -> unit
```

For an example of using an Alignment widget, refer to the example for the Progress Bar widget.

12.3. Fixed Container

The Fixed container allows you to place widgets at a fixed position within it's window, relative to it's upper left hand corner. The position of the widgets can be changed dynamically.

`GPack.fixed` is the function associated with the fixed widget:

```
val GPack.fixed :
 ?has_window:bool ->
 ?border_width:int ->
 ?width:int ->
 ?height:int ->
 ?packing:(GObj.widget -> unit) ->
 ?show:bool -> unit -> fixed

method put : GObj.widget -> x:int -> y:int -> unit
method move : GObj.widget -> x:int -> y:int -> unit
```

The function GPack.fixed allows you to create a new Fixed container.

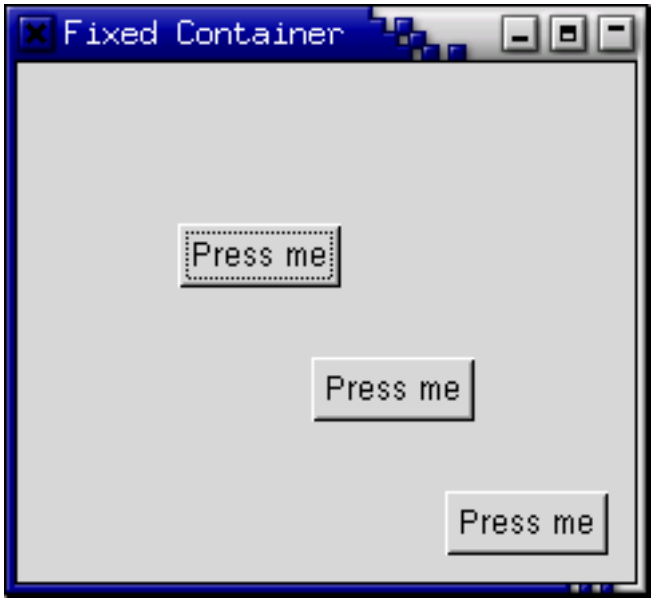
put method places `widget` in the container `fixed` at the position specified by `x` and `y`.

move method allows the specified widget to be moved to a new position.

```
method set_has_window : bool -> unit
method has_window : bool
```

Normally, Fixed widgets don't have their own X window. Since this is different from the behaviour of Fixed widgets in earlier releases of GTK, the set_has_window method allows the creation of Fixed widgets *with* their own window. It has to be called before realizing the widget.

The following example illustrates how to use the Fixed Container.




```
(* file: fixed.ml *)

(* Global variables to store the position of the widget
 * within the fixed container *)
let rx = ref 50
let ry = ref 50

(* This callback function moves the button to a new position
 * in the Fixed container. *)
let move_button but fixed () =
  rx := (!rx + 30) mod 300;
  ry := (!ry + 50) mod 300;
  fixed#move but#coerce ~x:!rx ~y:!ry

let main () =
  (* Create a new window; set title and border width *)
  let window = GWindow.window ~title:"Fixed Container" ~border_width:10 () in

  (* Here we connect the "destroy" event to a signal handler *)
  window#connect#destroy ~callback:GMain.Main.quit;

  (* Create a Fixed Container *)
  let fixed = GPack.fixed ~packing:window#add () in

  for i = 1 to 3 do
    (* Creates a new button with the label "Press me"
     * and packs the button into the fixed containers window. *)
    let button = GButton.button ~label:"Press me"
   ~packing:(fixed#put ~x:(i*50) ~y:(i*50)) ()
    in

    (* When the button receives the "clicked" signal, it will call the
     * function move_button passing it the Fixed Container as its
     * argument. *)
    button#connect#clicked ~callback:(move_button button fixed)
  done;

  (* Display the window and enter the event loop *)
  window#show ();
  GMain.Main.main ()

let _ = Printexc.print main ()
```

12.4. Layout Container

The Layout container is similar to the Fixed container except that it implements an infinite (where infinity is less than 2^32) scrolling area. The X window system has a limitation where windows can be at most 32767 pixels wide or tall. The Layout container gets around this limitation by doing some exotic stuff using window and bit gravities, so that you can have smooth scrolling even when you have many child widgets in your scrolling area.

A Layout container is created using `GPack.layout`:

```
val GPack.layout :
 ?hadjustment:GData.adjustment ->
 ?vadjustment:GData.adjustment ->
 ?layout_width:int ->
 ?layout_height:int ->
 ?border_width:int ->
 ?width:int ->
 ?height:int ->
 ?packing:(GObj.widget -> unit) ->
 ?show:bool -> unit -> layout
```

As you can see, you can optionally specify the Adjustment objects that the Layout widget will use for its scrolling.

You can add and move widgets in the Layout container using the following two functions:

```
method move : GObj.widget -> x:int -> y:int -> unit
method put : GObj.widget -> x:int -> y:int -> unit
```

The size of the Layout container can be set using the next functions:

```
method set_height : int -> unit
method set_width : int -> unit
```

The final four functions for use with Layout widgets are for manipulating the horizontal and vertical adjustment widgets:

```
method hadjustment : GData.adjustment
method vadjustment : GData.adjustment
method set_hadjustment : GData.adjustment -> unit
method set_vadjustment : GData.adjustment -> unit
```

12.5. Frames

Frames can be used to enclose one or a group of widgets with a box which can optionally be labelled. The position of the label and the style of the box can be altered to suit.

A Frame can be created with the function `GBin.frame` :

```
val frame :
 ?label:string ->
 ?label_xalign:Gtk.clampf ->
 ?label_yalign:Gtk.clampf ->
 ?shadow_type:Gtk.Tags.shadow_type ->
 ?border_width:int ->
 ?width:int ->
 ?height:int ->
 ?packing:(GObj.widget -> unit) ->
 ?show:bool -> unit -> frame
```

The label is by default placed in the upper left hand corner of the frame. A value of NULL for the `label` argument will result in no label being displayed. The text of the label can be changed using the next function.

```
method set_label : string option -> unit
```

The position of the label can be changed using these functions:

```
method set_label_xalign : float -> unit
method set_label_yalign : float -> unit
```

`xalign` and `yalign` take values between 0.0 and 1.0. `xalign` indicates the position of the label along the top horizontal of the frame. `yalign` is not currently used. The default value of xalign is 0.0 which places the label at the left hand end of the frame.

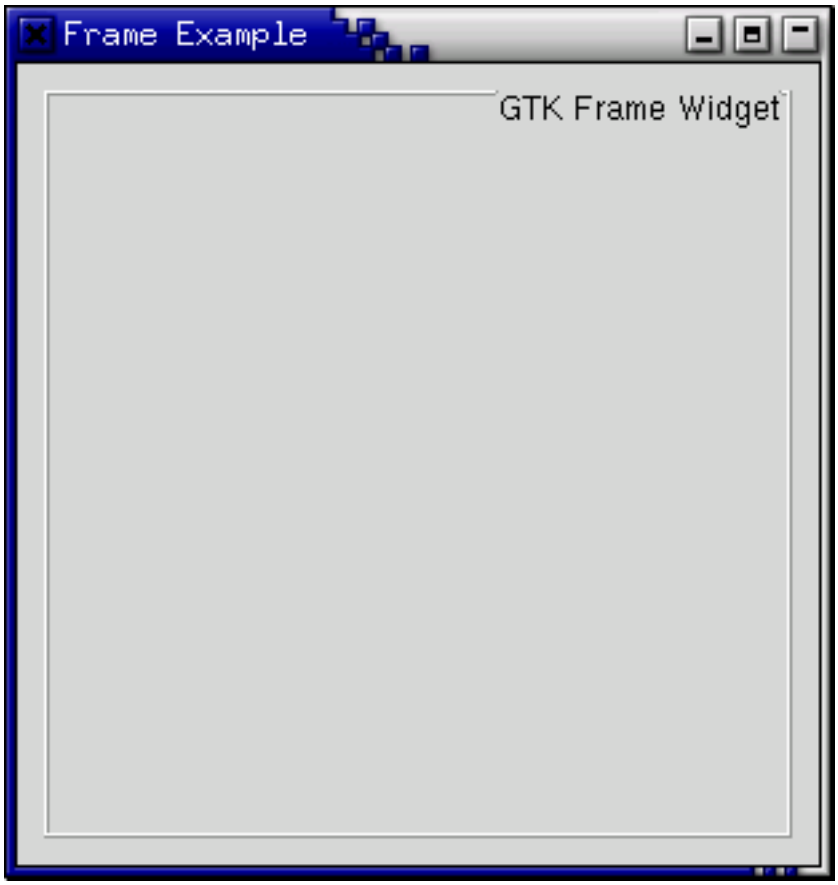
The next function alters the style of the box that is used to outline the frame.

```
method set_shadow_type : Gtk.Tags.shadow_type -> unit
```

The `type` argument can take one of the following values:

```
  ‘NONE
  ‘IN
  ‘OUT
  ‘ETCHED_IN (the default)
  ‘ETCHED_OUT
```

The following code example illustrates the use of the Frame widget.




```
(* file: frame.ml *)

let main () =
  (* Create a new window; set title and border width *)
  let window = GWindow.window ~title:"Frame Example" ~width:300 ~height:300 ~border_width:10 () in

  (* Here we connect the "destroy" event to a signal handler *)
  window#connect#destroy ~callback:GMain.Main.quit;

  (* Create a Frame
   * Set the frame’s label
   * Align the label at the right of the frame
   * Set the style of the frame *)
  let frame = GBin.frame ~label:"Frame Widget" ~label_xalign:1.0 ~shadow_type:‘ETCHED_OUT ~packing:wi

  window#show ();
  GMain.Main.main ()

let _ = Printexc.print main ()
```

12.6. Aspect Frames

The aspect frame widget is like a frame widget, except that it also enforces the aspect ratio (that is, the ratio of the width to the height) of the child widget to have a certain value, adding extra space if necessary. This is useful, for instance, if you want to preview a larger image. The size of the preview should vary when the user resizes the window, but the aspect ratio needs to always match the original image.

To create a new aspect frame use `GBin.aspect_frame`:

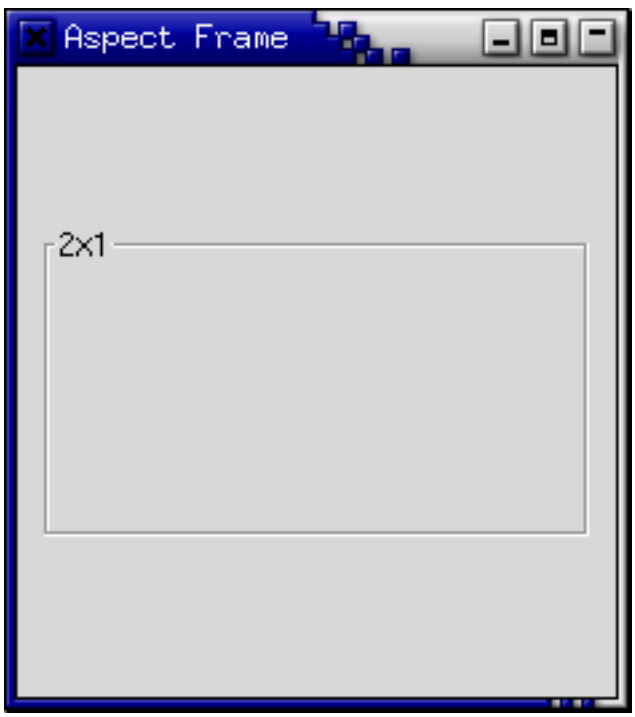
```
val GBin.aspect_frame :
 ?obey_child:bool ->
 ?ratio:float ->
 ?xalign:Gtk.clampf ->
 ?yalign:Gtk.clampf ->
 ?label:string ->
 ?label_xalign:Gtk.clampf ->
 ?label_yalign:Gtk.clampf ->
 ?shadow_type:Gtk.Tags.shadow_type ->
 ?border_width:int ->
 ?width:int ->
 ?height:int ->
 ?packing:(GObj.widget -> unit) ->
 ?show:bool -> unit -> aspect_frame
```

`xalign` and `yalign` specify alignment as with Alignment widgets. If `obey_child` is true, the aspect ratio of a child widget will match the aspect ratio of the ideal size it requests. Otherwise, it is given by `ratio`.

To change the options of an existing aspect frame, you can use:

```
method set_xalign : float -> unit
method set_yalign : float -> unit
method set_ratio : float -> unit
method set_obey_child : bool -> unit
```

As an example, the following program uses an AspectFrame to present a drawing area whose aspect ratio will always be 2:1, no matter how the user resizes the top-level window.




```
(* file: aspectframe.ml *)

let main () =
  (* Create a new window; set title and border width *)
  let window = GWindow.window ~title:"Aspect Frame" ~border_width:10 () in

  (* Here we connect the "destroy" event to a signal handler *)
```

```
  window#connect#destroy ~callback:GMain.Main.quit;

  (* Create a Frame
   * Set the frame's label
   * Align the label at the right of the frame
   * Set the style of the frame *)
  let aspect_frame = GBin.aspect_frame ~label:"2x1"
    ~xalign:0.5 (* center x *)
    ~yalign:0.5 (* center y *)
    ~ratio:2.0 (* xsize/ysize = 2.0 *)
    ~obey_child:false (* ignore child's aspect *)
    ~packing:window#add () in

