GtkAda User's Guide

Version 2.4.0
Document revision level 1.123
Date: 2004/07/15 22:23:44

E. Briot, J. Brobecker, A. Charlet

1 Introduction: What is GtkAda ?

GtkAda is a high-level portable graphical toolkit, based on the gtk+ toolkit, one of the official GNU toolkits. It makes it easy to create portable user interfaces for multiple platforms, including most platforms that have a X11 server and Win32 platforms.

Although it is based on a C library, GtkAda uses some advanced Ada95 features like tagged types, generic packages, access to subprograms, exceptions, etc... to make it easier to use and design interfaces. For efficiency reasons, it does not use controlled types, but takes care of all the memory management for you in other ways.

As a result, this library provides a *secure*, *easy to use* and *extensible* toolkit.

Compared to the C library, GtkAda provides type safety (especially in the callbacks area), and object-oriented programming. As opposed to common knowledge, it requires *less* type casting than with in C. Its efficiency is about the same as the C library through the use of inline subprograms.

GtkAda comes with a complete integration to the graphical interface builder `Glade`[1]. This makes it even easier to develop interfaces, since you just have to click to create a window and all the dialogs. Ada code can then be generated with a single click.

GtkAda also has a full support for OpenGL, and comes with a very thin binding to the OpenGL (`Mesa`) libraries. You can thus create graphical applications that display 3D graphics, and display them in a GtkAda window, as with any other 2D graphics. This manual does not document OpenGL at all, see any book on OpenGL, or the specification that came with your OpenGL library, for more information.

The following Internet sites will always contain the latest public packages for `GtkAda`, `gtk+` and `Glade`.

`http://libre.act-europe.fr/GtkAda`

`http://www.gtk.org`

`http://glade.gnome.org`

The scheme used for GtkAda's version numbers is the following: the major version number is the same as for the underlying gtk+ library (e.g 1.2 or 1.3). Thus, stable versions of GtkAda will have even major numbers, and development versions (potentially less stable, although we are trying to make sure that no big problem exists in those versions) will have odd version numbers. The minor version numbers depends on GtkAda's release number.

This documentation is distributed with GtkAda **version 2.4.0**. At this stage, every widget found in gtk+ 2.2 has been implemented, and the test program found in the gtk release has been fully coded in Ada (have a look at the `testgtk/` directory in the distribution).

This toolkit was tested on the following systems:

- GNU Linux/x86
- Solaris/sparc
- Tru64
- HP/UX

[1] Glade was written by Damon Chaplin

- NT 4.0
- Windows 2000
- Windows XP

with the latest version of the `GNAT` compiler, developed and supported by Ada Core Technologies (see `http://www.gnat.com`).

If you manage to use it on other systems (which should probably be straightforward - just recompile GtkAda), please let us know so that we can add to the above list.

This version of GtkAda is known to be compatible with `gtk+` **2.2.1** through **2.4.4**. This release is no longer compatible with older versions of gtk+.

This version of GtkAda is compatible with `Glade` **version 2.0.0**. Due to some modification in the output format of Glade, this release will not work with older versions. It is also not guaranteed to work with more recent versions.

This document does not describe all the widgets available in GtkAda, nor does it try to explain all the subprograms. The GtkAda Reference Manual provides this documentation instead, as well as the GtkAda sources spec files themselves, whose extension is '`.ads`'.

No complete example is provided in this documentation. Instead, please refer to the examples that you can find in the '`testgtk/`' and '`examples/`' directory in the GtkAda distribution, since these are more up-to-date (and more extensive). They are heavily commented, and are likely to contain a lot of information that you might find interesting.

If you are interested in getting support for GtkAda (including priority bug fixes, early releases, help in using the toolkit, help in designing your interface, on site consulting...), please contact ACT Europe (`mailto:sales@act-europe.fr`) or Ada Core Technologies (`mailto:sales@gnat.com`).

2 Getting started with GtkAda

This chapter describes how to start a new GtkAda application. It explains the basic features of the toolkit, and shows how to compile and run your application.

It also gives a brief overview of the extensive widget hierarchy available in GtkAda.

2.1 How to build and install GtkAda

This section explains how to build and install GtkAda on your machine. It is Unix-oriented, since GtkAda is distributed in binary format on Windows machines, and comes with all the dependent packages, including the gtk+ libraries and `Glade`. If you are a Windows-user, you should skip this section.

On Unix systems, you first need to install the glib and gtk+ libraries. Download the compatible packages from the gtk+ web site (`http://www.gtk.org`, compile and install it. When compiling glib, make sure that the support for multithreading is enabled (this is the default, and can be enforced with the `--enable-threads` and `--with-threads` command line switches.

Change your PATH environment variable so that the script `gtk-config`, which indicates where gtk+ was installed and what libraries it needs is automatically found by GtkAda. You will no longer need this script once GtkAda is installed, unless you develop part of your application in C.

OpenGL support will not be activated in GtkAda unless you already have the OpenGL libraries on your systems. You can for instance look at Mesa, which is free implementation.

Optionally, you can also install the `Glade` interface builder. Get the compatible package from the Glade web site, compile and install it. The official version already knows about Ada (at least enough to call GtkAda's own programs), so no patch is needed.

You can finally download the latest version of GtkAda from the web site. Untar and uncompress the package, then simply do the following steps:

```
> ./configure
> make install
```

As usual with the `configure` script, you can specify where you want to install the GtkAda libraries by using the `--prefix` switch.

If you have some OpenGL libraries installed on your system, you can make sure that `configure` finds them by specifying the `--with-GL-prefix` switch on the command line. `configure` should be able to automatically detect the libraries however.

You must then make sure that the system will be able to find the dynamic libraries at run time if your application uses them. Typically, you would do one of the following:

- run `ldconfig` if you installed GtkAda in one of the standard location and you are super-user on your machine
- edit `/etc/ld.conf` if you are super-user but did not install GtkAda in one of the standard location. Add the path that contains libgtkada.so (by default '`/usr/local/lib`' or '`$prefix/lib`'.
- modify your `LD_LIBRARY_PATH` environment variable if you are not super-user. You should simply add the path to libgtkada.

2.2 Organization of the GtkAda package

In addition to the full sources, the GtkAda package contains a lot of heavily commented examples. If you haven't been through those examples, we really recommend that you look at them and try to understand them, since they contain some examples of code that you might find interesting for your own application.

- '`testgtk/`' directory:

  This directory contains an application that tests all the widgets in GtkAda. It gives you a quick overview of what can be found in the toolkit, as well as some detailed information on the widgets and their parameters.

  Each demo is associated with contextual help pointing to aspects worth studying.

  It also contains an OpenGL demo, if GtkAda was compiled with support for OpenGL.

  This program is far more extensive that its C counterpart, and the GtkAda team has added a lot of new examples.
- '`examples/`' directory:

  This directory contains some small examples, unrelated to testgtk. For instance, this is where you will find some sample XML files for `Gate` and `Dgate`, as well as some new widgets created directly in Ada, as examples of how to create your own callback marshallers.

  On the whole these examples are a little more complex than testgtk but, since they focus on demonstrating a precise concept, they are still quite easy to understand.
- '`docs/`' directory:

  It contains the html, info, text and TeX versions of the documentation you are currently reading. Note that the documentation is divided into two subdirectories, one containing the user guide, which you are currently reading, the other containing the reference manual, which gives detailed information on all the widgets found in GtkAda. The docs directory also contains a subdirectory with some slides that were used to present GtkAda at various shows.

2.3 How to compile an application with GtkAda

This section explains how you can compile your own applications. The procedure is system-dependent, and thus is divided into two subsections.

2.3.1 Unix systems

On Unix systems, a script called `gtkada-config` is automatically created when you build GtkAda. This script is copied in a subdirectory '`bin/`' in the installation directory.