  (* Now add a child widget to the aspect frame *)
  (* Ask for a 200x200 widnow, but the AspectFrame will give us a 200x100
   * window since we are forcing a 2x1 aspect ratio *)
  let drawing_area = GMisc.drawing_area ~width:200 ~height:200 ~packing:aspect_frame#add () in

  window#show ();
  GMain.Main.main ()

let _ = Printexc.print main ()
```

12.7. Paned Window Widgets

The paned window widgets are useful when you want to divide an area into two parts, with the relative size of the two parts controlled by the user. A groove is drawn between the two portions with a handle that the user can drag to change the ratio. The division can either be horizontal ('HORIZONTAL) or vertical ('VERTICAL).

To create a new paned window, call `GPack.paned` with orientation('HORIZONTAL or 'VERTICAL):

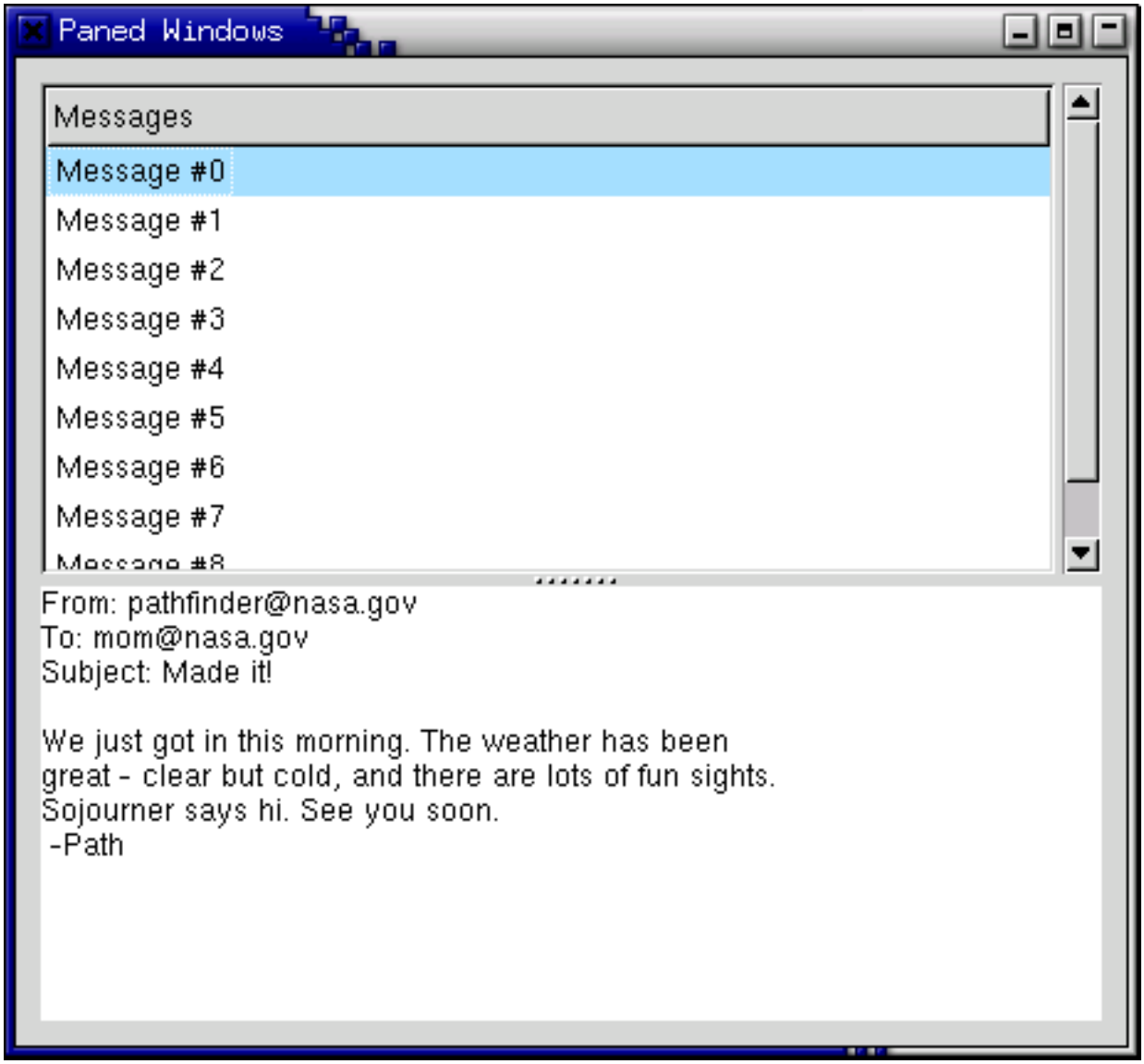
```
val GPack.paned :
 Gtk.Tags.orientation ->
 ?border_width:int ->
 ?width:int ->
 ?height:int ->
 ?packing:(GObj.widget -> unit) ->
 ?show:bool -> unit -> paned
```

After creating the paned window widget, you need to add child widgets to its two halves. To do this, use the functions:

```
method add1 : GObj.widget -> unit
method add2 : GObj.widget -> unit
```

`add1` method adds the child widget to the left or top half of the paned window. `add2` method adds the child widget to the right or bottom half of the paned window.

As an example, we will create part of the user interface of an imaginary email program. A window is divided into two portions vertically, with the top portion being a list of email messages and the bottom portion the text of the email message. Most of the program is pretty straightforward. A couple of points to note: text can't be added to a Text widget until it is realized. This could be done by calling #misc#realize method, but as a demonstration of an alternate technique, we connect a handler to the "realize" signal to add the text. Also, we need to add the `GTK_SHRINK` option to some of the items in the table containing the text window and its scrollbars, so that when the bottom portion is made smaller, the correct portions shrink instead of being pushed off the bottom of the window.




```
(* file: paned.ml *)

let cols = new GTree.column_list
let str_col = cols#add Gobject.Data.string

(* Create the list of "messages" *)
let create_list () =
  (* Create a new scrolled window, with scrollbars only if needed *)
  let scrolled_window = GBin.scrolled_window
    ~hpolicy:`AUTOMATIC ~vpolicy:`AUTOMATIC () in

  let model = GTree.list_store cols in
  let treeview = GTree.view ~model ~packing:(scrolled_window#add_with_viewport) () in

  for i = 0 to 10 do
    let iter = model#append () in
 model#set ~row:iter ~column:str_col (Printf.sprintf "Message #%d" i)
  done;
  let renderer = GTree.cell_renderer_text [] in
  let column = GTree.view_column ~title:"Messages"
    ~renderer:(renderer, ["text", str_col]) () in
  treeview#append_column column;
  scrolled_window#coerce

(* Add some text to our text widget - this is a callback that is invoked
 * when our window is realized. We could also force our window to be
 * realized with #misc#realize, but it would have to be part of
```

```
 * a hierarchy first *)
let insert_text (buffer: GText.buffer) =
  let iter = buffer#get_iter `START in
  buffer#insert ~iter (
    "From: pathfinder@nasa.gov\n" ^
    "To: mom@nasa.gov\n" ^
    "Subject: Made it!\n" ^
    "\n" ^
    "We just got in this morning. The weather has been\n" ^
    "great - clear but cold, and there are lots of fun sights.\n" ^
    "Sojourner says hi. See you soon.\n" ^
    " -Path\n")

(* Create a scrolled text area that displays a "message" *)
let create_text () =
  let scrolled_window = GBin.scrolled_window
    ~hpolicy:`AUTOMATIC ~vpolicy:`AUTOMATIC () in
  let view = GText.view ~packing:scrolled_window#add () in
  let buffer = view#buffer in
  insert_text buffer;
  scrolled_window#coerce

let main () =
  (* Create a new window; set title and border width *)
  let window = GWindow.window ~title:"Paned Windows" ~border_width:10
    ~width:450 ~height:400 () in

  (* Set a handler for destroy event that immediately exits GTK. *)
  window#connect#destroy ~callback:GMain.Main.quit;

  (* create a vpaned widget and add it to our toplevel window *)
  let vpaned = GPack.paned `VERTICAL ~packing:window#add () in

  (* Now create the contents of the two halves of the window *)
  let list = create_list () in
  vpaned#add1 list;

  let text = create_text () in
  vpaned#add2 text;

  window#show ();
  GMain.Main.main ()

let _ = Printexc.print main ()
```

12.8. Viewports

It is unlikely that you will ever need to use the Viewport widget directly. You are much more likely to use the Scrolled Window widget which itself uses the Viewport.

A viewport widget allows you to place a larger widget within it such that you can view a part of it at a time. It uses Adjustments to define the area that is currently in view.

A Viewport is created with the function `GBin.viewport`

```
val GBin.viewport :
 ?hadjustment:GData.adjustment ->
 ?vadjustment:GData.adjustment ->
 ?shadow_type:Gtk.Tags.shadow_type ->
 ?border_width:int ->
 ?width:int ->
 ?height:int ->
 ?packing:(GObj.widget -> unit) ->
 ?show:bool -> unit -> viewport
```

As you can see you can specify the horizontal and vertical Adjustments that the widget is to use when you create the widget. It will create its own if you don't pass the arguments.

You can get and set the adjustments after the widget has been created using the following four functions:

```
method hadjustment : GData.adjustment
method vadjustment : GData.adjustment
method set_hadjustment : GData.adjustment -> unit
method set_vadjustment : GData.adjustment -> unit
```

The other viewport function is used to alter its appearance:

```
method set_shadow_type : Gtk.Tags.shadow_type -> unit
```

Possible values for the `Gtk.Tags.shadow_type` parameter are:

```
  'NONE,
  'IN,
  'OUT,
  'ETCHED_IN,
  'ETCHED_OUT
```

12.9. Scrolled Windows

Scrolled windows are used to create a scrollable area with another widget inside it. You may insert any type of widget into a scrolled window, and it will be accessible regardless of the size by using the scrollbars.

The function `GBin.scrolled_window` is used to create a new scrolled window.

```
val GBin.scrolled_window :
 ?hadjustment:GData.adjustment ->
 ?vadjustment:GData.adjustment ->
 ?hpolicy:Gtk.Tags.policy_type ->
 ?vpolicy:Gtk.Tags.policy_type ->
 ?placement:Gtk.Tags.corner_type ->
 ?shadow_type:Gtk.Tags.shadow_type ->
 ?border_width:int ->
 ?width:int ->
 ?height:int ->
 ?packing:(GObj.widget -> unit) ->
 ?show:bool -> unit -> scrolled_window
```

Where the argument `hadjustment` is the adjustment for the horizontal direction, and `vadjustment`, the adjustment for the vertical direction. These are almost always not given.

```
method set_hpolicy : Gtk.Tags.policy_type -> unit
method set_vpolicy : Gtk.Tags.policy_type -> unit
```

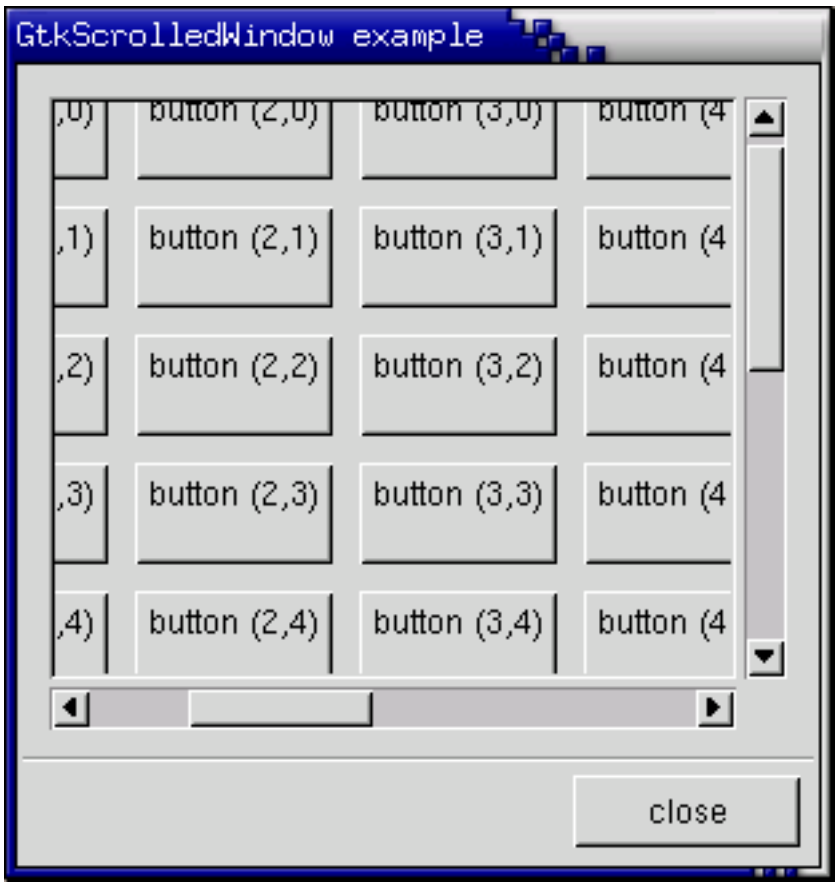
This sets the policy to be used with respect to the scrollbars. The `set_hpolicy` sets the policy for the horizontal scrollbar, and the `set_vpolicy` for the vertical scrollbar.

The policy may be one of `'AUTOMATIC` or `'ALWAYS`. `'AUTOMATIC` will automatically decide whether you need scrollbars, whereas `'ALWAYS` will always leave the scrollbars there.

You can then place your object into the scrolled window using the following function.

```
method add_with_viewport : GObj.widget -> unit
```

Here is a simple example that packs a table with 100 toggle buttons into a scrolled window. I've only commented on the parts that may be new to you.




```
(* file: scrolledwin.ml *)

let main () =
  (* Create a new dialog window for the scrolled window to be
   * packed into. *)
  let window = GWindow.dialog ~title:"ScrolledWindow example" ~width:300 ~height:300 ~border_width:0 
  window#connect#destroy ~callback:GMain.Main.quit;

  (* Create a new scrolled window *)
  let scrolled_window = GBin.scrolled_window ~border_width:10
    ~hpolicy:`AUTOMATIC ~vpolicy:`AUTOMATIC ~packing:window#vbox#add () in

  (* Create a table of 10 by 10 squares.
   * Set the spacing to 10 on x and 10 on y *)
  let table = GPack.table ~rows:10 ~columns:10 ~row_spacings:10 ~col_spacings:10
    ~packing:scrolled_window#add_with_viewport () in

  for i = 0 to 10 do
    for j=0 to 10 do
      GButton.toggle_button
        ~label:("button ("^ string_of_int i ^","^ string_of_int j ^")\n")
        ~packing:(table#attach ~left:i ~top:j ~expand:`BOTH) ()
    done
  done;

  (* Add a "close" button to the bottom of the dialog *)
  let button = GButton.button ~label:"close" ~packing:window#action_area#add () in
  button#connect#clicked ~callback:(window#destroy);

  (* This grabs this button to be the default button. Simply hitting
   * the "Enter" key will cause this button to activate. *)
  button#grab_default ();

  window#show ();
  GMain.Main.main ()

let _ = Printexc.print main ()
```

Try playing with resizing the window. You'll notice how the scrollbars react. You may also wish to use the #misc#set_size_request method call to set the default size of the window or other widgets.

12.10. Button Boxes

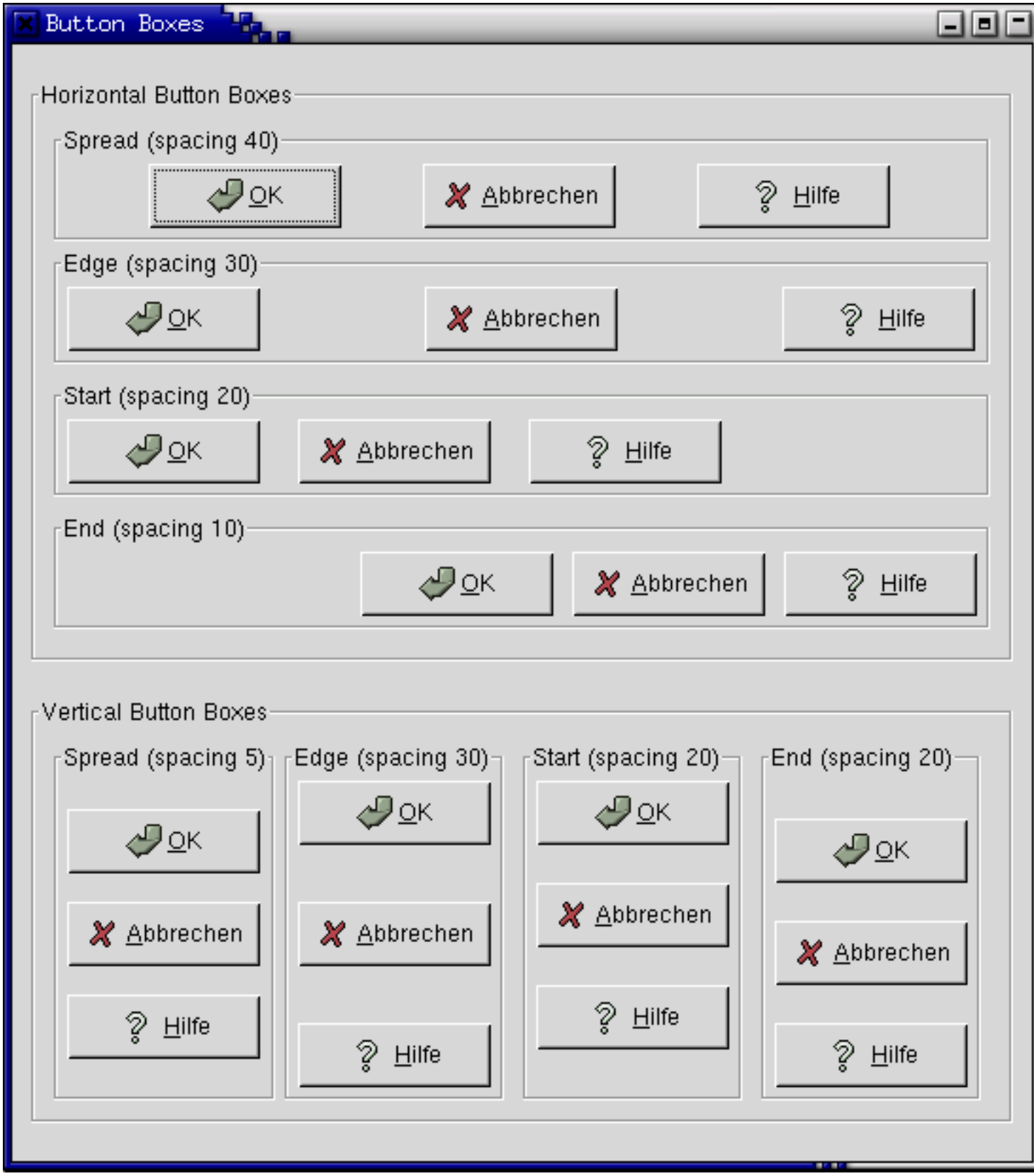
Button Boxes are a convenient way to quickly layout a group of buttons. They come in both horizontal and vertical flavours. You create a new Button Box with the following function `GPack.button_box`, which create a horizontal or vertical box according to the given argument ;'HORIZONTAL or 'VERTICAL:

```
val GPack.button_box :
 Gtk.Tags.orientation ->
 ?spacing:int ->
 ?child_width:int ->
 ?child_height:int ->
 ?child_ipadx:int ->
 ?child_ipady:int ->
 ?layout:GtkPack.BBox.bbox_style ->
 ?border_width:int ->
 ?width:int ->
 ?height:int ->
 ?packing:(GObj.widget -> unit) ->
 ?show:bool -> unit -> button_box
```

Buttons are added to a Button Box using the usual function:

```
method add : GObj.widget -> unit
```

Here's an example that illustrates all the different layout settings for Button Boxes.




```
(* file: buttonbox.ml *)

(* Create a Buttn Box with the specified parameters *)
let create_bbox direction title spacing child_width child_height layout =
  let frame = GBin.frame ~label:title () in
  let bbox = GPack.button_box direction ~border_width:5 ~layout
    ~child_height ~child_width ~spacing ~packing:frame#add () in
  GButton.button ~stock:'OK ~packing:bbox#add ();
```

```
  GButton.button ~stock:‘CANCEL ~packing:bbox#add ();
  GButton.button ~stock:‘HELP ~packing:bbox#add ();
  frame#coerce

let main () =
  let window = GWindow.window ~title:"Button Boxes" ~border_width:10 () in
  window #connect#destroy ~callback:GMain.Main.quit;

  let main_vbox = GPack.vbox ~packing:window#add () in

  let frame_horz = GBin.frame ~label:"Horizontal Button Boxes"
    ~packing:(main_vbox#pack ~expand:true ~fill:true ~padding:10) () in

  let vbox = GPack.vbox ~border_width:10 ~packing:frame_horz#add () in

  vbox#add (create_bbox ‘HORIZONTAL "Spread (spacing 40)" 40 85 20 ‘SPREAD);
  vbox#pack (create_bbox ‘HORIZONTAL "Edge (spacing 30)" 30 85 20 ‘EDGE)
    ~expand:true ~fill:true ~padding:5;
  vbox#pack (create_bbox ‘HORIZONTAL "Start (spacing 20)" 20 85 20 ‘START)
    ~expand:true ~fill:true ~padding:5;
  vbox#pack (create_bbox ‘HORIZONTAL "End (spacing 10)" 10 85 20 ‘END)
    ~expand:true ~fill:true ~padding:5;

  let frame_vert = GBin.frame ~label:"Vertical Button Boxes"
    ~packing:(main_vbox#pack ~expand:true ~fill:true ~padding:10) () in

  let hbox = GPack.hbox ~border_width:10 ~packing:frame_vert#add () in
  hbox#add (create_bbox ‘VERTICAL "Spread (spacing 5)" 5 85 20 ‘SPREAD);
  hbox#pack (create_bbox ‘VERTICAL "Edge (spacing 30)" 30 85 20 ‘EDGE)
    ~expand:true ~fill:true ~padding:5;
  hbox#pack (create_bbox ‘VERTICAL "Start (spacing 20)" 20 85 20 ‘START)
    ~expand:true ~fill:true ~padding:5;
  hbox#pack (create_bbox ‘VERTICAL "End (spacing 20)" 20 85 20 ‘END)
    ~expand:true ~fill:true ~padding:5;
  window#show ();

  (* Enter the event loop *)
  GMain.Main.main ()

let _ = Printexc.print main ()
```

12.11. Toolbar

Toolbars are usually used to group some number of widgets in order to simplify customization of their look and layout. Typically a toolbar consists of buttons with icons, labels and tooltips, but any other widget can also be put inside a toolbar. Finally, items can be arranged horizontally or vertically and buttons can be displayed with icons, labels, or both.

Creating a toolbar is (as one may already suspect) done with the function `GButton.toolbar`:

```
val GButton.toolbar :
 ?orientation:Gtk.Tags.orientation ->
 ?style:Gtk.Tags.toolbar_style ->
 ?tooltips:bool ->
 ?border_width:int ->
 ?width:int ->
 ?height:int ->
 ?packing:(GObj.widget -> unit) ->
 ?show:bool -> unit -> toolbar
```

After creating a toolbar one can insert button, radio_button, toggle_button or any widget types into the toolbar. To describe an item we need a label text, a tooltip text, a private tooltip text, an icon for the button and a callback function for it. For example, to insert an item you may use the following functions:

```
method insert_button :
 ?text:string ->
 ?tooltip:string ->
 ?tooltip_private:string ->
 ?icon:GObj.widget ->
```

```
 ?pos:int ->
 ?callback:(unit -> unit) -> unit -> button

method insert_radio_button :
 ?text:string ->
 ?tooltip:string ->
 ?tooltip_private:string ->
 ?icon:GObj.widget ->
 ?pos:int ->
 ?callback:(unit -> unit) -> unit -> radio_button

method insert_toggle_button :
 ?text:string ->
 ?tooltip:string ->
 ?tooltip_private:string ->
 ?icon:GObj.widget ->
 ?pos:int ->
 ?callback:(unit -> unit) -> unit -> toggle_button

method insert_widget :
 ?tooltip:string ->
 ?tooltip_private:string ->
 ?pos:int ->
 GObj.widget -> unit

pos: default value is (-1)
```

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. It's default value is -1.

To simplify adding spaces between toolbar items, you may use the following function:

```
method insert_space : ?pos:int -> unit -> unit
```

If it's required, the orientation of a toolbar and its style can be changed "on the fly" using the following functions:

```
method set_orientation : Gtk.Tags.orientation -> unit
method set_style : Gtk.Tags.toolbar_style -> unit
method set_tooltips : bool -> unit
```

Where `orientation` is one of `'HORIZONTAL` or `'VERTICAL`. The `style` is used to set appearance of the toolbar items by using one of `'ICONS`, `'TEXT`, or `'BOTH`.

To show some other things that can be done with a toolbar, let's take the following program (we'll interrupt the listing with some additional explanations):

```
(* file: toolbar.ml *)

(* that's easy... when one of the buttons is toggled,
 * set the style of the toolbar accordingly *)
let radio_event toolbar style () =
  toolbar#set_style style

(* just check given toggle button and enable/disable
 * tooltips *)
let toggle_event toolbar button () =
  toolbar#set_tooltips button#active
```

The above are just two callback functions that will be called when one of the buttons on a toolbar is pressed. You should already be familiar with things like this if you've already used toggle buttons (and radio buttons).

```
let main () =
  (* Create a new window with a given title, and nice size *)
  let dialog = GWindow.dialog ~title:"Toolbar Tutorial" () in

  (* typically we quit if someone tries to close us *)
  dialog#connect#destroy ~callback:GMain.Main.quit;

  (* we neecd to realize the window because we use pixmaps for
   * items on the toolbar in the context of it *)
  dialog#misc#realize ();
```

```
  (* to make it nice we'll put the toolbar into the handle box,
   * so that it can be detached from the main window *)
  let handlebox = GBin.handle_box ~packing:dialog#vbox#add () in
```

The above should be similar to any other GTK application. Just initialization of GTK, creating the window, etc. There is only one thing that probably needs some explanation: a handle box. A handle box is just another box that can be used to pack widgets in to. The difference between it and typical boxes is that it can be detached from a parent window (or, in fact, the handle box remains in the parent, but it is reduced to a very small rectangle, while all of its contents are reparented to a new freely floating window). It is usually nice to have a detachable toolbar, so these two widgets occur together quite often.

```
  (* toolbar will be horizontal, with both icons and text, and
   * with 5pxl spaces between items and finally,
   * we'll also put it into our handlebox *)
  let toolbar = GButton.toolbar
    ~orientation:'HORIZONTAL
    ~style:'BOTH
    ~border_width:5 (* ~space_size:5 *)
    ~packing:handlebox#add () in

  (* we need icon for toolbar buttons *)
  let icon () =
    let info = GDraw.pixmap_from_xpm ~file:"gtk.xpm" () in
    (GMisc.pixmap info ())#coerce
  in
```

Well, what we do above is just a straightforward initialization of the toolbar widget.

```
  (* our first item is "close" button *)
  let button = toolbar#insert_button
    ~text:"Close"
    ~tooltip:"Close this app"
    ~tooltip_private:"Private"
    ~icon:(icon ())
    ~callback:GMain.Main.quit () in
  toolbar#insert_space (); (* space after item *)
```

In the above code you see the simplest case: adding a button to toolbar. Just before appending a new item, we have to construct an image widget to serve as an icon for this item; this step will have to be repeated for each new item. Just after the item we also add a space, so the following items will not touch each other. As you see toolbar#insert_button returns a our newly created button widget, so that we can work with it in the normal way.

```
  (* now, lets make our radio buttons group... *)
  let icon_button = toolbar#insert_radio_button
    ~text:"Icon"
    ~tooltip:"Only icons in toolbar"
    ~tooltip_private:"Private"
    ~icon:(icon ())
    ~callback:(radio_event toolbar 'ICONS) () in
  toolbar#insert_space ();
```

Here we begin creating a radio buttons group. To do this we use toolbar#insert_radio_button. In the above case we start creating a radio group. In creating other radio buttons for this group the previous button in the group is required, so that a list of buttons can be easily constructed (see the section on Radio Buttons earlier in this tutorial).

```
  (* following radio buttons refer to previous ones *)
  let text_button = toolbar#insert_radio_button
    ~text:"Text"
    ~tooltip:"Only texts in toolbar"
    ~tooltip_private:"Private"
    ~icon:(icon ())
    ~callback:(radio_event toolbar 'TEXT) () in
  text_button#set_group icon_button#group;
  toolbar#insert_space ();

  let both_button = toolbar#insert_radio_button
    ~text:"Both"
    ~tooltip:"Icons and text in toolbar"
    ~tooltip_private:"Private"
```

```
    ~icon:(icon ())
    ~callback:(radio_event toolbar 'BOTH) () in
  both_button#set_group text_button#group;
  both_button#set_active true;
  toolbar#insert_space ();
```

In the end we have to set the state of one of the buttons manually.

```
  (* here we have just a simple toggle button *)
  let tooltip_button = toolbar#insert_toggle_button
    ~text:"Tooltips"
    ~tooltip:"Toolbar with or without tips"
    ~tooltip_private:"Private"
    ~icon:(icon ()) () in
  tooltip_button#connect#clicked ~callback:(toggle_event toolbar tooltip_button);
  tooltip_button#set_active true;
  toolbar#insert_space ();
```

A toggle button can be created in the obvious way (if one knows how to create radio buttons already).

```
  (* to pack a widget into toolbar, we only have to
   * create it and append it with appropriate tooltip *)
  let entry = GEdit.entry () in
  toolbar#insert_widget
    ~tooltip:"This is just an entry"
    ~tooltip_private:"Private"
    entry#coerce;
```

As you see, adding any kind of widget to a toolbar is simple.

```
  (* that's it! let's show everything *)
  dialog#show ();
  (* rest in GMain.Main.main () and wait for the fun to begin! *)
  GMain.Main.main ()

let _ = Printexc.print main ()
```

So, here we are at the end of toolbar tutorial. Of course, to appreciate it in full you need also this nice XPM icon, so here it is:

> **Note:** The following xpm file is c style but it's ok in ocaml too.

```
/* XPM */
static char * gtk_xpm[] = {
"32 39 5 1",
".      c none",
"+      c black",
"@      c #3070E0",
"#      c #F05050",
"$      c #35E035",
".............+.............",
"............+++++...........",
"...........+++++@@++........",
".........+++++@@@@@@++......",
"........++++@@@@@@@@@@++....",
"......++++@@++++++++@@@++...",
".....+++@@@++++++++++@@@++..",
"...+++@@@@+++@@@@@@++++@@@@+",
"..+++@@@@+++@@@@@@@@+++@@@@@++",
".++@@@@@@+++@@@@@@@@@@@@@@@@@++",
".+#+@@@@@@++@@@@+++@@@@@@@@@@@+",
".+###++@@@@+++@@@+++++@@@@@@@$@.",
".+####+@@@@+++@@@+++@@@@@++$$$@.",
".+########@@@+++++++@@@@@+@$$$$@.",
".+#########@@@@+++@@@@++@$$$$$$+.",
".+##########++++@@@@@@@++@$$$$$$$$+.",
".+########+##+@@@@+++$$$$$$@@$$+.",
".+########+##+@@++@$$$$$$++$$$+.",
```

```
".+########+##+@@$$$$$$$@+@$$@+.",
".+##########+++@$$@+@$$@++$$$@+.",
".+############+#++$$@+@$$++$$$$+..",
".++############+#++$$@+@$++@$$$$+..",
".+############+##++$$++@+++$$$$$+..",
".++##########+##+#++$$+++++@$$$$$+..",
".++#########+#####++$$++++++@$$$@+..",
".+######++######++$$+++@++++@$@+..",
".+######++######++$$++@$$@+++$@@..",
".++#####++######++$$++$$$$$+@$@++.",
".++#####++######++$$++$$$$$$$$+++.",
".+++####+######++$$++$$$$$$$@+++.",
"..+++##########+@$$+@$$$$$$+++...",
"...+++#########+@$$$$$$$$@+++....",
".....+++#######+@$$$$$$$+++......",
"......+++######+@$$$$$@++........",
".......+++#####+@$$$$+++.........",
".........++###+$$$@++............",
"..........++##+$@+++.............",
"...........+++++++..............",
".............++++................"};
```

12.12. Notebooks

The NoteBook Widget is a collection of "pages" that overlap each other, each page contains different information with only one page visible at a time. This widget has become more common lately in GUI programming, and it is a good way to show blocks of similar information that warrant separation in their display.

The first function call you will need to know, as you can probably guess by now, is used to create a new notebook widget: see `GPack.notebook`.

```
val GPack.notebook :
 ?enable_popup:bool ->
 ?homogeneous_tabs:bool ->
 ?scrollable:bool ->
 ?show_border:bool ->
 ?show_tabs:bool ->
 ?tab_border:int ->
 ?tab_pos:Gtk.Tags.position ->
 ?border_width:int ->
 ?width:int ->
 ?height:int ->
 ?packing:(GObj.widget -> unit) ->
 ?show:bool -> unit -> notebook
```

Once the notebook has been created, there are a number of functions that operate on the notebook widget. Let's look at them individually.

The first one we will look at is how to position the page indicators. These page indicators or "tabs" as they are referred to, can be positioned in four ways: top, bottom, left, or right.