The easiest and recommended way to build a GtkAda application is to use the `gnatmake` program distributed with GNAT, that takes care of all the dependencies for you. Use the `gtkada-config` to specify where GtkAda and gtk+ libraries have been installed.

```
> gnatmake <main-file> `gtkada-config`
```

Note the use of back-ticks around gtkada-config, which force the shell to evaluate the script and put the output on the command line.

However, on complex systems, gnatmake might not be enough. Users frequently like to create `Makefiles`. The script `gtkada-config` remains useful in that case, since you can call it from your Makefile (same syntax as above with the back-ticks) to create variables like FLAGS and LIBS. See the switches of `gtkada-config` below for more information.

The script `gtkada-config` understands the following command line switches (chosen to be compatible with the ones set by `gtk-config`):

- `--cflags`: Output only the compiler flags, i.e the include directories where the GtkAda spec files are found. This should be used if you only want to compile your files, but do not want to bind or link them.
- `--libs`: Output only the switches for the linker. This lists the directories where all the GtkAda, gtk+, and dependant libraries are found. For instance, if GtkAda was compiled with support for OpenGL, the OpenGL libraries will automatically be present.
- `--static`: Forces linking with the static gtkada library. This option will still use the dynamic gtk+ libraries.

2.3.2 Windows systems

Things are somewhat easier on Windows systems. You don't have access to the `gtkada-config` script. On the other hang you also don't have to specify which libraries to use or where to find them.

The only thing you should specify on the `gnatmake` command line is where the GtkAda spec files are found, as in:

```
> gnatmake <main-file> -Ic:\gtkada\include\gtkada
```

if GtkAda was installed under '`c:\gtkada`'.

2.4 Architecture of the toolkit

The gtk+ toolkit has been designed from the beginning to be portable. It is made of three libraries, `gtk`, `gdk` and `glib`.

`Glib` is a non-graphical library, that includes support for lists, h-tables, threads, etc... It is a highly optimized, platform-independent library. Since most of its contents are already available in Ada (or in the '`GNAT.*`' hierarchy in the GNAT distribution), GtkAda does not include a binding to it, except for a few required packages. These are the '`Glib.*`' packages in the GtkAda distribution.

`Gdk` is the platform-dependent part of gtk+. Its implementation is completely different on win32 systems and X11 systems, although the interface is of course the same. It provides a set of functions to draw lines, rectangles and pixmaps on the screen, manipulate colors, etc... It has a complete equivalent in GtkAda, through the '`Gdk.*`' packages.

`Gtk` is the top level library. It is platform independent, and does all its drawing through calls to Gdk. This is where the high-level widgets are defined. It also includes support for callbacks. Its equivalent in the GtkAda libraries are the '`Gtk.*`' packages. It is made of a fully object-oriented hierarchy of widgets (see Section 2.5 [Widgets Hierarchy], page 7).

Since your application only calls GtkAda, it is fully portable, and can be recompiled as-is on other platforms.

```
+---------------------------------------------+
|             Your Application                |
+---------------------------------------------+
|                 GtkAda                      |
|               +-----------------------------+
|               |          GTK                |
|       +-------+-----------------------------+
|       |           GDK                       |
+-------+--------------+--+-------------------+
|          GLIB        |  | X-Window / Win32  |
+----------------------+  +-------------------+
```

Although the packages have been evolving a lot since the first versions of GtkAda, the specs are stabilizing now. We will try as much as possible to provide backward compatibility whenever possible.

Since GtkAda is based on gtk+ we have tried to stay as close to it as possible while using high-level features of the Ada95 language. It is thus relatively easy to convert external examples from C to Ada.

We have tried to adopt a consistent naming scheme for Ada identifiers:

- The widget names are the same as in C, except that an underscore sign (_) is used to separate words, e.g

  `Gtk_Button   Gtk_Color_Selection_Dialog`

- Because of a clash between Ada keywords and widget names, there are two exceptions to the above general rule:

  `Gtk.GEntry.Gtk_Entry   Gtk.GRange.Gtk_Range`

- The function names are the same as in C, ignoring the leading `gtk_` and the widget name, e.g

  `gtk_misc_set_padding` ⇒ `Gtk.Misc.Set_Padding`
  `gtk_toggle_button_set_state` ⇒ `Gtk.Toggle_Button.Set_State`

- Most enum types have been grouped in the '`gtk-enums.ads`' file
- Some features have been implemented as generic packages. These are the timeout functions (see Gtk.Main.Timeout), the idle functions (see Gtk.Main.Idle), and the data that can be attached to any object (see Gtk.Object.User_Data). Type safety is ensured through these generic packages.
- Callbacks were the most difficult thing to interface with. These are extremely powerful and versatile, since the callbacks can have any number of arguments, can return values, or not,etc... These are once again implemented as generic packages, that require more explanation (see Chapter 4 [Signal handling], page 11).

WARNING: all the generic packages allocate some memory for internal structures, and call internal functions. This memory is freed by gtk itself, by calling some Ada functions. Therefore the generic packages have to be instanciated at library level, not inside a subprogram, so that the functions are still defined when gtk needs to free the memory.

WARNING Before any other call to the GtkAda library is performed, `Gtk.Main.Init` must be invoked first. Most of the time, this procedure is invoked from the main procedure of the application, in which case no use of GtkAda can be done during the application elaboration.

2.5 Widgets Hierarchy

All widgets in `GtkAda` are implemented as tagged types. They all have a common ancestor, called `Gtk.Object.Gtk_Object`. All visual objects have a common ancestor called `Gtk.Widget.Gtk_Widget`.

The following table describes the list of objects and their inheritance tree. As usual with tagged types, all the primitive subprograms defined for a type are also known for all of its children. This is a very powerful way to create new widgets, as will be explained in Section 9.3 [Creating new widgets in Ada], page 23.

Although gtk+ was written in C its design is object-oriented, and thus GtkAda has the same structure. The following rules have been applied to convert from C names to Ada names: a widget `Gtk_XXX` is defined in the Ada package `Gtk.XXX`, in the file '`gtk-xxx.ads`'. This follows the GNAT convention for file names. For instance, the `Gtk_Text` widget is defined in the package `Gtk.Text`, in the file '`gtk-text.ads`'.