```
method set_tab_pos : Gtk.Tags.position -> unit
```

Gtk.Tags.position will be one of the following, which are pretty self explanatory:

```
  'LEFT
  'RIGHT
  'TOP
  'BOTTOM
```

`'TOP` is the default.

Next we will look at how to add pages to the notebook. There are three ways to add pages to the NoteBook. Let's look at the first two together as they are quite similar.

```
method append_page :
 ?tab_label:GObj.widget ->
 ?menu_label:GObj.widget ->
 GObj.widget -> unit
```

```
method prepend_page :
 ?tab_label:GObj.widget ->
 ?menu_label:GObj.widget ->
 GObj.widget -> unit
```

These functions add pages to the notebook by inserting them from the back of the notebook (append), or the front of the notebook (prepend). The given widget is placed within the notebook page, and `tab_label` is the label for the page being added. The given widget must be created separately, and is typically a set of options setup within one of the other container widgets, such as a table.

The final function for adding a page to the notebook contains all of the properties of the previous two, but it allows you to specify what position you want the page to be in the notebook.

```
method insert_page :
 ?tab_label:GObj.widget ->
 ?menu_label:GObj.widget ->
 pos:int ->
 GObj.widget -> unit
```

The parameters are the same as _append_ and _prepend_ except it contains an extra parameter, `pos`. This parameter is used to specify what place this page will be inserted into the first page having position zero.

Now that we know how to add a page, lets see how we can remove a page from the notebook.

```
method remove_page : int -> unit
```

This function takes the page number of int type and removes it from the notebook.

To find out what the current page is in a notebook use the function:

```
method current_page : int
```

These next two functions are simple calls to move the notebook page forward or backward. Simply provide the respective function call with the notebook widget you wish to operate on. Note: When the NoteBook is currently on the last page, and next_page() is called, the notebook will wrap back to the first page. Likewise, if the NoteBook is on the first page, and prev_page() is called, the notebook will wrap to the last page.

```
method next_page : unit -> unit
method previous_page : unit -> unit
```

This next function sets the "active" page. If you wish the notebook to be opened to page 5 for example, you would use this function. Without using this function, the notebook defaults to the first page.

```
method goto_page : int -> unit
```

The next two functions add or remove the notebook page tabs and the notebook border respectively.

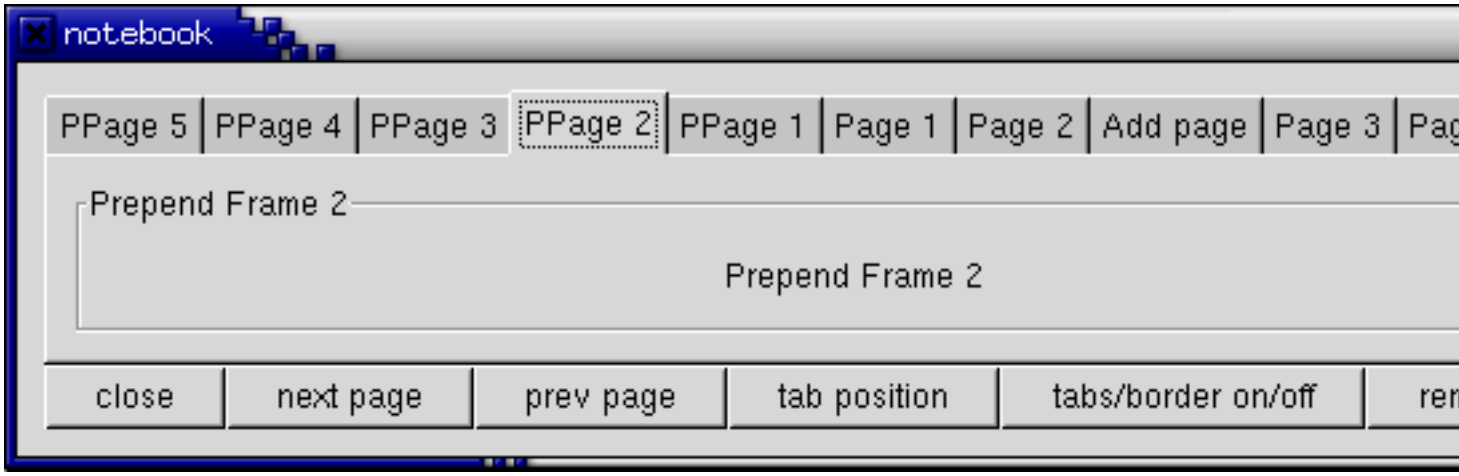
```
method set_show_tabs : bool -> unit
method set_show_border : bool -> unit
```

The next function is useful when the you have a large number of pages, and the tabs don't fit on the page. It allows the tabs to be scrolled through using two arrow buttons.

```
method set_scrollable : bool -> unit
```

`show_tabs`, `show_border` and `scrollable` methods can be either true or false.

Now let's look at an example, it is expanded from the `testgtk.c` code that comes with the GTK distribution. This small program creates a window with a notebook and six buttons. The notebook contains 11 pages, added in three different ways, appended, inserted, and prepended. The buttons allow you rotate the tab positions, add/remove the tabs and border, remove a page, change pages in both a forward and backward manner, and exit the program.




```
(* file: notebook.ml *)

(* This function rotates the position of the tabs *)
let rotate_book notebook () =
  notebook#set_tab_pos
    (match notebook#tab_pos with
    | `BOTTOM -> `LEFT
    | `LEFT -> `TOP
    | `TOP -> `RIGHT
    | `RIGHT -> `BOTTOM
 )

(* Add/Remove the page tabs and the borders *)
let tabsborder_book notebook () =
  notebook#set_show_tabs (not notebook#show_tabs);
  notebook#set_show_border (not notebook#show_border)

(* Remove a page from the notebook *)
let remove_book notebook () =
  notebook#remove_page notebook#current_page;
  ()

let main () =
  let window = GWindow.window ~title:"Notebook demo" ~border_width:10 ()in
  window#connect#destroy ~callback:GMain.Main.quit;

  let table = GPack.table ~rows:3 ~columns:6 ~packing:window#add () in

  (* Create a new notebook, place the position of the tabs *)
  let notebook = GPack.notebook ~tab_pos:`TOP
    ~packing:(table#attach ~left:0 ~right:6 ~top:0) () in

  for i = 1 to 5 do
 let text = "Append Frame " ^ string_of_int i in
 let label = GMisc.label ~text:("Page " ^ string_of_int i) () in
    let frame = GBin.frame ~label:text ~width:100 ~height:75 ~border_width:10
   ~packing:(notebook#append_page ~tab_label:label#coerce) () in
 let label = GMisc.label ~text ~packing:frame#add () in
 ()
  done;

  (* Now let's add a page to a specified spot *)
  let label = GMisc.label ~text:"Add page" () in
  let checkbutton = GButton.check_button ~label:"Check me please!"
 ~packing:(notebook#insert_page ~tab_label:label#coerce ~pos:2) ()
 in
  checkbutton#misc#set_size_request ~width:100 ~height:75 ();

  (* Now finaly let's prepend pages to the notebook *)
  for i = 1 to 5 do
 let text = "Prepend Frame " ^ string_of_int i in
 let label = GMisc.label ~text:("PPage " ^ string_of_int i) () in
```

```
    let frame = GBin.frame ~label:text ~width:100 ~height:75 ~border_width:10
   ~packing:(notebook#prepend_page ~tab_label:label#coerce) () in
 let label = GMisc.label ~text ~packing:frame#add () in
 ()
 done;

 (* Set what page to start at (page 4) *)
 notebook#goto_page 3;

 (* Create a bunch of buttons *)
 let button = GButton.button ~label:"close"
   ~packing:(table#attach ~left:0 ~top:1) () in
 button#connect#clicked ~callback:GMain.Main.quit;

 let button = GButton.button ~label:"next page"
   ~packing:(table#attach ~left:1 ~top:1) () in
 button#connect#clicked ~callback:notebook#next_page;

 let button = GButton.button ~label:"prev page"
   ~packing:(table#attach ~left:2 ~top:1) () in
 button#connect#clicked ~callback:notebook#previous_page;

 let button = GButton.button ~label:"tab position"
   ~packing:(table#attach ~left:3 ~top:1) () in
 button#connect#clicked ~callback:(rotate_book notebook);

 let button = GButton.button ~label:"tabs/border on/off"
   ~packing:(table#attach ~left:4 ~top:1) () in
 button#connect#clicked ~callback:(tabsborder_book notebook);

 let button = GButton.button ~label:"remove page"
   ~packing:(table#attach ~left:5 ~top:1) () in
 button#connect#clicked ~callback:(remove_book notebook);

 window#show ();
 GMain.Main.main ()

let _ = Printexc.print main ()
```

I hope this helps you on your way with creating notebooks for your GTK applications.

Chapter 13. Menu Widget

There are two ways to create menus: there's the easy way, and there's the hard way. Both have their uses, but you can usually use the Itemfactory (the easy way). The "hard" way is to create all the menus using the calls directly. The easy way is to use the gtk_item_factory calls. This is much simpler, but there are advantages and disadvantages to each approach.

The Itemfactory is much easier to use, and to add new menus to, although writing a few wrapper functions to create menus using the manual method could go a long way towards usability. With the Itemfactory, it is not possible to add images or the character '/' to the menus.

13.1. Manual Menu Creation

In the true tradition of teaching, we'll show you the hard way first. :)

There are three widgets that go into making a menubar and submenus:

- a menu item, which is what the user wants to select, e.g., "Save"
- a menu, which acts as a container for the menu items, and
- a menubar, which is a container for each of the individual menus.

This is slightly complicated by the fact that menu item widgets are used for two different things. They are both the widgets that are packed into the menu, and the widget that is packed into the menubar, which, when selected, activates the menu.

Let's look at the functions that are used to create menus and menubars. This first function `GMenu.menu_bar` is used to create a new menubar.

```
GMenu.menu_bar : ?border_width:int ->
 ?width:int ->
 ?height:int ->
 ?packing:(GObj.widget -> unit) ->
 ?show:bool ->
 unit -> menu_shell
```

This rather self explanatory function creates a new menubar. You use gtk_container_add() to pack this into a window, or the box_pack functions to pack it into a box - the same as buttons: `GMenu.menu`.

```
GMenu.menu : ?accel_path:string ->
 ?border_width:int ->
 ?packing:(menu -> unit) ->
 ?show:bool ->
 unit -> menu
```

This function returns a new menu; it is never actually shown, it is just a container for the menu items. I hope this will become more clear when you look at the example below.

The next call `GMenu.menu_item` is used to create menu item that is packed into the menu (and menubar).

```
GMenu.menu_item : ?use_mnemonic:bool ->
 ?label:string ->
 ?right_justified:bool ->
 ?packing:(menu_item -> unit) ->
 ?show:bool ->
 unit -> menu_item
```

This call is used to create a menu item that is to be displayed. Remember to differentiate between a "menu" as created with GMenu.menu () and a "menu item" as created by the GMenu.menu_item () functions. The menu item will be an actual button with an associated action, whereas a menu will be a container holding menu items.

The ~label option of `GMenu.menu_item` creates a new menu item with a label already packed into it.

Once you've created a menu item you have to put it into a menu. This is done using the function `GMenu.menu#append`. In order to capture when the item is selected by the user, we need to connect to the `activate` signal in the usual way. So, if we wanted to create a standard `File` menu, with the options `Open`, `Save`, and `Quit`, the code would look something like:

```
let create_menu () =
```

```
  let file_menu = GMenu.menu () in
  let item = GMenu.menu_item ~label:"Open" ~packing:file_menu#append () in
  item#connect#activate ~callback:(uprint "Open");
  let item = GMenu.menu_item ~label:"Save" ~packing:file_menu#append () in
  item#connect#activate ~callback:(uprint "Save");
  let item = GMenu.menu_item ~label:"Quit" ~packing:file_menu#append () in
  item#connect#activate ~callback:GMain.Main.quit;
  file_menu
```

At this point we have our menu. Now we need to create a menubar and a menu item for the `File` entry, to which we add our menu. The code looks like this:

```
  let file_menu = create_menu () in
  let menu_bar = GMenu.menu_bar ~packing:window#add () in
  let file_item = GMenu.menu_item ~label:"File" () in
```

Now we need to associate the menu with `file_item`. This is done with the function

```
GMenu.menu_item#set_submenu : menu -> unit
```

So, our example would continue with

```
  file_item#set_submenu file_menu;
```

All that is left to do is to add the menu to the menubar, which is accomplished using the function

```
GMenu.menu_shell#append : GMenu.menu_item -> unit
```

which in our case looks like this:

```
  menu_bar#append file_item;
```

The complete code looks like this:

```
let uprint msg () =
  print_endline msg;
  flush stdout

let create_menu () =
  let file_menu = GMenu.menu () in
  let item = GMenu.menu_item ~label:"Open" ~packing:file_menu#append () in
  item#connect#activate ~callback:(uprint "Open");
  let item = GMenu.menu_item ~label:"Save" ~packing:file_menu#append () in
  item#connect#activate ~callback:(uprint "Save");
  let item = GMenu.menu_item ~label:"Quit" ~packing:file_menu#append () in
  item#connect#activate ~callback:GMain.Main.quit;
  file_menu

let main () =
  let window = GWindow.window () in
  window#connect#destroy ~callback:GMain.Main.quit;
  let file_menu = create_menu () in
  let menu_bar = GMenu.menu_bar ~packing:window#add () in
  let file_item = GMenu.menu_item ~label:"File" () in
  file_item#set_submenu file_menu;
  menu_bar#append file_item;
  window#show ();
  GMain.Main.main ()

let _ = main ()
```

The more compact code looks like this:

```
let uprint msg () =
  print_endline msg;
  flush stdout

let create_menu ~packing () =
  let file_menu = GMenu.menu ~packing () in
  let item = GMenu.menu_item ~label:"Open" ~packing:file_menu#append () in
```

```
  item#connect#activate ~callback:(uprint "Open");
  let item = GMenu.menu_item ~label:"Save" ~packing:file_menu#append () in
  item#connect#activate ~callback:(uprint "Save");
  let item = GMenu.menu_item ~label:"Quit" ~packing:file_menu#append () in
  item#connect#activate ~callback:GMain.Main.quit

let main () =
  let window = GWindow.window () in
  window#connect#destroy ~callback:GMain.Main.quit;
  let menu_bar = GMenu.menu_bar ~packing:window#add () in
  let file_item = GMenu.menu_item ~label:"File" ~packing:menu_bar#append () in
  create_menu ~packing:file_item#set_submenu ();
  window#show ();
  GMain.Main.main ()

let _ = main ()
```

If we wanted the menu right justified on the menubar, such as help menus often are, we can use the ~right_justify option (again on `file_item` in the current example) on menu_item createion.