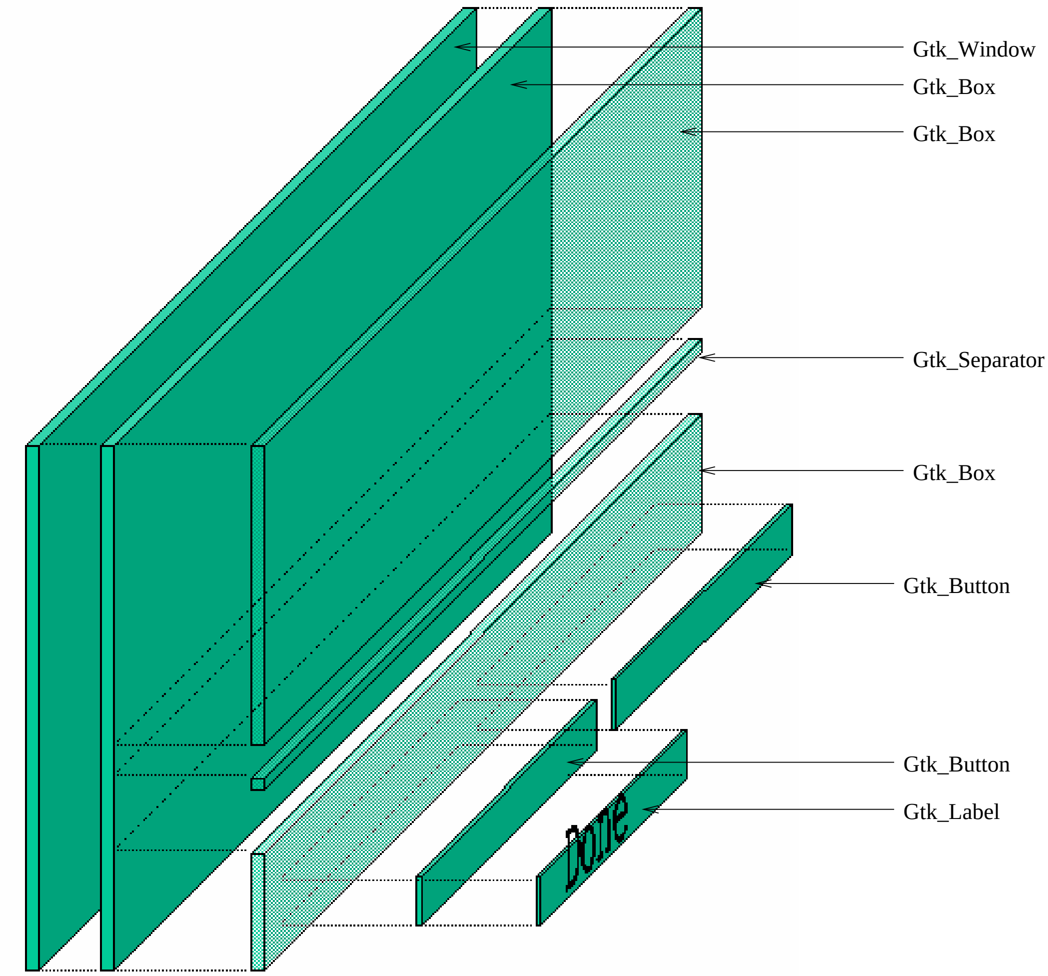
Note also that most of the documentation for GtkAda is found in the spec files themselves.

It is important to be familiar with this hierarchy. It is then easier to know how to build and organize your windows. Most widgets are demonstrated in the '`testgtk/`' directory in the GtkAda distribution.

Hierarchy of widgets in GtkAda

3 Hierarchical composition of a window

Interfaces in GtkAda are built in layers, as in Motif. For instance, a typical dialog is basically a Gtk_Window, that in turn contains a Gtk_Box, itself divided into two boxes and a Gtk_Separator, and so on.



Altough this may seem more complicated than setting absolute positions for children, this is the simplest way to automatically handle the resizing of windows. Each container that creates a layer knows how it should behave when it is resized, and how it should move its children. Thus almost everything is handled automatically, and you don't have to do anything to support resizing.

If you really insist on moving the children to a specific position, look at the `Gtk_Fixed` widget and its demo in '`testgtk/`'. But you really should not use this container, since you will then have to do everything by hand.

All the containers are demonstrated in '`testgtk/`', in the GtkAda distribution. This should help you understand all the parameters associated with the containers. It is very important to master these containers, since using the appropriate containers will make building interfaces a lot easier.

If you look at the widget hierarchy (see Section 2.5 [Widgets Hierarchy], page 7), you can see that a Gtk_Window inherits from Gtk_Bin, and thus can have only one child. In most cases, the child of a Gtk_Window will thus be a Gtk_Box, which can have any number of children.

Some widgets in GtkAda itself are built using this strategy, from the very basic `Gtk_Button` to the more advanced `Gtk_File_Selection`.

For example, by default a Gtk_Button contains a Gtk_Label, which displays the text of the button (like "OK" or "Cancel").

However, it is easy to put a pixmap in a button instead. When you create the button, do not specify any label. Thus, no child will be added, and you can give it your own. See '`testgtk/create_pixmap.adb`' for an example on how to do that.

4 Signal handling

In GtkAda, the interaction between the interface and the core application is done via signals. Most user actions on the graphical application trigger some signals to be '`emitted`'.

A signal is a message that an object wants to broadcast. It is identified by its name, and each one is associated with certain events which happen during the widget's lifetime. For instance, when the user clicks on a Gtk_Button, a "clicked" signal is emitted by that button. More examples of signals can be found in the GtkAda reference manual.

It is possible to cause the application to react to such events by '`connecting`' to a signal a special procedure called a '`handler`' or '`callback`'. This handler will be called every time that signal is emitted, giving the application a chance to do any processing it needs. More than one handler can be connected to the same signal on the same object; the handlers are invoked in the order they were connected.

4.1 Predefined signals

Widgets, depending on their type, may define zero or more different signals. The signals defined for the parent widget are also automatically inherited; thus every widget answers many signals.

The easiest way to find out which signals can be emitted by a widget is to look at the GtkAda reference manual. Every widget will be documented there. The GtkAda RM explains when particular signals are emitted, and the general form that their handlers should have (although you can always add a `User_Data` if you wish, see below).

You can also look directly at the C header files distributed with the gtk+ library. Each widget is described in its own C file and has two C structures associated with it. One of them is the "class" structure, which contains a series of pointers to functions. Each of these functions has the same name as the signal name.

For instance, consider the following extract from gtkbutton.h:

```
struct _GtkButtonClass
{
  GtkBinClass        parent_class;

  void (* pressed)  (GtkButton *button);
  void (* released) (GtkButton *button);
  void (* clicked)  (GtkButton *button);
  void (* enter)    (GtkButton *button);
  void (* leave)    (GtkButton *button);
};
```

This means that the Gtk_Button widget redefines five new signals, respectively called `"pressed"`, `"released"`, ...

The profile of the handler can also be deduced from those pointers: The handler has the same arguments, plus an optional `User_Data` parameter that can be used to pass any kind of data to the handler. When the `User_Data` parameter is used, the value of this data is specified when connecting the handler to the signal. It is then given back to the handler when the signal is raised.

Therefore, the profile of a handler should look like:

```
procedure Pressed_Handler
  (Button    : access Gtk_Button_Record'Class;
   User_Data : ...);
```

The callback does not need to use all the arguments. It is legal to use a procedure that "drops" some of the last arguments. There is one special case, however: if, at connection time, you decided to use `User_Data`, your callback must handle it. This is checked by the compiler.

Any number of arguments can be dropped as long as those arguments are the last ones in the list and you keep the first one. For instance, the signal "button_press_event" normally can be connected to a handler with any of the following profiles:

```
--  with a user_data argument
procedure Handler
  (Widget    : access Gtk_Widget_Record'Class;
   Event     : Gdk.Event.Gdk_Event;
   User_Data : ...);
procedure Handler
  (Widget    : access Gtk_Widget_Record'Class;
   User_Data : ...);

--  without a user_data argument
procedure Handler
  (Widget : access Gtk_Widget_Record'Class;
   Event  : Gdk.Event.Gdk_Event);
procedure Handler (Widget : access Gtk_Widget_Record'Class);
```

Beware that adding new arguments is not possible, since no value would be provided for them. When connecting a handler, GtkAda will not always verify that your handler does not have more arguments than expected, so caution is recommended (it only does so if you use the `Gtk.Marshallers` package, see below).

4.2 Connecting signals

All the signal handling work is performed by using the services provided by the `Gtk.Handlers` package. This package is self-documented, so please read the documentation for this package either in the GtkAda Reference Manual or in the specs themselves. The rest of this section assumes that you have this documentation handy.

A short, annotated example of connecting signals follows; a complete example can be found in create_file_selection.adb (inside the testgtk/ directory). In our example, an application opens a file selector to allow the user to select a file. GtkAda provides a high-level widget called Gtk_File_Selection which can be used in this case:

```
declare
   Window : Gtk_File_Selection;
begin
   Gtk.File_Selection.Gtk_New (Window, Title => "Select a file");
end;
```

When the "OK" button is pressed, the application needs to retrieve the selected file and then close the dialog. The only information that the handler for the button press needs is which widget to operate upon. This can be achieved by the following handler:

```
procedure OK (Files : access Gtk_File_Selection_Record'Class) is
begin
   Ada.Text_IO.Put_Line ("Selected " & Get_Filename (Files));
   --  Prints the name of the selected file.
   Destroy (Files);
   --  Destroys the file selector dialog
end Ok;
```

We now need to connect the object we created in the first part with the new callback we just defined. `Gtk.Handlers` defines four types of generic packages, depending on the arguments one expects in the callback and whether the callback returns a value or not. Note that you can not use an arbitrary list of arguments; this depends on the signal, as explained in the previous section.

In our example, since the callback does not return any value and does not handle any `User_Data` (that is, we don't pass it extra data, which will be specified at connection time), the appropriate package to use is `Gtk.Handlers.Callback`. We thus instantiate that package.

Remember that generic package instantiations in GtkAda must be present in memory at all times, since they take care of freeing allocated memory when finished. GtkAda generic package instantiations must therefore always be performed at the library level, and not inside any inner block.

```
package Files_Cb is new
  Handlers.Callback (Gtk_File_Selection_Record);
```

The `Files_Cb` package now provides a set of Connect subprograms that can be used to establish a tie between a widget and a handler. It also provides a set of other subprograms which you can use to emit the signals manually, although most of the time, the signals are simply emitted internally by GtkAda. We will not discuss the Emit_By_Name subprograms here.

The general form of handler, as used in `Gtk.Handlers`, expects some handlers that take two or three arguments: the widget on which the signal was applied, an array of all the extra arguments sent internally by GtkAda, and possibly some user data given when the connection was made.

This is the most general form of handler and it covers all the possible cases. However, it also expects the user to manually extract the needed values from the array of arguments. This is not always the most convenient solution. This is why GtkAda provides a second package related to signals, `Gtk.Marshallers`.

The `Gtk.Marshallers` package provides a set of functions that can be used as callbacks directly for GtkAda, and that will call your application's handlers after extracting the required values from the array of arguments. Although this might sound somewhat complicated, in practice it simplifies the task of connecting signals. In fact, the techniques employed are similar to what is done internally by gtk+ in C. Because of the similarity of techniques, there is no overhead involved in using `Gtk.Marshallers` with Ada over the C code in gtk+.

A set of functions `To_Marshaller` is found in every generic package in `Gtk.Handlers`. They each take a single argument, the name of the function you want to call, and return a handler that can be used directly in `Connect`.

The connection is then done with the following piece of code. Note that this can be done just after creating the widget, in the same block. As soon as it is created, a widget is ready to accept connections (although no signals will be emitted before the widget is shown on the screen).

Note that we use `To_Marshaller` since our handler does not accept the array of arguments as a parameter, and we use the special `Object_Connect` procedure. This means that the parameter to our callback (Files) will be the Slot_Object given in Object_Connect, instead of being the button itself.

```
Files_Cb.Object_Connect
  (Get_Ok_Button (Window),  --  The object to connect to the handler
   "clicked",               --  The name of the signal
   Files_Cb.To_Marshaller (Ok'Access),  --  The signal handler
   Slot_Object => Window);
```

4.3 Handling user data

As described above, it is possible to define some data that is that passed to the callback when it is called. This data is called user_data, and is passed to the `Connect` or `Object_Connect` subprograms.

GtkAda will automatically free any memory it has allocated internally to store these user data. For instance, if you instanciated the generic package `User_Callback` with a String, it means that you want to be able to have a callback of the form:

```
procedure My_Callback (Widget : access Gtk_Widget_Record'Class;
                       User_Data : String);
```

and connect it with a call similar to:

```
Connect (Button, "Clicked", To_Marshaller (My_Callback'Access),
         User_Data => "any string");
```

GtkAda needs to allocate some memory to store the string (an unconstrained type). However, this memory is automatically freed when the callback is destroyed.

There are a few subtleties in the use of user_data, most importantly when the user data is itself a widget.

The following four examples do exactly the same thing, ie create two buttons, and clicking on the first one will destroy the second one. They all work fine the first time, while the two buttons exist. However, some of them will fail if you press on the first button a second time.

The code for this example can be found in the distribution, in the '`examples/user_data`' directory. The examples below do not include the creation of the main window, or of the buttons themselves, to emphasize the important part.

4.3.1 First case: simple user data

This code will fail: when `Button2` has been destroyed, the Ada type still points to some random memory, and the second call to `Destroy` will fail with a Storage_Error.

```
package User_Callback is new Gtk.Handlers.User_Callback
  (Gtk_Widget_Record, Gtk_Widget);

procedure My_Destroy2
  (Button : access Gtk_Widget_Record'Class; Data : Gtk_Widget) is
begin
   Destroy (Data);
end My_Destroy2;

begin
   User_Callback.Connect
     (Button1, "clicked",
      User_Callback.To_Marshaller (My_Destroy2'Access),
      Gtk_Widget (Button2));
end;
```

4.3.2 Second case: using Object_Connect instead

One of the solutions to fix the above problem is to use `Object_Connect` instead of `Connect`. In that case, GtkAda automatically takes care of disconnecting the callback when either of the two widgets is destroyed.

```
procedure My_Destroy (Button : access Gtk_Widget_Record'Class) is
begin
   Destroy (Button);
end My_Destroy;

begin
   Widget_Callback.Object_Connect
     (Button1, "clicked",
      Widget_Callback.To_Marshaller (My_Destroy'Access),
      Button2);
end;
```

4.3.3 Third case: manually disconnecting the callback

Using `Object_Connect` is not always possible. In that case, one of the possibilities is to store the `Id` of the callback, and properly disconnect it when appropriate. This is the most complex method, and very often is not applicable, since you cannot know for sure when the callback is no longer needed.

```
type My_Data3 is record
   Button, Object : Gtk_Widget;
   Id             : Handler_Id;
end record;
type My_Data3_Access is access My_Data3;
```

```
package User_Callback3 is new Gtk.Handlers.User_Callback
  (Gtk_Widget_Record, My_Data3_Access);

procedure My_Destroy3
  (Button : access Gtk_Widget_Record'Class;
   Data   : My_Data3_Access) is
begin
   Destroy (Data.Button);
   Disconnect (Data.Object, Data.Id);
end My_Destroy3;

   Id : Handler_Id;
begin
   Data3 := new My_Data3' (Object => Gtk_Widget (Button1),
                           Button => Gtk_Widget (Button2),
                           Id     => (Null_Signal_Id, null));
   Id := User_Callback3.Connect
     (Button1, "clicked",
      User_Callback3.To_Marshaller (My_Destroy3'Access),
      Data3);
   Data3.Id := Id;
end;
```

4.3.4 Fourth case: setting a watch on a specific widget

GtkAda provides a function `Add_Watch`, that will automatically disconnect a callback when a given widget is destroyed. This is the function used internally by `Object_Connect`. In the example below, the callback is automatically disconnected whenever `Button2` is destroyed.

```
procedure My_Destroy2
  (Button : access Gtk_Widget_Record'Class; Data : Gtk_Widget) is
begin
   Destroy (Data);
end My_Destroy2;

   Id : Handler_Id;
begin
   Id := User_Callback.Connect
     (Button1, "clicked",
      User_Callback.To_Marshaller (My_Destroy2'Access),
      Gtk_Widget (Button2));
   Add_Watch (Id, Button2);
end;
```

5 Starting an application with GtkAda

You need to perform some initializations to start a GtkAda application:

```
--  predefined units of the library
with Gtk.Rc;
with Gtk.Main;
with Gtk.Enums;
with Gtk.Window;
...
--  My units
with Callbacks;
...
procedure Application is
   procedure Create_Window is ...

begin
   --  Set the locale specific datas (e.g time and date format)
   Gtk.Main.Set_Locale;

   --  Initializes GtkAda
   Gtk.Main.Init;

   --  Load the resources. Note that this part is optional.
   Gtk.Rc.Parse ("application.rc");

   --  Create the main window
   Create_Window;

   --  Signal handling loop
   Gtk.Main.Main;
end Application;
```

the `Create_Window` procedure looks like

```
   procedure Create_Window is
      Main_Window : Gtk.Window.Gtk_Window;
      ...
   begin
      Gtk.Window.Gtk_New
        (Window   => Main_Window,
         The_Type => Gtk.Enums.Window_Toplevel);

      --  From Gtk.Widget:
      Gtk.Window.Set_Title (Window => Main_Window, Title  => "Editor");

      --  Construct the window and connect various callbacks

      ...
      Gtk.Window.Show_All (Main_Window);
   end Create_Window;
```

6 Resource files

Resource files let you parametrize aspects of the widgets in a GtkAda application without having to recompile it.