```
  let item = GMenu.menu_item ~right_justify:true () in
  ...
```

Here is a summary of the top down steps needed to create a menu bar with menus attached:

- Create a new menubar using GMenu.menu_bar (). This step only needs to be done once when creating a series of menus on one menu bar.
- Create a menu item using GMenu.menu_item (). This will be the root of the menu, the text appearing here will be on the menubar itself.

  Use GMenu.menu_bar#append method to put the root menu item onto the menubar.

- Create a new menu using GMenu.menu ()

  Use GMenu.menu_item#set_submenu method to attach the menu to the root menu item (the one created in the above step).

- Use multiple calls to GMenu.menu_item () for each item you wish to have on your menu. And use GMenu.menu#append method to put each of these new items on to the menu.

Creating a popup menu is nearly the same. The difference is that the menu is not posted "automatically" by a menubar, but explicitly by calling the function GMenu.menu#popup method from a button-press event, for example. Take these steps:

- Create an event handling function. It needs to have the prototype

  ```
  let button_pressed ev =
    ...
  ```

  But for poping up menu, you may give one more argument for the event handling function like this:

  ```
  let button_pressed menu ev =
    ...
    menu#popup ~button ~time:(GdkEvent.Button.time ev);
    ...
  ```

  and it will use the event to find out where to pop up the menu.

- In the event handler, if the event is a mouse button press, treat `event` as a button event (which it is) and use it as shown in the sample code to pass information to GMenu.menu#popup method.
- Bind that event handler to a widget with

  ```
  widget#event#connect#button_press ~callback:(button_pressed menu);
  ```

  where `widget` is the widget you are binding to, `handler` is the handling function, and `menu` is a menu created with GMenu.menu (). This can be a menu which is also posted by a menu bar, as shown in the sample code.

13.2. Manual Menu Example

That should about do it. Let's take a look at an example to help clarify.


```
let uprint msg () =
  print_endline msg;
  flush stdout

let create_file_menu ~packing () =
  let file_menu = GMenu.menu ~packing () in
  let f (label, callback) =
    let item = GMenu.menu_item ~label ~packing:file_menu#append () in
    ignore (item#connect#activate ~callback)
  in
  List.iter f [("Open", uprint "Open"); ("Save", uprint "Save"); ("Quit", GMain.Main.quit)];
  file_menu

let button_pressed menu ev =
  let button = GdkEvent.Button.button ev in
  if button = 3
  then (
    menu#popup ~button ~time:(GdkEvent.Button.time ev);
    true
  ) else false

let main () =
  let window = GWindow.window ~title:"GMenu Demo" () in
  window#connect#destroy ~callback:GMain.Main.quit;
  let vbox = GPack.vbox ~packing:window#add () in
  let menu_bar = GMenu.menu_bar ~packing:vbox#add () in
  let file_item = GMenu.menu_item ~label:"File" ~packing:menu_bar#append () in
  let menu = create_file_menu ~packing:file_item#set_submenu () in
  let view = GText.view ~width:200 ~height:100 ~packing:vbox#add () in
  view#event#connect#button_press ~callback:(button_pressed menu);
  window#show ();
  GMain.Main.main ()

let _ = main ()
```

You may also set a menu item to be insensitive and, using an accelerator table, bind keys to menu functions.

13.3. Automatic Menu Generation

You can generate menu automatically using `GToolbox.build_menu`.

```
val GToolbox.build_menu :
 GMenu.menu ->
 entries:menu_entry list -> unit
```

The first argument of this function is the `GMenu.menu` with which various menu entries will be associated. And the function takes a value of `GToolbox.menu_entry type` as the second argument:

```
type menu_entry =
  [ 'I of string * (unit -> unit)
  | 'C of string * bool * (bool -> unit)
```

```
  | `R of (string * bool * (bool -> unit)) list
  | `M of string * menu_entry list
  | `S ]
```

- `` `I ``: means `GMenu.menu_item`. It takes as arguments the label of menu_item and the callback function which is called when the menu_item is selected.
- `` `C ``: means `GMenu.check_menu_item`. It takes as arguments the label of check_menu_item, the default state value, and the callback function which is called when the menu_item is selected.
- `` `R ``: means `GMenu.radio_menu_item`. It takes a radio_menu_item description list as an argument. Each radio_menu_item description consists of (label, default state, callback function).
- `` `M ``: means `GMenu.menu`. It takes as arguments the label of menu and the list of menu_entry which will be associated with this menu.
- `` `S ``: means `GMenu.separator_item`.

You can use `GToolbox.menu_entry` for popup menu using `GToolbox.popup_menu`:

```
val GToolbox.popup_menu :
 entries:menu_entry list ->
 button:int ->
 time:int32 -> unit
```

13.3.1. Automatic Menu Generation Example

```
(* file: menu_entry.ml *)

let print msg () =
  print_endline msg;
  flush stdout

let print_toggle selected =
  if selected
  then print_endline "On"
  else print_endline "Off";
  flush stdout

let print_selected n selected =
  if selected then (
    print_endline (string_of_int n);
    flush stdout
  )

let file_entries = [
  `I ("New", print "New");
  `I ("Open", print "Open");
  `I ("Save", print "Save");
  `I ("Save As", print "Save As");
  `S;
  `I ("Quit", GMain.Main.quit)
]

let option_entries = [
  `C ("Check", false, print_toggle);
  `S;
  `R [("Rad1", true, print_selected 1);
      ("Rad2", false, print_selected 2);
      ("Rad3", false, print_selected 3)]
]

let help_entries = [
  `I ("About", print "About");
]

let entries = [
  `M ("File", file_entries);
  `M ("Options", option_entries);
```

```
  'M ("Help", help_entries)
]

let create_menu label menubar =
  let item = GMenu.menu_item ~label ~packing:menubar#append () in
  GMenu.menu ~packing:item#set_submenu ()

let main () =
  (* Make a window *)
  let window = GWindow.window ~title:"Menu Entry" ~border_width:10 () in
  window#connect#destroy ~callback:GMain.Main.quit;

  let main_vbox = GPack.vbox ~packing:window#add () in

  let menubar = GMenu.menu_bar ~packing:main_vbox#add () in

  let menu = create_menu "File" menubar in
  GToolbox.build_menu menu ~entries:file_entries;

  let menu = create_menu "Options" menubar in
  GToolbox.build_menu menu ~entries:option_entries;

  let menu = create_menu "Help" menubar in
  GToolbox.build_menu menu ~entries:help_entries;

  (* Popup menu *)
  let button = GButton.button ~label:"Popup" ~packing:main_vbox#add () in
  button#connect#clicked ~callback:(fun () ->
    GToolbox.popup_menu ~entries ~button:0
   ~time:(GtkMain.Main.get_current_event_time ())
    );

  window#show ();
  GMain.Main.main ()

let _ = Printexc.print main ()
```

Chapter 14. Undocumented Widgets

These all require authors! :) Please consider contributing to our tutorial.

If you must use one of these widgets that are undocumented, I strongly suggest you take a look at their respective header files in the GTK distribution. GTK's function names are very descriptive. Once you have an understanding of how things work, it's not difficult to figure out how to use a widget simply by looking at its function declarations. This, along with a few examples from others' code, and it should be no problem.

When you do come to understand all the functions of a new undocumented widget, please consider writing a tutorial on it so others may benefit from your time.

14.1. Accel Label

14.2. Option Menu

14.3. Menu Items

14.3.1. Check Menu Item

14.3.2. Radio Menu Item

14.3.3. Separator Menu Item

14.3.4. Tearoff Menu Item

14.4. Curves

14.5. Drawing Area

14.6. Font Selection Dialog

14.7. Message Dialog

14.8. Gamma Curve

14.9. Image

14.10. Plugs and Sockets

14.11. Tree View

14.12. Text View

Chapter 15. Setting Widget Attributes

This describes the functions used to operate on widgets. These can be used to set style, padding, size, etc. You can get access these method through misc method, for example, button#misc#hide ().

For full descriptions, see `GObj.misc_ops`.

(Maybe I should make a whole section on accelerators.)

```
method activate : unit -> bool
method set_name : string -> unit
method name : string
method set_sensitive : bool -> unit
method set_style : style -> unit
method style : style
method set_size_request : ?width:int -> ?height:int -> unit -> unit
method set_can_focus : bool -> unit
method grab_focus : unit -> unit
method realize : unit -> unit
method show : unit -> unit
method hide : unit -> unit
```

Chapter 16. Timeouts and Idle Functions

16.1. Timeouts

You may be wondering how you make GTK do useful work when in gtk_main. Well, you have several options. Using the following function you can create a timeout function that will be called every "interval" milliseconds.

```
val GMain.Timeout.add : ms:int -> callback:(unit -> bool) -> id
```

The first argument is the number of milliseconds between calls to your function. The second argument is the function you wish to have called. The return value is an integer "tag" which may be used to stop the timeout by calling:

```
val GMain.Timeout.remove : id -> unit
```

You may also stop the timeout function by returning false from your callback function. Obviously this means if you want your function to continue to be called, it should return true.

16.2. Idle Functions

What if you have a function which you want to be called when nothing else is happening ?

```
val GMain.Idle.add : callback:(unit -> bool) -> id
```

This causes GTK to call the specified function whenever nothing else is happening.

```
val GMain.Idle.remove : id -> unit
```

I won't explain the meaning of the arguments as they follow very much like the ones above. The callback function of GMain.Idle.add will be called whenever the opportunity arises. As with the others, returning false will stop the idle function from being called.

Chapter 17. Advanced Event and Signal Handling

17.1. Signal Functions

17.1.1. Connecting and Disconnecting Signal Handlers

```
let handler_id = [widget]#connect#[signal name] ~callback:... in

let handler_id = [widget]#event#connect#[event signal name] ~callback:... in

let handler_id = [widget]#event#connect#after#[event signal name] ~callback:... in

[widget]#misc#disconnect [handler_id];
```

17.1.2. Blocking and Unblocking Signal Handlers

```
[widget]#misc#handler_block [handler_id];
[widget]#misc#handler_unblock [handler_id];
```

17.2. Signal Emission and Propagation

Signal emission is the process whereby GTK runs all handlers for a specific object and signal.

First, note that the return value from a signal emission is the return value of the *last* handler executed. Since event signals are all of type `GTK_RUN_LAST`, this will be the default (GTK supplied) handler, unless you connect with event#connect#after.

The way an event (say "button_press event") is handled, is:

- Start with the widget where the event occured.
- Emit the generic "event" signal. If that signal handler returns a value of TRUE, stop all processing.
- Otherwise, emit a specific, "button_press event" signal. If that returns TRUE, stop all processing.
- Otherwise, go to the widget's parent, and repeat the above two steps.
- Continue until some signal handler returns TRUE, or until the top-level widget is reached.

Some consequences of the above are:

- Your handler's return value will have no effect if there is a default handler, unless you connect with #event#connect#after.
- To prevent the default handler from being run, you need to connect with #event#connect and use GtkSignal.emit_stop_by_name - the return value only affects whether the signal is propagated, not the current emission.

Chapter 18. Clipboard

Note: There is the "Managing Selections" chapter in the original C version tutorial. Even thought LablGtk supports "selection" widget too, "clipboard" widget is simpler so we will make document only for this widget for the time being.

18.1. Overview

Text copy and paste is a good example of clipboard widget. You can do inter-processor communication throuth the widget.

Before copy or paste, you have the access point to the clipboard: see `Gdk.Atom` and `GData.clipboard`

```
val GData.clipboard : Gdk.atom -> clipboard
```

After that, you can do clear clipboard, set data and get data: see `GData.clipboard class`

```
method clear : unit -> unit
method set_text : string -> unit
method text : string option
```

18.2. Clipboard Example

The following code is the program that copy and paste the buttons' state from one application to the other.

Please launch two instance of the same program and select buttons and click "Copy" button in one application. You can view that the button state are changing to the same when you click "Paste" button in the another instance of the program.

```
(* file: clipboard.ml *)

(* Translate string to char list *)
let explode str =
  let len = String.length str in
  let rec loop clist i =
    if i >= len
    then List.rev clist
    else loop (str.[i] :: clist) (i+1)
  in
  loop [] 0

(* Put the status of the buttons to clipboard *)
let put_data buttons clipboard () =
  let append_active str but = if but#active then str ^ "1" else str ^ "0" in
  clipboard#set_text (List.fold_left append_active "" buttons)

(* Get the status of the buttons from clipboard and apply them *)
let get_data buttons clipboard () =
  match clipboard#text with
  | Some txt ->
      if String.length txt = 4 then (
        let to_bool c = if c = '1' then true else false in
        let active = List.map to_bool (explode txt) in
        let button_status = List.combine buttons active in
        List.iter (fun (but, b) -> but#set_active b) button_status
      ) else
        failwith "String length mismatch"
  | None -> ()

let main () =
  (* Create the toplevel window *)
  let window = GWindow.window ~title:"Clipboard Example" ~border_width:10 () in
  window#connect#destroy ~callback:GMain.Main.quit;

  let vbox = GPack.vbox ~packing:window#add () in
  let table = GPack.table ~columns:2 ~rows:2
```

```
    ~row_spacings:5 ~col_spacings:5 ~border_width:10 ~packing:vbox#add () in
  let make_button (left, top, label) =
    GButton.toggle_button ~label ~packing:(table#attach ~left ~top) ()
  in
  let buttons = List.map make_button [(0, 0, "Hello World");
    (1, 0, "Launch two of this program");
    (0, 1, "On one process, select buttons and copy");
    (1, 1, "On the other process, paste");] in

  (* Clipboard *)
  let clipboard = GData.clipboard Gdk.Atom.clipboard in

  let box = GPack.button_box `HORIZONTAL ~spacing:5 ~layout:`END
    ~border_width:10 ~packing:vbox#add () in

  (* "Copy" button *)
  let button = GButton.button ~stock:`COPY ~packing:box#add () in
  button#connect#clicked ~callback:(put_data buttons clipboard);

  (* "Paste" button *)
  let button = GButton.button ~stock:`PASTE ~packing:box#add () in
  button#connect#clicked ~callback:(get_data buttons clipboard);

  window#show ();
  GMain.Main.main ()

let _ = Printexc.print main ()
```

Chapter 19. Drag-and-drop (DND)

GTK+ has a high level set of functions for doing inter-process communication via the drag-and-drop system. GTK+ can perform drag-and-drop on top of the low level Xdnd and Motif drag-and-drop protocols.

19.1. Overview

An application capable of GTK+ drag-and-drop first defines and sets up the GTK+ widget(s) for drag-and-drop. Each widget can be a source and/or destination for drag-and-drop.

Source widgets can send out drag data, thus allowing the user to drag things off of them, while destination widgets can receive drag data. Drag-and-drop destinations can limit who they accept drag data from, e.g. the same application or any application (including itself).

Sending and receiving drop data makes use of GTK+ signals. Dropping an item to a destination widget requires both a data request (for the source widget) and data received signal handler (for the target widget). Additional signal handers can be connected if you want to know when a drag begins (at the very instant it starts), to when a drop is made, and when the entire drag-and-drop procedure has ended (successfully or not).

Your application will need to provide data for source widgets when requested, that involves having a drag data request signal handler. For destination widgets they will need a drop data received signal handler.

So a typical drag-and-drop cycle would look as follows:

1. Drag begins.
2. Drag data request (when a drop occurs).
3. Drop data received (may be on same or different application).
4. Drag data delete (if the drag was a move).
5. Drag-and-drop procedure done.

There are a few minor steps that go in between here and there, but we will get into detail about that later.

19.2. Properties

Drag data has the following properties:

- Drag action type (ie 'COPY, 'MOVE).
- Client specified arbitrary drag-and-drop type (a name and number pair).
- Sent and received data format type.

Drag actions are quite obvious, they specify if the widget can drag with the specified action(s), e.g. 'COPY and/or 'MOVE. A 'COPY would be a typical drag-and-drop without the source data being deleted while 'MOVE would be just like 'COPY but the source data will be 'suggested' to be deleted after the received signal handler is called. There are additional drag actions including 'LINK which you may want to look into when you get to more advanced levels of drag-and-drop.

The client specified arbitrary drag-and-drop type is much more flexible, because your application will be defining and checking for that specifically. You will need to set up your destination widgets to receive certain drag-and-drop types by specifying a name and/or number. It would be more reliable to use a name since another application may just happen to use the same number for an entirely different meaning.

19.3. Functions

You can find the full DragAndDrop specification in `GObj.drag_ops`. And you can use these functions(methods) and events like this:

```
[widget]#drag#[method name]
[widget]#drag#connect#[event name]
```

19.3.1. Setting up the source widget

The method `drag#source_set` specifies a set of target types for a drag operation on a widget.