A resource file needs to be loaded (`Gtk.Rc.Parse`) *before* setting the corresponding window.

In this file, it is possible to specify the visual characteristics of the widgets (colors, fonts, ...). Under X, the `xfontsel` command allows you to easily select a font. The FontSelection widget is also a simple way to select fonts.

Here is an example of a resource file:

```
# application.rc
#
# resource file for "Application"

# Buttons style
style "button"
{
# BackGround Colors
#                  Red  Green  Blue
  bg[PRELIGHT] = { 0.0,  0.75, 0.0 } # Green when the mouse is on
                                     # the button
  bg[ACTIVE]   = { 0.75, 0.0,  0.0 } # Red on click
# ForeGround Colors
#                  Red  Green  Blue
  fg[PRELIGHT] = { 1.0,  1.0,  1.0 } # White when the mouse is on
                                     # the button
  fg[ACTIVE]   = { 1.0,  1.0,  1.0 } # White on click
}

# All the buttons will have the style "button"
widget_class "*GtkButton*" style "button"

# Text style
style "text"
{
  font = "-adobe-courier-medium-r-normal-*-15-*-*-*-*-*-*-*"
  text[NORMAL] = { 0.0, 0.0, 0.0 } # black
  fg[NORMAL]   = { 0.0, 0.0, 0.0 } # black
  base[NORMAL] = { 1.0, 1.0, 1.0 } # white : background color
}

# All Gtk_Text will have the "text" style
widget_class "*GtkText" style "text"
```

7 Memory management

GtkAda takes care of almost all the memory management for you. Here is a brief overview of how this works, you'll have to check the sources if you want more detailed information. Gtk+ (the C library) does its own memory management through reference counting, i.e. any widget is destroyed when it is no longer referenced anywhere in the application.

In GtkAda itself, a "user_data" is associated with each object allocated by a `Gtk_New` procedure. A "destroy" callback is also associated, to be called when the object to which the user_data belongs is destroyed. Thus, every time a C object is destroyed, the equivalent Ada structure is also destroyed (see Gtk.Free_User_Data).

Concerning widgets containing children, every container holds a reference to its children, whose reference counting is thus different from 0 (and generally 1). When the container is destroyed, the reference of all its children and grand-children is decremented, and they are destroyed in turn if needed. So the deallocation of a widget hierarchy is also performed automatically.

8 Tasking with GtkAda

Note that Gtk+ under Windows does not interact properly with threads, so the only safe approach under this operating system is to perform all your Gtk+ calls in the same task.

Under other platforms, the Glib library can be used in a task-safe mode by calling Gdk.Thread.G_Init and Gdk.Thread.Init before making any other Glib/Gdk calls. In this mode Gdk automatically locks all internal data structures as needed. This does not mean that two tasks can simultaneously access, for example, a single hash table, but they can access two different hash tables simultaneously. If two different tasks need to access the same hash table, the application is responsible for locking itself (e.g by using protected objects).

When Gdk is initialized to be task-safe, GtkAda is task aware. There is a single global lock that you must acquire with Gdk.Threads.Enter before making any Gdk/Gtk call, and which you must release with Gdk.Threads.Leave afterwards.

Thus, Gtk.Main.Main should be called with the lock acquired (see example below), ensuring that all the functions executed in the task that started the main loop do not need to protect themselves again.

Beware that the GtkAda main loop (Gtk.Main.Main) can only be be run inside one specific task. In other words, you cannot call Gtk.Main.Main from any task other than the one that started the outer level main loop.

Note that Gdk.Threads assumes that you are using a tasking run time that maps Ada tasks to native threads.

A minimal main program for a tasking GtkAda application looks like:

```
with Gdk.Threads;
with Gtk.Main;
with Gtk.Enums; use Gtk.Enums;
with Gtk.Window; use Gtk.Window;

procedure GtkAda_With_Tasks is
   Window : Gtk_Window;
begin
   Gdk.Threads.G_Init;
   Gdk.Threads.Init;
   Gtk.Main.Init;

   Gtk_New (Window, Window_Toplevel);
   Show (Window);

   Gdk.Threads.Enter;
   Gtk.Main.Main;
   Gdk.Threads.Leave;
end GtkAda_With_Tasks;
```

Callbacks require a bit of attention. Callbacks from GtkAda (signals) are made within the GtkAda lock. However, callbacks from Glib (timeouts, IO callbacks, and idle functions) are made outside of the GtkAda lock. So, within a signal handler you do not need to call Gdk.Threads.Enter, but within the other types of callbacks, you do.

9 Object-oriented features

GtkAda has been designed from the beginning to provide a full Object oriented layer over gtk+. This means that features such as type extension, dynamic dispatching, ... are made available through the standard Ada language.

This section will describe both how things work and how you can extend existing widgets or even create your own.

9.1 General description of the tagged types

9.1.1 Why should I use object-oriented programing ?

Every widget in GtkAda is a tagged type and has a number of primitive subprograms which all its children inherit.

This means that most of the time, as opposed to what you see in C code, you don't have to explicitly cast types and, even when you have to, Ada always makes sure that the conversion is valid.

Thus your programs are much safer and most errors are found at compile time, as usual with Ada.

For instance, if you create a table, put some widgets in it, and then, later in your program, try to access those widgets, then you do not need to know beforehand what their type is, when and by whom they were created, ... You simply ask for the children of the table, and you get in return a tagged type that contains all the information you need. You can even use dynamic dispatching without ever having to cast to a known type.

This makes GtkAda a very powerful tool for designing graphical interfaces.

If you think one of the standard widgets is nice, but would be even better if it was drawing itself in a slighlty different way, or if it could contain some other data that you need in your application, there is a very simple way to do it: just create a new type that extends the current one (see the section Section 9.2 [Using tagged types to extend Gtk widgets], page 23 below.

Maybe you want to create your own brand new widget, that knows how to draw itself, how to react to events, ... and you want to be able to reuse it anytime you need ? Once again, using the standard Ada features, you can simply create a new tagged type and teach it how to interact with the user. See the section Section 9.3 [Creating new widgets in Ada], page 23 below.

9.1.2 Type conversions from C to Ada widgets

There are basically three kinds of widgets that you can use with GtkAda:

- *Ada widgets*: These are widgets that are written directly in Ada, using the object oriented features of GtkAda
- *Standard widgets*: These are the widgets that are part of the standard gtk+ and GtkAda distributions. This include all the basic widgets you need to build advanced interfaces.

- *third party C widgets* These are widgets that were created in C, and for which you (or someone else) created an Ada binding. This is most probably the kind of widgets you will have if you want to use third party widgets.

GtkAda will always be able to find and/or create a valid tagged type in the first case, no matter if you explicitly created the widget or if it was created automatically by gtk+. For instance, if you created a widget in Ada, put it in a table, and later on extracted it from the table, then you will still have the same widget.

There are two issues: if the widget was explictly created by you, or at least by GtkAda, then it will always be and remain associated with a correct Ada type.