```
method source_set :
 ?modi:Gdk.Tags.modifier list ->
 ?actions:Gdk.Tags.drag_action list ->
 Gtk.target_entry list -> unit
```

The parameters signify the following:

- `modi` specifies a list of buttons that can start the drag (e.g. 'BUTTON1): see `Gdk.Tags.modifier`
- `Gtk.target_entry list` specifies a table of target data types the drag will support
- `actions` specifies a list of possible actions for a drag from this window

The `Gtk.target_entry` type is the following structure:

```
type target_entry = {
 target : string;
 flags : Tags.target_flags list;
 info : int;
}

type target_flags = [ 'SAME_APP | 'SAME_WIDGET ]
```

The fields specify a string representing the drag type, optional flags and application assigned integer identifier.

If a widget is no longer required to act as a source for drag-and-drop operations, the method `drag#source_unset` can be used to remove a set of drag-and-drop target types.

```
method source_unset : unit -> unit
```

19.3.2. Signals on the source widget:

The source widget is sent the following signals during a drag-and-drop operation.

Table 19-1. Source widget signals

drag_begin	`method beginning : callback:(drag_context -> unit) -> GtkSignal.id`
drag_motion	`method motion : callback:(drag_context -> x:int -> y:int -> time:int32 -> bool) -> GtkSignal.id`
drag_data_get	`method data_get : callback:(drag_context -> selection_context -> info:int -> time:int32 -> unit) -> GtkSignal.id`
drag_data_delete	`method data_delete : callback:(drag_context -> unit) -> GtkSignal.id`
drag_drop	`method drop : callback:(drag_context -> x:int -> y:int -> time:int32 -> bool) -> GtkSignal.id`
drag_end	`method ending : callback:(drag_context -> unit) -> GtkSignal.id`

19.3.3. Setting up a destination widget:

`drag#dest_set` specifies that this widget can receive drops and specifies what types of drops it can receive.

`drag#dest_unset` specifies that the widget can no longer receive drops.

```
method dest_set :
 ?flags:Gtk.Tags.dest_defaults list ->
 ?actions:Gdk.Tags.drag_action list ->
 Gtk.target_entry list -> unit
```

```
method dest_unset : unit -> unit
```

19.3.4. Signals on the destination widget:

The destination widget is sent the following signals during a drag-and-drop operation.

Table 19-2. Destination widget signals

drag_data_received	`method data_received : callback:(drag_context -> x:int -> y:int -> selection_data -> info:int -> time:int32 -> unit) -> GtkSignal.id`

Chapter 20. GTK's rc Files

GTK has its own way of dealing with application defaults, by using rc files. These can be used to set the colors of just about any widget, and can also be used to tile pixmaps onto the background of some widgets.

20.1. Functions For rc Files

When your application starts, it reads the default RC files. which are [SYSCONFDIR]/gtk-2.0/gtkrc and .gtkrc-2.0 in the users home directory. ([SYSCONFDIR] defaults to /usr/local/etc.) You can add default file:

```
val GMain.Rc.add_default_file : string -> unit
```

If you wish to have a special set of widgets that can take on a different style from others, or any other logical division of widgets, use a call to:

```
method misc#set_name : string -> unit
```

Passing your newly created widget as the first argument, and the name you wish to give it as the second. This will allow you to change the attributes of this widget by name through the rc file.

If we use a call something like this:

```
let button = GButton.button ~label:"Special Button" in
button#misc#set_name "special button"
```

Then this button is given the name "special button" and may be addressed by name in the rc file as "special button.GtkButton". [<--- Verify ME!]

The example rc file below, sets the properties of the main window, and lets all children of that main window inherit the style described by the "main button" style. The code used in the application is:

```
let window = GWindow.window () in
window#misc#set_name "main window"
```

And then the style is defined in the rc file using:

```
widget "main window.*GtkButton*" style "main_button"
```

Which sets all the Button widgets in the "main window" to the "main_buttons" style as defined in the rc file.

As you can see, this is a fairly powerful and flexible system. Use your imagination as to how best to take advantage of this.

> **Note:** I have also found, by experimentation, that you have to call this function (GMain.Rc.add_default_file) very early in the program for it to have any effect at all. I have now placed by call right at the very start of the program, before all other code, as that is seemingly the only way to get Gtk to actually read my local resource file. - Rich.

20.2. GTK's rc File Format

The format of the GTK file is illustrated in the example below. This is the testgtkrc file from the GTK distribution, but I've added a few comments and things. You may wish to include this explanation in your application to allow the user to fine tune his application.

There are several directives to change the attributes of a widget.

- fg - Sets the foreground color of a widget.
- bg - Sets the background color of a widget.
- bg_pixmap - Sets the background of a widget to a tiled pixmap.
- font - Sets the font to be used with the given widget.

In addition to this, there are several states a widget can be in, and you can set different colors, pixmaps and fonts for each state. These states are:

- NORMAL - The normal state of a widget, without the mouse over top of it, and not being pressed, etc.
- PRELIGHT - When the mouse is over top of the widget, colors defined using this state will be in effect.
- ACTIVE - When the widget is pressed or clicked it will be active, and the attributes assigned by this tag will be in effect.
- INSENSITIVE - When a widget is set insensitive, and cannot be activated, it will take these attributes.
- SELECTED - When an object is selected, it takes these attributes.

When using the "fg" and "bg" keywords to set the colors of widgets, the format is:

```
fg[<STATE>] = { Red, Green, Blue }
```

Where STATE is one of the above states (PRELIGHT, ACTIVE, etc), and the Red, Green and Blue are values in the range of 0 - 1.0, { 1.0, 1.0, 1.0 } being white. They must be in float form, or they will register as 0, so a straight "1" will not work, it must be "1.0". A straight "0" is fine because it doesn't matter if it's not recognized. Unrecognized values are set to 0.

bg_pixmap is very similar to the above, except the colors are replaced by a filename.

pixmap_path is a list of paths separated by ":"'s. These paths will be searched for any pixmap you specify.

The font directive is simply:

```
font = "<font name>"
```

The only hard part is figuring out the font string. Using xfontsel or a similar utility should help.

The "widget_class" sets the style of a class of widgets. These classes are listed in the widget overview on the class hierarchy.

The "widget" directive sets a specifically named set of widgets to a given style, overriding any style set for the given widget class. These widgets are registered inside the application using the misc#set_name call. This allows you to specify the attributes of a widget on a per widget basis, rather than setting the attributes of an entire widget class. I urge you to document any of these special widgets so users may customize them.

When the keyword `parent` is used as an attribute, the widget will take on the attributes of its parent in the application.

When defining a style, you may assign the attributes of a previously defined style to this new one.

```
style "main_button" = "button"
{
  font = "-adobe-helvetica-medium-r-normal--*-100-*-*-*-*-*-*"
  bg[PRELIGHT] = { 0.75, 0, 0 }
}
```

This example takes the "button" style, and creates a new "main_button" style simply by changing the font and prelight background color of the "button" style.

Of course, many of these attributes don't apply to all widgets. It's a simple matter of common sense really. Anything that could apply, should.

20.3. Example rc file

```
# pixmap_path "<dir 1>:<dir 2>:<dir 3>:..."
#
pixmap_path "/usr/include/X11R6/pixmaps:/home/imain/pixmaps"
#
# style <name> [= <name>]
# {
#   <option>
# }
#
# widget <widget_set> style <style_name>
# widget_class <widget_class_set> style <style_name>

# Here is a list of all the possible states.  Note that some do not apply to
# certain widgets.
#
# NORMAL - The normal state of a widget, without the mouse over top of
```

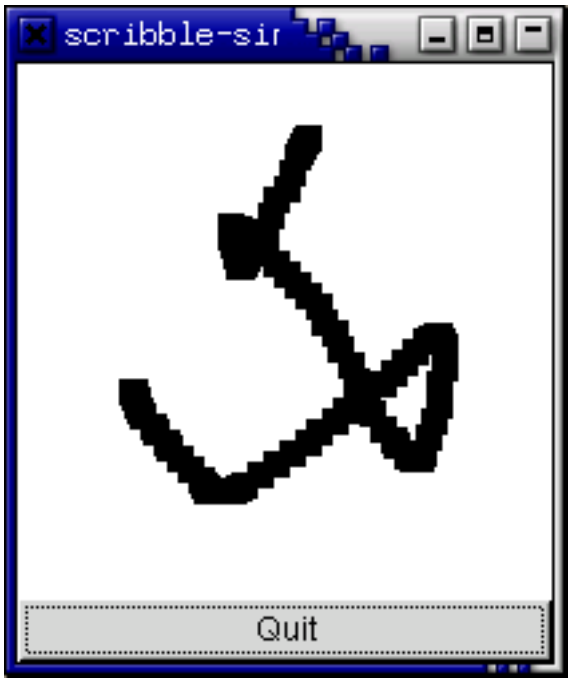
```
# it, and not being pressed, etc.
#
# PRELIGHT - When the mouse is over top of the widget, colors defined
# using this state will be in effect.
#
# ACTIVE - When the widget is pressed or clicked it will be active, and
# the attributes assigned by this tag will be in effect.
#
# INSENSITIVE - When a widget is set insensitive, and cannot be
# activated, it will take these attributes.
#
# SELECTED - When an object is selected, it takes these attributes.
#
# Given these states, we can set the attributes of the widgets in each of
# these states using the following directives.
#
# fg - Sets the foreground color of a widget.
# fg - Sets the background color of a widget.
# bg_pixmap - Sets the background of a widget to a tiled pixmap.
# font - Sets the font to be used with the given widget.
#

# This sets a style called "button".  The name is not really important, as
# it is assigned to the actual widgets at the bottom of the file.

style "window"
{
  #This sets the padding around the window to the pixmap specified.
  #bg_pixmap[<STATE>] = "<pixmap filename>"
  bg_pixmap[NORMAL] = "warning.xpm"
}

style "scale"
{
  #Sets the foreground color (font color) to red when in the "NORMAL"
  #state.

  fg[NORMAL] = { 1.0, 0, 0 }

  #Sets the background pixmap of this widget to that of its parent.
  bg_pixmap[NORMAL] = "<parent>"
}

style "button"
{
  # This shows all the possible states for a button.  The only one that
  # doesn't apply is the SELECTED state.

  fg[PRELIGHT] = { 0, 1.0, 1.0 }
  bg[PRELIGHT] = { 0, 0, 1.0 }
  bg[ACTIVE] = { 1.0, 0, 0 }
  fg[ACTIVE] = { 0, 1.0, 0 }
  bg[NORMAL] = { 1.0, 1.0, 0 }
  fg[NORMAL] = { .99, 0, .99 }
  bg[INSENSITIVE] = { 1.0, 1.0, 1.0 }
  fg[INSENSITIVE] = { 1.0, 0, 1.0 }
}

# In this example, we inherit the attributes of the "button" style and then
# override the font and background color when prelit to create a new
# "main_button" style.

style "main_button" = "button"
{
  font = "-adobe-helvetica-medium-r-normal--*-100-*-*-*-*-*-*"
  bg[PRELIGHT] = { 0.75, 0, 0 }
}

style "toggle_button" = "button"
{
  fg[NORMAL] = { 1.0, 0, 0 }
```

```
  fg[ACTIVE] = { 1.0, 0, 0 }

  # This sets the background pixmap of the toggle_button to that of its
  # parent widget (as defined in the application).
  bg_pixmap[NORMAL] = "<parent>"
}

style "text"
{
  bg_pixmap[NORMAL] = "marble.xpm"
  fg[NORMAL] = { 1.0, 1.0, 1.0 }
}

style "ruler"
{
  font = "-adobe-helvetica-medium-r-normal--*-80-*-*-*-*-*-*"
}

# pixmap_path "~/.pixmaps"

# These set the widget types to use the styles defined above.
# The widget types are listed in the class hierarchy, but could probably be
# just listed in this document for the users reference.

widget_class "GtkWindow" style "window"
widget_class "GtkDialog" style "window"
widget_class "GtkFileSelection" style "window"
widget_class "*Gtk*Scale" style "scale"
widget_class "*GtkCheckButton*" style "toggle_button"
widget_class "*GtkRadioButton*" style "toggle_button"
widget_class "*GtkButton*" style "button"
widget_class "*Ruler" style "ruler"
widget_class "*GtkText" style "text"

# This sets all the buttons that are children of the "main window" to
# the main_button style.  These must be documented to be taken advantage of.
widget "main window.*GtkButton*" style "main_button"
```

Chapter 21. Scribble, A Simple Example Drawing Program

21.1. Overview

In this section, we will build a simple drawing program. In the process, we will examine how to handle mouse events, how to draw in a window, and how to do drawing better by using a backing pixmap. After creating the simple drawing program, we will extend it by adding support for XInput devices, such as drawing tablets. GTK provides support routines which makes getting extended information, such as pressure and tilt, from such devices quite easy.



21.2. Event Handling

The GTK signals we have already discussed are for high-level actions, such as a menu item being selected. However, sometimes it is useful to learn about lower-level occurrences, such as the mouse being moved, or a key being pressed. There are also GTK signals corresponding to these low-level *events*. The handlers for these signals have an extra parameter which is a structure containing information about the event. For instance, motion event handlers are passed a GdkEvent.Motion structure which looks (in part) like: see `GdkEvent.Motion`

```
type t = [ `MOTION_NOTIFY ] Gdk.event
val cast : GdkEvent.any -> t

val time : [< GdkEvent.timed ] Gdk.event -> int32
val x : t -> float
val y : t -> float
val axes : t -> (float * float) option
val state : t -> int
val is_hint : t -> bool
val device : t -> Gdk.device
val x_root : t -> float
val y_root : t -> float
```

`x` and `y` give the coordinates of the event. `state` specifies the modifier state when the event occurred (that is, it specifies which modifier keys and mouse buttons were pressed). It can contain some of the following:

```
`SHIFT
`LOCK
`CONTROL
`MOD1
`MOD2
`MOD3
```

```
‘MOD4
‘MOD5
‘BUTTON1
‘BUTTON2
‘BUTTON3
‘BUTTON4
‘BUTTON5
```

You can test the state whether it includes the given modifier or not, using one of the followings:

```
val Gdk.Convert.test_modifier : Gdk.Tags.modifier -> int -> bool
val Gdk.Convert.modifier : int -> Gdk.Tags.modifier list
```

As for other signals, to determine what happens when an event occurs we call `connect` method. But we also need let GTK know which events we want to be notified about. To do this, we call the method:

```
method event#add : Gdk.Tags.event_mask list -> unit
```

The argument specifies the events we are interested in. It is the list of constants that specify different types of events. For future reference the Gdk.Tags.event_mask are:

```
type event_mask = [ ‘ALL_EVENTS
  | ‘BUTTON1_MOTION
  | ‘BUTTON2_MOTION
  | ‘BUTTON3_MOTION
  | ‘BUTTON_MOTION
  | ‘BUTTON_PRESS
  | ‘BUTTON_RELEASE
  | ‘ENTER_NOTIFY
  | ‘EXPOSURE
  | ‘FOCUS_CHANGE
  | ‘KEY_PRESS
  | ‘KEY_RELEASE
  | ‘LEAVE_NOTIFY
  | ‘POINTER_MOTION
  | ‘POINTER_MOTION_HINT
  | ‘PROPERTY_CHANGE
  | ‘PROXIMITY_IN
  | ‘PROXIMITY_OUT
  | ‘SCROLL
  | ‘STRUCTURE
  | ‘SUBSTRUCTURE
  | ‘VISIBILITY_NOTIFY ]
```

There are a few subtle points that have to be observed when calling `event#add` method. First, it must be called before the X window for a GTK widget is created. In practical terms, this means you should call it immediately after creating the widget. Second, the widget must have an associated X window. For efficiency, many widget types do not have their own window, but draw in their parent’s window. These widgets are:

```
GtkAlignment
GtkArrow
GtkBin
GtkBox
GtkImage
GtkItem
GtkLabel
GtkPixmap
GtkScrolledWindow
GtkSeparator
GtkTable
GtkAspectFrame
GtkFrame
GtkVBox
GtkHBox
GtkVSeparator
GtkHSeparator
```

To capture events for these widgets, you need to use an EventBox widget. See the section on the EventBox widget for details.

For our drawing program, we want to know when the mouse button is pressed and when the mouse is moved, so we specify `'POINTER_MOTION` and `'BUTTON_PRESS`. We also want to know when we need to redraw our window, so we specify `'EXPOSURE`. Although we want to be notified via a Configure event when our window size changes, we don't have to specify the corresponding `'STRUCTURE` flag, because it is automatically specified for all windows.

It turns out, however, that there is a problem with just specifying `'POINTER_MOTION`. This will cause the server to add a new motion event to the event queue every time the user moves the mouse. Imagine that it takes us 0.1 seconds to handle a motion event, but the X server queues a new motion event every 0.05 seconds. We will soon get way behind the users drawing. If the user draws for 5 seconds, it will take us another 5 seconds to catch up after they release the mouse button! What we would like is to only get one motion event for each event we process. The way to do this is to specify `'POINTER_MOTION_HINT`.

When we specify `'POINTER_MOTION_HINT`, the server sends us a motion event the first time the pointer moves after entering our window, or after a button press or release event. Subsequent motion events will be suppressed until we explicitly ask for the position of the pointer using the function:

```
val Gdk.Window.get_pointer_location : Gdk.window -> int * int
```

There is another method, `misc#pointer` method which has a simpler interface:

```
method misc#pointer : int * int
```

The code to set the events for our window then looks like:

```
  drawing_area#event#connect#expose ~callback:expose;
  drawing_area#event#connect#configure ~callback:configure;
  drawing_area#event#connect#motion_notify ~callback:motion_notify;
  drawing_area#event#connect#button_press ~callback:button_pressed;

  drawing_area#event#add [‘EXPOSURE;
    ‘LEAVE_NOTIFY;
    ‘BUTTON_PRESS;
    ‘POINTER_MOTION;
    ‘POINTER_MOTION_HINT];
```

We'll save the "expose" and "configure" callbacks for later. The "motion_notify" and "button_pressed" callbacks are pretty simple:

```
let button_pressed area backing ev =
  if GdkEvent.Button.button ev = 1 then (
    let x = int_of_float (GdkEvent.Button.x ev) in
    let y = int_of_float (GdkEvent.Button.y ev) in
    draw_brush area backing x y
  );
  true

let motion_notify area backing ev =
  let (x, y) =
    if GdkEvent.Motion.is_hint ev
 then area#misc#pointer
 else
      (int_of_float (GdkEvent.Motion.x ev), int_of_float (GdkEvent.Motion.y ev))
  in
  let state = GdkEvent.Motion.state ev in
  if Gdk.Convert.test_modifier ‘BUTTON1 state
  then draw_brush area backing x y;
  true
```

21.3. The DrawingArea Widget, And Drawing

We now turn to the process of drawing on the screen. The widget we use for this is the DrawingArea widget. A drawing area widget is essentially an X window and nothing more. It is a blank canvas in which we can draw whatever we like. A drawing area is created using the call:

```
val GMisc.drawing_area :
 ?width:int ->
 ?height:int ->
 ?packing:(GObj.widget -> unit) ->
```

```
 ?show:bool ->
 unit -> drawing_area
```

The arguments `width` and `height` specifies the default size of the drawing area.

The default size can be overridden, as is true for all widgets, by calling `misc#set_size_request` method, and that, in turn, can be overridden if the user manually resizes the the window containing the drawing area.

It should be noted that when we create a DrawingArea widget, we are *completely* responsible for drawing the contents. If our window is obscured then uncovered, we get an exposure event and must redraw what was previously hidden.

Having to remember everything that was drawn on the screen so we can properly redraw it can, to say the least, be a nuisance. In addition, it can be visually distracting if portions of the window are cleared, then redrawn step by step. The solution to this problem is to use an offscreen *backing pixmap*. Instead of drawing directly to the screen, we draw to an image stored in server memory but not displayed, then when the image changes or new portions of the image are displayed, we copy the relevant portions onto the screen.

To create an offscreen pixmap, we call the function:

```
val GDraw.pixmap :
 width:int ->
 height:int ->
 ?mask:bool ->
 ?window:< misc : #misc_ops; .. > ->
 ?colormap:Gdk.colormap ->
 unit -> pixmap
```

The `window` parameter specifies a GDK window that this pixmap takes some of its properties from. `width` and `height` specify the size of the pixmap. `colormap` tells the *color depth*, that is the number of bits per pixel, for the new window. If the colormap is not specified, `default_colormap()` is used.

We create the pixmap in our "configure" handler. This event is generated whenever the window changes size, including when it is originally created.

```
(* Backing pixmap for drawing area *)
let backing = ref (GDraw.pixmap ~width:200 ~height:200 ())

(* Create a new backing pixmap of the appropriate size *)
let configure window backing ev =
  let width = GdkEvent.Configure.width ev in
  let height = GdkEvent.Configure.height ev in
  let pixmap = GDraw.pixmap ~width ~height ~window () in
  pixmap#set_foreground `WHITE;
  pixmap#rectangle ~x:0 ~y:0 ~width ~height ~filled:true ();
  backing := pixmap;
  true
```

The call to `rectangle` method clears the pixmap initially to white. We'll say more about that in a moment.

Our exposure event handler then simply copies the relevant portion of the pixmap onto the screen (we determine the area we need to redraw by using GdkEvent.Expost.area method to the exposure event):

```
(* Redraw the screen from the backing pixmap *)
let expose (drawing_area:GMisc.drawing_area) (backing:GDraw.pixmap ref) ev =
  let area = GdkEvent.Expose.area ev in
  let x = Gdk.Rectangle.x area in
  let y = Gdk.Rectangle.y area in
  let width = Gdk.Rectangle.width area in
  let height = Gdk.Rectangle.width area in
  let drawing =
    drawing_area#misc#realize ();
    new GDraw.drawable (drawing_area#misc#window)
  in
  drawing#put_pixmap ~x ~y ~xsrc:x ~ysrc:y ~width ~height !backing#pixmap;
  false
```

We've now seen how to keep the screen up to date with our pixmap, but how do we actually draw interesting stuff on our pixmap? There are a large number of calls in GTK's GDK library for drawing on *drawables*. A drawable is simply something that can be drawn upon. It can be a window, a pixmap, or a bitmap (a black and white

image). We've already seen two such calls above, `rectangle` and `put_pixmap` methods. The some of them are: see `GDraw.drawable`

```
method arc :
 x:int ->
 y:int ->
 width:int ->
 height:int ->
 ?filled:bool ->
 ?start:float ->
 ?angle:float ->
 unit -> unit
method line :
 x:int ->
 y:int ->
 x:int ->
 y:int -> unit
method point :
 x:int ->
 y:int -> unit
method polygon :
 ?filled:bool ->
 (int * int) list -> unit
method rectangle :
 x:int ->
 y:int ->
 width:int ->
 height:int ->
 ?filled:bool ->
 unit -> unit
method string :
 string ->
 font:Gdk.font ->
 x:int ->
 y:int -> unit
method points : (int * int) list -> unit
method lines : (int * int) list -> unit
method segments : ((int * int) * (int * int)) list -> unit

method put_layout :
 x:int ->
 y:int ->
 ?fore:color ->
 ?back:color ->
 Pango.layout -> unit
method put_image :
 x:int ->
 y:int ->
 ?xsrc:int ->
 ?ysrc:int ->
 ?width:int ->
 ?height:int ->
 Gdk.image -> unit
method put_pixmap :
 x:int ->
 y:int ->
 ?xsrc:int ->
 ?ysrc:int ->
 ?width:int ->
 ?height:int ->
 Gdk.pixmap -> unit
method put_rgb_data :
 width:int ->
 height:int ->
 ?x:int ->
 ?y:int ->
 ?dither:Gdk.Tags.rgb_dither ->
 ?row_stride:int ->
 Gpointer.region -> unit
method put_pixbuf :
 x:int ->
```

```
 y:int ->
 ?width:int ->
 ?height:int ->
 ?dither:Gdk.Tags.rgb_dither ->
 ?x_dither:int ->
 ?y_dither:int ->
 ?src_x:int ->
 ?src_y:int ->
 GdkPixbuf.pixbuf -> unit
```

All of these functions uses *graphics context* (GC). A graphics context encapsulates information about things such as foreground and background color and line width. GDK has a full set of functions for creating and modifying graphics contexts. `GDraw.drawable` has the default GC and you can change it using:

```
method set_background : color -> unit
method set_foreground : color -> unit
method set_clip_region : Gdk.region -> unit
method set_clip_origin : x:int -> y:int -> unit
method set_clip_mask : Gdk.bitmap -> unit
method set_clip_rectangle : Gdk.Rectangle.t -> unit
method set_line_attributes :
 ?width:int ->
 ?style:Gdk.GC.gdkLineStyle ->
 ?cap:Gdk.GC.gdkCapStyle ->
 ?join:Gdk.GC.gdkJoinStyle ->
 unit -> unit
```

Our function `draw_brush`, which does the actual drawing on the screen, is then:

```
(* Draw a rectangle on the screen *)
let draw_brush (area:GMisc.drawing_area) (backing:GDraw.pixmap ref) x y =
  let x = x - 5 in
  let y = y - 5 in
  let width = 10 in
  let height = 10 in
  let update_rect = Gdk.Rectangle.create ~x ~y ~width ~height in
  !backing#set_foreground `BLACK;
  !backing#rectangle ~x ~y ~width ~height ~filled:true ();
  area#misc#draw (Some update_rect)
```

After we draw the rectangle representing the brush onto the pixmap, we call the method:

```
method misc#draw : Gdk.Rectangle.t option -> unit
```

which notifies X that the given area needs to be updated. X will eventually generate an expose event (possibly combining the areas passed in several calls to `misc#draw`) which will cause our expose event handler to copy the relevant portions to the screen.

We have now covered the entire drawing program except for a few mundane details like creating the main window.

Chapter 22. Contributing

This document, like so much other great software out there, was created for free by volunteers. If you are at all knowledgeable about any aspect of GTK that does not already have documentation, please consider contributing to this document.

If you do decide to contribute, please mail your text to Tony Gale, `gale@gtk.org`. Also, be aware that the entirety of this document is free, and any addition by you provide must also be free. That is, people may use any portion of your examples in their programs, and copies of this document may be distributed at will, etc.

Thank you.

22.1. Ocaml Version Contributing

This document, like so much other great software out there, was created for free by volunteers. If you are at all knowledgeable about any aspect of GTK that does not already have documentation, please consider contributing to this document.

If you do decide to contribute, please mail your text to SooHyoung Oh, `shoh@compiler.kaist.ac.kr`. Also, be aware that the entirety of this document is free, and any addition by you provide must also be free. That is, people may use any portion of your examples in their programs, and copies of this document may be distributed at will, etc.

Thank you.

Chapter 23. Credits

We would like to thank the following for their contributions to this text.

- Bawer Dagdeviren, chamele0n@geocities.com for the menus tutorial.
- Raph Levien, raph@acm.org for hello world ala GTK, widget packing, and general all around wisdom. He's also generously donated a home for this tutorial.
- Peter Mattis, petm@xcf.berkeley.edu for the simplest GTK program.. and the ability to make it :)
- Werner Koch werner.koch@guug.de for converting the original plain text to SGML, and the widget class hierarchy.
- Mark Crichton crichton@expert.cc.purdue.edu for the menu factory code, and the table packing tutorial.
- Owen Taylor owt1@cornell.edu for the EventBox widget section (and the patch to the distro). He's also responsible for the selections code and tutorial, as well as the sections on writing your own GTK widgets, and the example application. Thanks a lot Owen for all you help!
- Mark VanderBoom mvboom42@calvin.edu for his wonderful work on the Notebook, Progress Bar, Dialogs, and File selection widgets. Thanks a lot Mark! You've been a great help.
- Tim Janik timj@gtk.org for his great job on the Lists Widget. His excellent work on automatically extracting the widget tree and signal information from GTK. Thanks Tim :)
- Rajat Datta rajat@ix.netcom.com for the excellent job on the Pixmap tutorial.
- Michael K. Johnson johnsonm@redhat.com for info and code for popup menus.
- David Huggins-Daines bn711@freenet.carleton.ca for the Range Widgets and Tree Widget sections.
- Stefan Mars mars@lysator.liu.se for the CList section.
- David A. Wheeler dwheeler@ida.org for portions of the text on GLib and various tutorial fixups and improvements. The GLib text was in turn based on material developed by Damon Chaplin DAChaplin@msn.com
- David King for style checking the entire document.

And to all of you who commented on and helped refine this document.

Thanks.

23.1. Ocaml Version Credits

Thanks.

Chapter 24. Tutorial Copyright and Permissions Notice

24.1. Ocaml Version Tutorial Copyright and Permissions Notice

Appendix A. GTK Signals

As GTK is an object oriented widget set, it has a hierarchy of inheritance. This inheritance mechanism applies for signals. Therefore, you should refer to the widget hierarchy tree when using the signals listed in this section.

A.1. GObj

A.1.1. GObj.gtkobj

```
method destroy : callback:(unit -> unit) -> GtkSignal.id
```

A.1.2. GObj.widget

`widget_signals` is same as the above gtkobj_signals.

A.2. Widget

A.2.1. Misc signals

The misc signals are used like this:

```
[widget]#misc#connect#[signal name] ~callback:... ;
```

The misc signals are one of followings: see `GObj.misc_signals`

```
inherits
 GObj.gtkobj_signals
method hide : callback:(unit -> unit) -> GtkSignal.id
method map : callback:(unit -> unit) -> GtkSignal.id
method parent_set : callback:(widget option -> unit) -> GtkSignal.id
method realize : callback:(unit -> unit) -> GtkSignal.id
method unrealize : callback:(unit -> unit) -> GtkSignal.id
method selection_get :
 callback:(selection_context -> info:int -> time:int32 -> unit) ->
 GtkSignal.id
method selection_received :
 callback:(selection_data -> time:int32 -> unit) -> GtkSignal.id
method show : callback:(unit -> unit) -> GtkSignal.id
method size_allocate : callback:(Gtk.rectangle -> unit) -> GtkSignal.id
method state_changed : callback:(Gtk.Tags.state_type -> unit) -> GtkSignal.id
method style_set : callback:(unit -> unit) -> GtkSignal.id
method unmap : callback:(unit -> unit) -> GtkSignal.id
```

A.2.2. Drag signals

The drag signals are used as one of the folloing form:

```
[widget]#drag#connect#[signal name] ~callback:... ;
[widget]#drag#connect#after#[signal name] ~callback:... ;
```

The drag signals are: see `GObj.drag_signals`

```
method beginning : callback:(drag_context -> unit) -> GtkSignal.id
method data_delete : callback:(drag_context -> unit) -> GtkSignal.id
method data_get :
 callback:(drag_context -> selection_context -> info:int -> time:int32 -> unit) ->
 GtkSignal.id
method data_received :
 callback:(drag_context -> x:int -> y:int -> selection_data -> info:int -> time:int32 -> unit) ->
```

```
 GtkSignal.id
method drop :
 callback:(drag_context -> x:int -> y:int -> time:int32 -> bool) ->
 GtkSignal.id
method ending : callback:(drag_context -> unit) -> GtkSignal.id
method leave : callback:(drag_context -> time:int32 -> unit) -> GtkSignal.id
method motion :
 callback:(drag_context -> x:int -> y:int -> time:int32 -> bool) ->
 GtkSignal.id
```

A.3. GData.adjustment

```
inherits
 GObj.gtkobj_signals
method changed : callback:(unit -> unit) -> GtkSignal.id
method value_changed : callback:(unit -> unit) -> GtkSignal.id
```

A.4. GRange.range

```
inherits
 GObj.widget_signals
method adjust_bounds : callback:(float -> unit) -> GtkSignal.id
method move_slider : callback:(Gtk.Tags.scroll_type -> unit) -> GtkSignal.id
method value_changed : callback:(unit -> unit) -> GtkSignal.id
```

A.5. GContainer

A.5.1. GContainer.container

```
inherits
 GObj.widget_signals
method adjust_bounds : callback:(float -> unit) -> GtkSignal.id
method add : callback:(GObj.widget -> unit) -> GtkSignal.id
method remove : callback:(GObj.widget -> unit) -> GtkSignal.id
```

A.5.2. GContainer.item

```
inherits
 GContainer.container_signals
method deselect : callback:(unit -> unit) -> GtkSignal.id
method select : callback:(unit -> unit) -> GtkSignal.id
method toggle : callback:(unit -> unit) -> GtkSignal.id
```

A.6. GBin

A.6.1. GBin.handle_box

```
inherits
 GContainer.container_signals
method child_attached : callback:(GObj.widget -> unit) -> GtkSignal.id
method child_detached : callback:(GObj.widget -> unit) -> GtkSignal.id
```

A.6.2. GBin.expanderhandle_box

```
inherits
 GContainer.container_signals
method activate : callback:(unit -> unit) -> GtkSignal.id
```

A.7. GPack.notebook

```
inherits
 GContainer.container_signals
method switch_page : callback:(int -> unit) -> GtkSignal.id
```

A.8. GMenu

A.8.1. GMenu.menu_shell

```
inherits
 GContainer.container_signals
method deactivate : callback:(unit -> unit) -> GtkSignal.id
```

A.8.2. GMenu.menu_item

```
inherits
 GContainer.item_signals
method activate : callback:(unit -> unit) -> GtkSignal.id
```

A.8.3. GMenu.check_menu_item

```
inherits
 GMenu.menu_item_signals
method toggled : callback:(unit -> unit) -> GtkSignal.id
```

A.9. GEdit

A.9.1. GEdit.editable

```
inherits
 GObj.widget_signals
method changed : callback:(unit -> unit) -> GtkSignal.id
method delete_text : callback:(start:int -> stop:int -> unit) -> GtkSignal.id
method insert_text : callback:(string -> pos:int Pervasives.ref -> unit) -> GtkSignal.id
```

A.9.2. GEdit.entry

```
inherits
 GEdit.editable_signals
method activate : callback:(unit -> unit) -> GtkSignal.id
method copy_clipboard : callback:(unit -> unit) -> GtkSignal.id
method cut_clipboard : callback:(unit -> unit) -> GtkSignal.id
method delete_from_cursor : callback:(Gtk.Tags.delete_type -> int -> unit) -> GtkSignal.id
method insert_at_cursor : callback:(string -> unit) -> GtkSignal.id
method move_cursor : callback:(Gtk.Tags.movement_step -> int -> extend:bool -> unit) ->       GtkSigna
```

```
method paste_clipboard : callback:(unit -> unit) -> GtkSignal.id
method populate_popup : callback:(GMenu.menu -> unit) -> GtkSignal.id
method toggle_overwrite : callback:(unit -> unit) -> GtkSignal.id
```

A.9.3. GEdit.spin_button

```
inherits
 GEdit.entry_signals
method change_value : callback:(Gtk.Tags.scroll_type -> unit) -> GtkSignal.id
method input : callback:(unit -> int) -> GtkSignal.id
method output : callback:(unit -> bool) -> GtkSignal.id
method value_changed : callback:(unit -> unit) -> GtkSignal.id
```

A.9.4. GEdit.combo_box

```
inherits
 GContainer.container_signals
method changed : callback:(unit -> unit) -> GtkSignal.id
```

A.10. GButton

A.10.1. GButton.button

```
inherits
 GContainer.container_signals
method clicked : callback:(unit -> unit) -> GtkSignal.id
method enter : callback:(unit -> unit) -> GtkSignal.id
method leave : callback:(unit -> unit) -> GtkSignal.id
method pressed : callback:(unit -> unit) -> GtkSignal.id
method released : callback:(unit -> unit) -> GtkSignal.id
```

A.10.2. GButton.toggle_button

```
inherits
 GButton.button_signals
method toggled : callback:(unit -> unit) -> GtkSignal.id
```

A.10.3. GButton.color_button

```
inherits
 GButton.button_signals
method color_set : callback:(unit -> unit) -> GtkSignal.id
```

A.10.4. GButton.fontcolor_button

```
inherits
 GButton.button_signals
method font_set : callback:(unit -> unit) -> GtkSignal.id
```

A.10.5. GButton.toolbar

```
inherits
 GContainer.container_signals
method orientation_changed : callback:(GtkEnums.orientation -> unit) -> GtkSignal.id
method style_changed : callback:(GtkEnums.toolbar_style -> unit) -> GtkSignal.id
method focus_home_or_end : callback:(bool -> bool) -> GtkSignal.id
Since GTK 2.4

method move_focus : callback:(GtkEnums.direction_type -> bool) -> GtkSignal.id
Since GTK 2.4

method popup_context_menu : callback:(int -> int -> int -> bool) -> GtkSignal.id
Since GTK 2.4
```

A.10.6. GButton.tool_button

```
inherits
 GContainer.container_signals
method clicked : callback:(unit -> unit) -> GtkSignal.id
```

A.10.7. GButton.toggle_tool_button

```
inherits
 GButton.tool_button_signals
method toggled : callback:(unit -> unit) -> GtkSignal.id
```

A.11. GWindow

A.11.1. GWindow.dialog

```
inherits
 GContainer.container_signals
method response : callback:('a -> unit) -> GtkSignal.id
method close : callback:(unit -> unit) -> GtkSignal.id
```

A.11.2. GWindow.file_chooser_dialog

```
inherits
 GWindow.dialog_signals
 GFile.chooser_signals
```

A.12. GFile

A.12.1. GFile.chooser

```
method current_folder_changed : callback:(unit -> unit) -> GtkSignal.id
method selection_changed : callback:(unit -> unit) -> GtkSignal.id
method update_preview : callback:(unit -> unit) -> GtkSignal.id
method file_activated : callback:(unit -> unit) -> GtkSignal.id
```

A.12.2. GFile.chooser_widget

```
inherits
 GObj.widget_signals
 GFile.chooser_signals
```

A.13. GMisc

A.13.1. GMisc.calendar

```
inherits
 GObj.widget_signals
method day_selected : callback:(unit -> unit) -> GtkSignal.id
method day_selected_double_click : callback:(unit -> unit) -> GtkSignal.id
method month_changed : callback:(unit -> unit) -> GtkSignal.id
method next_month : callback:(unit -> unit) -> GtkSignal.id
method next_year : callback:(unit -> unit) -> GtkSignal.id
method prev_month : callback:(unit -> unit) -> GtkSignal.id
method prev_year : callback:(unit -> unit) -> GtkSignal.id
```

A.13.2. GMisc.tips_query

```
inherits
 GObj.widget_signals
method start_query : callback:(unit -> unit) -> GtkSignal.id
method stop_query : callback:(unit -> unit) -> GtkSignal.id
method widget_entered : callback:(GObj.widget option -> text:string -> privat:string -> unit) ->
method widget_selected : callback:(GObj.widget option ->                 text:string -> privat:string
```

A.14. GTree

A.14.1. GTree.model

```
method row_changed : callback:(Gtk.tree_path -> Gtk.tree_iter -> unit) -> GtkSignal.id
method row_deleted : callback:(Gtk.tree_path -> unit) -> GtkSignal.id
method row_has_child_toggled : callback:(Gtk.tree_path -> Gtk.tree_iter -> unit) -> GtkSignal.id
method row_inserted : callback:(Gtk.tree_path -> Gtk.tree_iter -> unit) -> GtkSignal.id
method rows_reordered : callback:(Gtk.tree_path -> Gtk.tree_iter -> unit) -> GtkSignal.id
```

A.14.2. GTree.tree_sortable

```
inherits
 GTree.model_signals
method sort_column_changed : callback:(unit -> unit) -> GtkSignal.id
```

A.14.3. GTree.selection

```
method changed : callback:(unit -> unit) -> GtkSignal.id
```

A.14.4. GTree.view_column

```
inherits
 GObj.gtkobj_signals
method clicked : callback:(unit -> unit) -> GtkSignal.id
```

A.14.5. GTree.view

```
inherits
 GContainer.container_signals
method columns_changed : callback:(unit -> unit) -> GtkSignal.id
method cursor_changed : callback:(unit -> unit) -> GtkSignal.id
method expand_collapse_cursor_row : callback:(logical:bool -> expand:bool -> all:bool -> bool) -> GtkS
method move_cursor : callback:(Gtk.Tags.movement_step -> int -> bool) -> GtkSignal.id
method row_activated : callback:(Gtk.tree_path -> view_column -> unit) -> GtkSignal.id
method row_collapsed : callback:(Gtk.tree_iter -> Gtk.tree_path -> unit) -> GtkSignal.id
method row_expanded : callback:(Gtk.tree_iter -> Gtk.tree_path -> unit) -> GtkSignal.id
method select_all : callback:(unit -> bool) -> GtkSignal.id
method select_cursor_parent : callback:(unit -> bool) -> GtkSignal.id
method select_cursor_row : callback:(start_editing:bool -> bool) -> GtkSignal.id
method set_scroll_adjustments : callback:(GData.adjustment option -> GData.adjustment option -> unit)
method start_interactive_search : callback:(unit -> bool) -> GtkSignal.id
method test_collapse_row : callback:(Gtk.tree_iter -> Gtk.tree_path -> bool) -> GtkSignal.id
method test_expand_row : callback:(Gtk.tree_iter -> Gtk.tree_path -> bool) -> GtkSignal.id
method toggle_cursor_row : callback:(unit -> bool) -> GtkSignal.id
method unselect_all : callback:(unit -> bool) -> GtkSignal.id
```

A.14.6. GTree.cell_renderer_text

```
inherits
 GObj.gtkobj_signals
method edited : callback:(Gtk.tree_path -> string -> unit) -> GtkSignal.id
```

A.14.7. GTree.cell_renderer_toggle

```
inherits
 GObj.gtkobj_signals
method toggled : callback:(Gtk.tree_path -> unit) -> GtkSignal.id
```

Appendix B. GDK Event Types

The following data types are passed into event handlers by GTK+. For each data type listed, the signals that use this data type are listed. (See `GObj.event_signals`)

- `Gdk.Tags.event_type` Gdk.event
  - any event

- ['DELETE] Gdk.event
  - delete event

- ['DESTROY] Gdk.event
  - destroy event

- ['MAP] Gdk.event
  - map event

- ['UNMAP] Gdk.event
  - unmap event

- `GdkEvent.Expose.t`
  - expose event

- `GdkEvent.Motion.t`
  - motion_notify event

- `GdkEvent.Button.t`
  - button_press event
  - button_release event

- `GdkEvent.Key.t`
  - key_press event
  - key_release event

- `GdkEvent.Crossing.t`
  - enter_notify event
  - leave_notify event

- `GdkEvent.Focus.t`
  - focus_in event
  - focus_out event

- `GdkEvent.Configure.t`
  - configure event

- `GdkEvent.Property.t`
  - property_notify event

- `GdkEvent.Selection.t`
  - selection_clear event
  - selection_request event
  - selection_notify event

- `GdkEvent.Proximity.t`
  - proximity_in event
  - proximity_out event

Appendix C. Code Examples

Below are the code examples that are used in the above text which are not included in complete form elsewhere.

C.1. Scribble

C.1.1. scribble.ml

```
(* file: scribble.ml *)

(* Backing pixmap for drawing area *)
let backing = ref (GDraw.pixmap ~width:200 ~height:200 ())

(* Create a new backing pixmap of the appropriate size *)
let configure window backing ev =
  let width = GdkEvent.Configure.width ev in
  let height = GdkEvent.Configure.height ev in
  let pixmap = GDraw.pixmap ~width ~height ~window () in
  pixmap#set_foreground `WHITE;
  pixmap#rectangle ~x:0 ~y:0 ~width ~height ~filled:true ();
  backing := pixmap;
  true

(* Redraw the screen from the backing pixmap *)
let expose (drawing_area:GMisc.drawing_area) (backing:GDraw.pixmap ref) ev =
  let area = GdkEvent.Expose.area ev in
  let x = Gdk.Rectangle.x area in
  let y = Gdk.Rectangle.y area in
  let width = Gdk.Rectangle.width area in
  let height = Gdk.Rectangle.width area in
  let drawing =
    drawing_area#misc#realize ();
    new GDraw.drawable (drawing_area#misc#window)
  in
  drawing#put_pixmap ~x ~y ~xsrc:x ~ysrc:y ~width ~height !backing#pixmap;
  false

(* Draw a rectangle on the screen *)
let draw_brush (area:GMisc.drawing_area) (backing:GDraw.pixmap ref) x y =
  let x = x - 5 in
  let y = y - 5 in
  let width = 10 in
  let height = 10 in
  let update_rect = Gdk.Rectangle.create ~x ~y ~width ~height in
  !backing#set_foreground `BLACK;
  !backing#rectangle ~x ~y ~width ~height ~filled:true ();
  area#misc#draw (Some update_rect)

let button_pressed area backing ev =
  if GdkEvent.Button.button ev = 1 then (
    let x = int_of_float (GdkEvent.Button.x ev) in
    let y = int_of_float (GdkEvent.Button.y ev) in
    draw_brush area backing x y
  );
  true

let motion_notify area backing ev =
  let (x, y) =
    if GdkEvent.Motion.is_hint ev
 then area#misc#pointer
 else
      (int_of_float (GdkEvent.Motion.x ev), int_of_float (GdkEvent.Motion.y ev))
  in
  let state = GdkEvent.Motion.state ev in
  if Gdk.Convert.test_modifier `BUTTON1 state
  then draw_brush area backing x y;
  true
```

```
let main () =
  let width = 200 in
  let height = 200 in

  let window = GWindow.window ~title:"Scribble" () in
  window#connect#destroy ~callback:GMain.Main.quit;

  let vbox = GPack.vbox ~packing:window#add () in

  (* Create the drawing area *)
  let area = GMisc.drawing_area ~width ~height ~packing:vbox#add () in

  (* Signals used to handle backing pixmap *)
  area#event#connect#expose ~callback:(expose area backing);
  area#event#connect#configure ~callback:(configure window backing);

  (* Event signals *)
  area#event#connect#motion_notify ~callback:(motion_notify area backing);
  area#event#connect#button_press ~callback:(button_pressed area backing);

  area#event#add [‘EXPOSURE; ‘LEAVE_NOTIFY; ‘BUTTON_PRESS; ‘POINTER_MOTION; ‘POINTER_MOTION_HINT];

  (* .. And a quit button *)
  let button = GButton.button ~label:"Quit" ~packing:vbox#add () in
  button#connect#clicked ~callback:window#destroy;

  window#show ();
  GMain.Main.main ()

let _ = Printexc.print main ()
```