If the widget was created implicitly (for instance every time you create a Gtk_Button, a Gtk_Label is also created for the text displayed), then GtkAda will only be able to create the corresponding type by default for the following widgets: Gtk_Label, Gtk_Button, Gtk_Item, Gtk_List_Item, Gtk_Menu_Item, Gtk_Check_Menu_Item, Gtk_Radio_Menu_Item, Gtk_Tearoff_Menu_Item, Gtk_Tree_Item and Gtk_Entry. For other widgets, it will instead create a Gtk_Widget, and you will have to either call Gtk.Unchecked_Cast to convert it back to the type you expect, or use **Gtk.Type_Conversion** as described below. Here is an example of use of Gtk.Unchecked_Cast:

```
declare
   Stub : Gtk_Window_Record;
begin
   Window := Gtk.Unchecked_Cast (Widget, Stub);
end;
```

In the third case (third party C widgets), GtkAda is not, by default, able to create the corresponding Ada type.

The solution we suggest to solve the first issue is to 'with' the **Gtk.Type_Conversion** unit. In that case, every standard widget, no matter who created them, will always be correctly converted to an appropriate Ada type. So, basically, if you put the following in your main unit:

```
with Gtk.Type_Conversion;

begin
   Gtk.Main.Init;
   ...
end
```

then you can safely get the children of any widget (table, boxes, ...) and be sure you have the right Ada type. You won't need to explictly convert your widget to something else.

However, 'with'ing this unit means that your application will depend on every package of GtkAda, which is a little bit heavier, and explains why this is not the default. We do recommend you use it if it is not extremely important whether your application depends on all the packages of GtkAda.

The case of third party C widgets is a little bit trickier. Since GtkAda does not know anything about them when it is built, it can't magically convert the C widgets to Ada widgets. This is your job to teach GtkAda how to do the conversion.

We thus provide a 'hook' function which you need to modify. This function is defined in the package **Gtk.Type_Conversion**. This function takes a string with the name of the C widget (ex/ "GtkButton"), and should return a newly allocated pointer. If you don't know this type either, simply return **null**.

9.2 Using tagged types to extend Gtk widgets

Since version 0.6 of this toolkit, it is possible to associate your own data with existing widgets simply by creating new types. This section will show you a simple example, but you should rather read the source code in testgtk/ where we used this feature instead of using `user_data` as is used in the C version.

```
type My_Button_Record is new Gtk_Button_Record with record
    --  whatever data you want to associate with your button
end record;
type My_Button is access all My_Button_Record'Class;
```

With the above statements, your new type is defined. Every function available for `Gtk_Button` is also available for `My_Button`. Of course, as with every tagged type in Ada, you can create your own primitive functions with the following prototype:

```
procedure My_Primitive_Func (Myb : access My_Button_Record);
```

To instanciate an object of type `My_Button` in your application, do the following:

```
declare
   Myb : My_Button;
begin
   Myb := new My_Button_Record;
   Initialize (Myb);   --  from Gtk.Button
end;
```

The first line creates the Ada type, whereas the `Initialize` call actually creates the C widget and associates it with the Ada type.

9.3 Creating new widgets in Ada

With GtkAda, you can now create widgets directly in Ada. These new widgets can be used directly, as if they were part of gtk itself.

Creating new widgets is a way to create reuseable components. You can apply to them the same functions as would for any other widget, such as Show, Hide, ...

This section will explain how to create two types of widgets: composite widgets and widgets created from scratch. Two examples are provided with GtkAda, in the directories '`examples/composite_widget`' and '`examples/base_widget`'. Please also refer to the gtk+ tutorial, which describes the basic mechanisms that you need to know to create a widget (even if the Ada code is really different from the C code...)

9.3.1 Creating composite widgets

A composite widget is a widget that does not do much by itself. Rather, this is a collection of subwidgets grouped into a more general entity. For instance, among the standard widgets, `Gtk_File_Selection` and `Gtk_Font_Selection` belong to this category.

The good news is that there is nothing special to know. Just create a new tagged type, extending one of the standard widgets (or even another of your own widgets), provide a `Gtk_New` function that allocates memory for this widget, and call the `Initialize` function that does the actual creation of the widget and the subwidgets. There is only one thing to do: `Initialize` should call the parent class's `Initialize` function, to create the underlying C widget.

The example directory '`examples/composite_widget`' reimplements the `Gtk_Dialog` widget as written in C by the creators of gtk+.

9.3.2 Creating widgets from scratch

First, an important note: please do not read this if this is your first time using GtkAda or if you don't really understand the signal mechanism. Creating a nice and working widget really takes a lot of messing with the low level signals.

Creating a widget from scratch is what you want to do if your widget should be drawn in a special way, should create and emit new signals, ... The example we give in '`examples/base_widget`' is a small target on which the user can click, and that sends one of two signals "bullseye" or "missed", depending on where the user has clicked.

See also the example in '`examples/tutorial/gtkdial`' for a more complex widget, that implements a gauge where the user can move the arrow to select a new value.

Once again, the only two functions that you must create are `Gtk_New` and `Initialize`. This time, `Initialize` has to do two things:

```
Parent_Package.Initialize (Widget);

--  The above line calls the Initialize function from the parent.
--  This creates the underlying C widget, which we are going to
--  modify with the following call:

Gtk.Object.Initialize_Class_Record
  (Widget, Signals, Class_Record);
--  This initializes the "class record" for the widget and
--  creates the signals.
```

In the above example, the new part is the second call. It takes three or four arguments:

- `Widget` This is the widget that you want to initialize
- `Signals` This is an array of string access containing the name of the signals you want to create. For instance, you could create Signals with

  ```
  Signals : Gtkada.Types.Chars_Ptr_Array := "bullseye" + "missed";
  ```

  This will create two signals, named "bullseye" and "missed", whose callbacks' arguments can be specified with the fourth parameter.
- `Class_Record` Every widget in C is associated with two records. The first one, which exists only once per widget type, is the "class record". It contains the list of signals that are known by this widget type, the list of default callbacks for the signals, ...; the second record is an "instance record", which contains data specific to a particular instance.

In GtkAda, the "instance record" is simply your tagged type and its fields. The call to `Initialize_Class_Record` is provided to initialize the "class record". As we said, there should be only one such record per widget type. This parameter "Class_Record" will point to this records, once it is created, and will be reused for every instanciation of the widget.

- `Parameters` This fourth argument is in fact optional, and is used to specify which kind of parameters each new signal is expecting. By default (ie if you don't give any value for this parameter), all the signals won't expect any argument, except of course a possible user_data. However, you can decide for instance that the first signal ("bullseye") should in fact take a second argument (say a Gint), and that "missed" will take two parameters (two Gints).

  `Parameters` should thus contain a value of

  ```
  (1 => (1 => Gtk_Type_Int, 2 => Gtk_Type_None),
   2 => (1 => Gtk_Type_Int, 2 => Gtk_Type_Int));
  ```

  Due to the way arrays are handled in Ada, each component must have the same number of signals. However, if you specify a type of `Gtk_Type_None`, this will in fact be considered as no argument. Thus, the first signal above has only one parameter.

  Note also that to be able to emit a signal such a the second one, ie with multiple arguments, you will have to extend the packages defined in Gtk.Handlers. By default, the provided packages can only emit up to one argument (and only for a few specific types). Creating your own `Emit_By_Name` subprograms should not be hard if you look at what is done in '`gtk-marshallers.adb`'. Basically, something like:

  ```
  procedure Emit_With_Two_Ints
    (Object : access Widget_Type'Class;
     Name   : String;
     Arg1   : Gint;
     Arg2   : Gint);
  pragma Import (C, Emit_With_Two_Ints,
      "gtk_signal_emit_by_name");

  Emit_With_Two_Ints (Gtk.Get_Object (Your_Widget),
      "missed" & ASCII.NUL, 1, 2);
  ```

  will emit the "missed" signal with the two parameters 1 and 2.

Then of course `Initialize` should set up some signal handlers for the functions you want to redefine. Three signals are especially useful:

- "size_request"

  This callback is passed one parameter, as in :

  ```
  procedure Size_Request
     (Widget      : access My_Widget_Record;
      Requisition : in out Gtk.Widget.Gtk_Requisition);
  ```

  This function should modify Requisition to specify the widget's ideal size. This might not be the exact size that will be set, since some containers might decide to enlarge or to shrink it.

- "size_allocate"

  This callback is called every time the widget is moved in its parent window, or it is resized. It is passed one paramater, as in :

  ```
  procedure Size_Allocate
     (Widget     : access My_Widget_Record;
      Allocation : in out Gtk.Widget.Gtk_Allocation)
  ```

  This function should take the responsability to move the widget, using for instance `Gdk.Window.Move_Resize`.

- "expose_event"

  This callback is called every time the widget needs to be redrawn. It is passed one parameter, the area to be redrawn (to speed things up, you don't need to redraw the whole widget, just this area).

10 Support for Glade, the Gtk GUI builder

10.1 Introduction

GtkAda now comes with support for the GUI builder Glade (this is not the glade released with Gnat for distributed systems). Using Glade itself is straightforward: it is an intuitive point and click GUI builder. The main difference from other builders is that, since GtkAda builds a UI with blocks by default, you will not be asked to set the size and position of your windows by default. If you are looking for this kind of interaction you should consider using the Gtk_Fixed container. However, please read the GtkAda reference manual before considering using Gtk_Fixed.

10.2 Using Glade

Note that we only recommend using version 2.0.0 of Glade, as previous versions are not compatible. Using this version you can directly create Ada files from Glade by selecting Ada95 as the language under project options. Glade saves your interface in a project file whose syntax is XML based. In the following sections, we will refer to this file as either `the XML file` or `the project file` interchangeably.

10.2.1 Taking advantage of Glade's options

In this XML file, Glade will save various options that Gate can take advantage of. In particular, going to the project options window, some General Options in the `C Options` section will also be used by Gate: Gettext support, if enabled, will generate an additional package called <project>_intl that will use Gtkada.Intl; The `Set Widget Names` option is also recognized and will generate additional calls to Set_Name for each widget.

10.2.2 Associating icons with buttons

Up to now, the only way to specify an icon for a button was to associate a pixmap file to it. As an alternative, if the name specified in the `Icon` field does not contain any dots, Gate will consider the name as a variable rather than a file name. It is up to you to provide the necessary definitions in order to compile properly the generated code.

10.2.3 Note for GNU/Linux users

Some GNU/Linux systems come with a precompiled version of Glade that will usually have support for Gnome. Gate currently does not provide support for Gnome, which means that you should not enable Gnome support in Glade.

10.3 Gate

10.3.1 Invoking Gate

Gate is a program that is delivered with GtkAda. Gate takes the Glade XML file as an argument and generates a set of Ada files. When compiled, these files will recreate the interface you just designed with Glade.

Gate is generally invoked through the `Write source code` button in Glade's menu bar.

In addition, you can also invoke it from the command line by providing a Glade project file. This will generate Ada files in the directory set as the `Source Directory` project option, or otherwise in the current working directory.

10.3.2 Structure of the generated files

10.3.2.1 The main file

The main file is the name of the program name specified in the XML Glade file: `<program name>.adb`. It contains initialization and creation code for each top level widget contained in the XML file. This is intended as a convenient default to visualize your GUI, but you will usually want to modify this initialization.

10.3.2.2 Top level widget files

For each top level widget, Gate will generate a package `<widget>_Pkg` in the files `<widget>_pkg.ads` and `.adb`. These packages contain all the GtkAda calls needed to create the widgets you designed within Glade. You will usually not need to modify these files yourself.

10.3.2.3 Main handler file

This file, called `callbacks_<program name>.ads` contains all the Handler package instantiations needed by your application. You will usually not need to modify it.

10.3.2.4 Top level widget signal files

These are the most important files created by Gate. Each is called `<widget_name>_pkg-callbacks.adb`. They contain stubs for all the callbacks related to a top level widget you declared in Glade. With the main file, this the only files you should modify yourself. Note that it is recommended that you structure your application such that the real code is put in separate packages (i.e not generated files) as much as possible, to avoid potential merging problems when your interface is significantly changed within Glade. See next section for more details.

Currently, Gate will generate callback stub procedures that can handle the expected number of arguments for each signal. This is done using a relatively low level mechanism explained in the reference manual (package Gtk.Handlers). Basically, callbacks expecting arguments will take a Gtk_Argument as their only parameter, and Gate will generate the appropriate variable declaration and conversion calls to provide the expected GtkAda arguments. You should not modify this code by hand unless you know what you are doing.

For example, given a signal `delete_event`, Gate will generate the following procedure body, giving access to the arguments of this callback: a `Gtk_Window_Record` (Object) and a `Gdk_Event` (Arg1)

```
function On_Main_Window_Delete_Event
  (Object : access Gtk_Window_Record'Class;
   Params : Gtk.Arguments.Gtk_Args) return Boolean
is
   Arg1 : Gdk_Event := To_Event (Params, 1);
begin
   return False;
end On_Main_Window_Delete_Event;
```

10.3.3 Modifying generated files

Note that you can easily go back to Glade any time, modify your interface, and have Gate re-generate a set of files. All your modifications will be kept in the new files. For that, Gate creates a directory `.gate` in the current directory. Please do not delete it if you want Gate to be able to keep your changes from one version to the next.

Also note that to keep track of your modifications, gate relies on either `merge`, or `patch` and `diff` being available on your system. If you don't have a working set of `diff/patch`, `configure` will simply replace them by null operations, which means that regenerated files will override the previous ones.

Under Win32 based systems, Gate will not attempt to merge changes. Instead, it will always regenerate and overwrite every file.

In some cases, due to major changes in the project file, Gate may not be able to merge all the changes. In this case, it will notify you so, and will either keep the old and new sections in the files (if you are using `merge`), or files with a `.rej` extension will be generated to help you do the merge manually.

10.3.4 Adding Gate support for new widgets

This section is intended for developpers that wish to add support for new GtkAda widgets in Gate. Note that this section is not complete yet.

10.3.4.1 Implementing the `Generate` procedure

To provide support for a new widget, you first need to write a `Generate` procedure that will take care of generating the piece of code related to the widget's properties. Note that the parents properties do not have to be handled by this function.

- Glib.Glade API The Glib.Glade package contains a self documented API that will help you automate most of the work for common widgets.
- Most common mapping
  - Gen_New
  - Gen_Set
  - Gen_Call_Child

  - example
- Registering `Generate` functions Gtk.Glade.Register_Generate (Widget, Func) This procedure associates a given Generate procedure with a given widget C name.

10.4 Dynamic loading of XML files

10.4.1 Introduction to Libglade

Libglade is an external library provided by default on some systems (e.g GNU/Linux, Gnome) that supports dynamic loading of XML files and creation of widgets at run time. The advantage of using libglade is that you do not need to recompile your application to change your user interface. On the other hand, everything is done at run time, meaning less compile checks, and also the inability to take advantage of the object oriented features of GtkAda and Ada such as deriving from a widget creating using the GUI builder, to add fields and properties. See packages Glade and Glade.XML for more information.

10.5 Limitations

Gate currently support all Gtk+ widgets and properties available under Glade. But, to help you identify widgets that may not be supported (e.g Gnome widgets), Gate will generate a warning on the standard error:

```
$ gate warning.glade
Generating Ada files...

GtkAda-WARNING **: Unsupported widget GnomeCanvas (canvas1)
The following files have been created/updated in src:
[...]
```

and add a comment in the `<widget>_pkg.adb` file that looks like:

```
--  WARNING: Unsupported widget GnomeCanvas (canvas1)
```

This means that while generating the file Gate detected an unsupported widget (in this case GnomeCanvas) whose name is canvas1. If you get such a warning your file may or may not compile properly, but you won't get the complete widget hierarchy at run time.

Feel free to send us (see Chapter 13 [How to report bugs], page 34) the XML file that causes this problem. We don't guarantee a rapid fix for each particular problem but receiving real examples of missing functionnalities will certainly help implementing them faster.

11 Binding new widgets

GtkAda comes with a Perl script to help you create a binding to a C widget (this is the script we have used ourselves). This will not fully automate the process, although it should really speed things up. You will probably need less than 15 min to create a new binding once you will get used to the way GtkAda works. Note that your C file should have the same format as is used by Gtk+ itself.

Here are the steps to create a new binding :

- Give the 'generate.pl' script the C header file
- The perl script should ask you for every field in the C structure if you want to create a function to make it accessible from Ada. Answer *y* or *n* (the default value is *n*, just press ENTER).

  ```
  $ generate.pl ../include/gtk/gtkbutton.h > temporary
    Create a function for the field child (of type GtkWidget*) [n]?
    Create a function for the field in_button (of type guint) [n]?
    Create a function for the field button_down (of type guint) [n]?
  ```
- If you answered yes to at least one of the above questions, create a new C file, and move the C part of the *temporary* file created by generate.pl to this C file.
- Do a gnatchop on the *temporary* file created by generate.pl

  ```
  $ gnatchop temporary
  ```

  This should create two Ada files (spec and body)
- Edit these two files if needed (generally to change some parameters from Gint to Boolean for instance). It can be a little bit tricky when one of your functions returns a GList* or a GSList*. Please have a look at `gtk-combo.ads` for examples how to do this.

12 Debugging GtkAda applications

This chapter presents a number of technics that can be used when debugging GtkAda applications. First, the standard tools to debug Ada applications can be used:

- Compile with -g
  You should almost always include debugging information when compiling and linking your code. This gives you the possibility to use the debugger. See below the variable GDK_DEBUG for how to disable grabs.
- bind with -E
  Using this argument on the `gnatbind` or `gnatmake` command line will force the compiler to include backtraces when an exception is raised. These backtraces can be converted to symbolic backtraces by using the `addr2line` tool.
- Link with -lgmem
  Using this switch gives access to the `gnatmem` tool, that helps you to detect memory leaks or doubly-deallocated memory. The latter often results in hard-to-fix Storage_Error exceptions. See the GNAT User's guide for more information.

There are also a number of technics specific to GtkAda or gtk+ applications. For most of them, you might need to recompile these libraries with the appropriate switches to get access to the extended debugging features.

- Use the `--sync` switch
  All applications compiled with gtk+ automatically support this switch, which forces events to be processed synchronously, thus making it easier to detect problems as soon as they happen.
- break on g_log
  In the debugger, it is often useful to put a breakpoint on the glib function `g_log`. When gtk+ is linked dynamically, you will need to first start your application with `begin`, then put the breakpoint and continue the application with `cont`. This helps understand internal errors or warnings reported by gtk+ and glib
- compile glib with `--disable-mem-pools`
  Glib, the underlying layer that provides system-independent services to gtk+, has an extensive and optimized system for memory allocation. Bigger chunks of Memory are allocated initially, and then subdivided by glib itself. Although this is extremely performant, this also make the debugging of memory-related problems (storage_error) more difficult. Compiling with the above switch forces glib to use the standard malloc() and free() system calls. On GNU/Linux systems, it might be useful to set the variable `MALLOC_CHECK_` to 1 to use error-detecting algorithms (see the man page for malloc()).
- compile glib and gtk+ with `--enable-debug=yes`
  It is recommended that you always compile these two libraries by specifying this switch on the `configure` command line. In addition to giving access to the debugger, this also provides a number of environment variables that can be set to visualize events, object creation,...

For these three variables, the possible values are given below. These are lists of colon-separated keywords. You can choose to remove any of these value from the variable

– 'GOBJECT_DEBUG=objects:signals'
: This sets up the debugging output for glib. The value 'objects' is probably the most useful, and displays, on exit of the application, the list of unfreed objects. This helps detect memory leaks. The second value 'signals' will display all the signals emitted by the objects. Note that this results in a significant amount of output.

– 'GDK_DEBUG=updates:nograbs:events:dnd:misc:xim:colormap:gdkrgb:gc:pixmap:image:input:cursor'
: This sets up the debugging output for gdk. The most useful value is 'nograbs', which prevents the application from ever grabbing the mouse or keyboards. If you don't set this, it might happen that the debugger becomes unusable, since you don't have access to the mouse when the debugger stops on a breakpoint. Another simpler solution is to debug remotely from another machine, in which case the grabs won't affect the terminal on which the debugger is running.

– 'GTK_DEBUG=misc:plugsocket:text:tree:updates:keybindings'
: This sets up the debugging output for gtk. Almost all of these values are mostly for internal use by gtk+ developpers, although 'keybindings' might prove useful sometimes.

• Import the C function ada_gtk_debug_get_ref_count

This function has the following Ada profile:

```
function Ref_Count (Add : System.Address) return Guint;
pragma Import (C, Ref_Count, "ada_gtk_debug_get_ref_count");
```

and should be called in a manner similar to

```
declare
   Widget : Gtk_Widget;
   Count  : Guint;
begin
   Count := Ref_Count (Get_Object (Widget));
end;
```

and returns the internal reference counter for the widget. When this counter reaches 0, the memory allocated for the widget is automatically freed.

This is mostly a debugging aid for people writting their own containers, and shouldn't generally be needed. You shouldn't rely on the internal reference counter in your actual code, which is why it isn't exported by default in GtkAda.

13 How to report bugs

GtkAda is becoming more and more stable due to its increasing use, but you may still find bugs while using it. We have tried to test it as much as possible, essentially by converting the testgtk.c file found in the gtk distribution, as well as with generating a significant enumber of interfaces using the GUI builder and Gate. We strongly suggest that you have a look at testgtk, which gives a lot of examples of how to use this toolkit.

There are two kinds of problems you can encounter:

- If the gtk library itself was compiled with warnings turned on, you may get some warnings messages, mainly because of types problems. These warnings should not appear, as we have tried to be as type safe as possible in this package. To know exactly where the problem is, compile your program with debug information, run gdb, and set a breakpoint on the function `g_log`. Then run your program as usual, using the `run` command. Then send us the result of the `where` command. Here is a summary:

```
$ gnatmake -f -g <your_program_name> ‘gtkada-config‘
$ gdb <your_program_name>
(gdb) break main
(gdb) run
(gdb) break g_log
(gdb) continue
....
(gdb) where
```

- In some (hopefully) rare cases, you can even get a segmentation fault within gtk. That means there is definitly something wrong either in your program or in the toolkit. Please check your program carefully and, if you think this is a problem in GtkAda itself, send us an e-mail.

If you are a supported user of GNAT, send mail to `mailto:report@gnat.com` to report errors, otherwise send mail to the authors (`mailto:gtkada@ada.eu.org`) explaining exactly what your are doing, what is the expected result and what you actually get. Please include the required sources to reproduce the problem, in a format usable by `gnatchop` (basically, insert all the required sources at the end of the mail). Please try to provide as small as possible a subset of your sources.

Of course, we will welcome any patch you can provide, so that this toolkit may be as useful as possible.

14 Bibliography

We recommand the following documents. Most of them were written with C in mind, but should be easily adapted after you've read the rest of this document.

- [1] "Gtk+/Gome Application Development" – Havoc Pennington This book, by one of the main authors of the the GNOME environment, describes in detail some of the inner mechanisms of gtk+, including signal handling, and a complete description of all the widgets and all the events found in `Gdk.Event`.

  It is worth noting that this book has been published under the Open Publication License. You can get an electronic copy of it at `http://www.opencontent.org/`.

Table of Contents