Using GtkAda in Practice

Ahlan Marriott, Urs Maurer

White Elephant GmbH, Beckengässchen 1, 8200 Schaffhausen, Switzerland; email: software@white-elephant.ch

Abstract

This article is an extract from the industrial presentation "Astronomical Ada" which was given at the 2017 Ada-Europe conference in Vienna.

The presentation was an experience report on the problems we encountered getting a program written entirely in Ada to work on three popular operating systems: Microsoft Windows (XP and later), Linux (Ubuntu Tahr) and OSX (Sierra).

The main problem we had concerned the implementation of the Graphical User Interface (GUI). This article describes our work using GtkAda.

Keywords: Gtk, GtkAda, GUI

1 Introduction

The industrial presentation was called "Astronomical Ada" because the program in question controls astronomical telescopes.

1.1 Telescopes

The simplest of telescopes have no motor. An object is viewed simply by pointing the telescope at it. However, due to the rotation of the earth, the viewed object, unless the telescope is continually adjusted, will gradually drift out of view.

To compensate for this, a fixed speed motor can be attached such that when aligned with the Earth's axis it effectively cancels out the Earth's rotation.

However many interesting objects appear to move relative to the Earth, for example satellites, comets and the planets. To track this type of object the telescope needs to have two motors and a system to control them.

Using two motors the control system can position the telescope to view anywhere in the night sky.

Our Ada program (*SkyTrack*) is one such program. It can drive the motors to position the telescope onto any given object from within its extensive database and thereafter follow the object either by calculating its path or, in the case of satellites and comets, follow the object according to a downloaded pre-calculated path.

1.2 Graphical User Interface

The GUI is used to instruct the program where to position the telescope and what astronomical object it should follow.

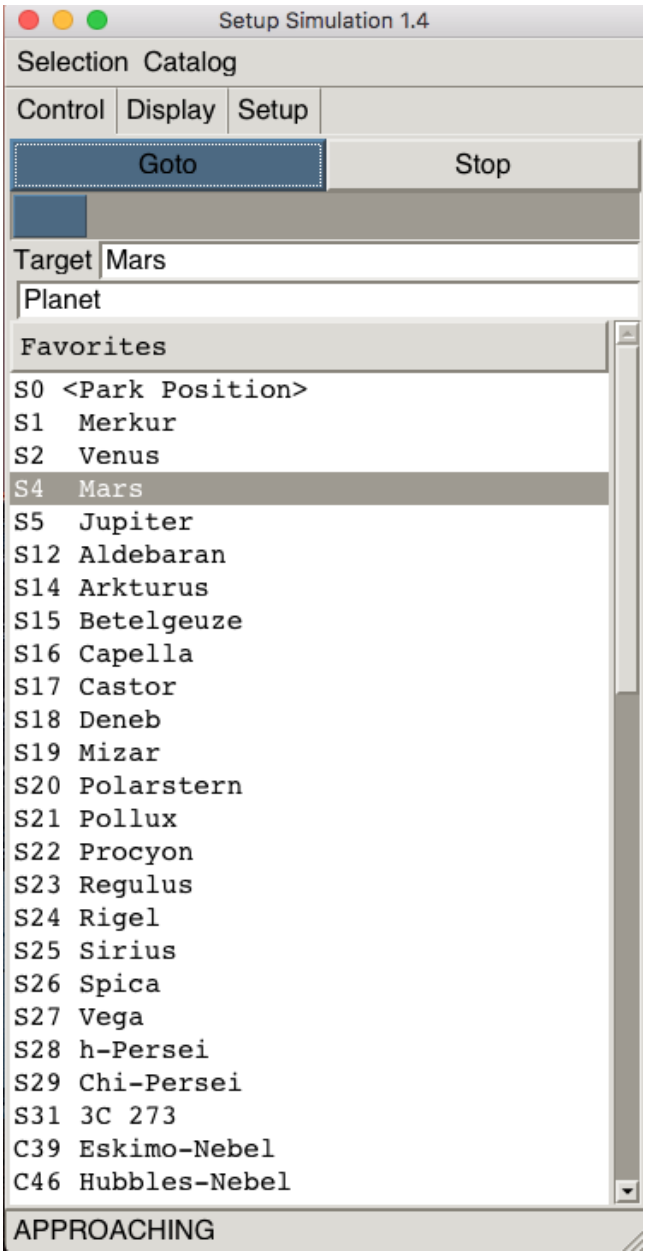

Figure 1 - SkyTrack GUI

The screen shot shown as figure 1 shows the *SkyTrack* program positioning the telescope on Mars. An object selected from the *Favourites* catalogue.

The GUI was implemented using a package that provides a simple interface to create and manipulate common graphical objects. It was originally implemented using direct calls to the Windows API so, at least in theory; all we had to do was re-implement the implementation.

We chose to re-implement the GUI based on Gtk because both Gtk and Ada bindings to Gtk were available on all the designated target platforms.

GtkAda are Ada bindings to Gtk that are available from AdaCore at their web site libre.adacore.com/download. Unfortunately, by themselves, these are not sufficient to implement a GUI of any complexity. A lot of extra code has to be written in order that Gtk can actually be used.

This article describes the code we developed in order to use GtkAda.

2 Restrictions

The Windows API is not task safe. By which we mean that although the Windows *SendMessage* and *PostMessage* procedures are thread-safe, the API generally requires the passing of pointers to external objects. This is unsafe because the referenced object must be kept until the message is processed. Also the object must be locked against concurrent access because Windows supports message loops in different threads and the sending/posting of messages across thread borders.

Therefore in our original Windows based implementation we used protected objects to prevent concurrent API calls and an Ada task to process the Windows message loop.

However Gtk has the even more dramatic restriction that all Gtk calls **must** be executed from the **same** thread.

This required us to develop a system that provided our GUI with a simple and reliable means to make Gtk calls whilst at the same time guaranteeing that they were executed by the same thread.

In our implementation a dedicated thread is provided to process all the Gtk calls. The GUI package makes calls to this thread to request that it execute Gtk calls on its behalf.

In this arrangement, the GUI can be considered to be the client and the dedicated thread, the Gtk server.

We identified two types of Gtk request that the client may make: Synchronous and Asynchronous.

A synchronous request is a request made by the Gtk client to the Gtk server that expects the server to return a value. For example retrieving the contents of an edit box.

An asynchronous request is a request made by the Gtk client to the Gtk server that does **not** return a value. For example writing a row to a list view.

3 Synchronous requests

The synchronous interface consists of an abstract type and an abstract procedure based on this type.

```
type Request_Data is abstract tagged null record;
procedure Synchronous_Service (Data : in out
   Request_Data) is abstract;
```

The Gtk client makes a synchronous request to the Gtk server by extending the abstract type to include data that is to be sent to the server as well as the data that the client expects to receive from the server.

The following is an example demonstrating how to determine whether or not a specified check box is checked.

First the abstract type *Request_Data* is extended to make a new type *Check_Enquiry_Data*. This is defined to be a record containing two fields: *Check_Box* to specify the check box to be enquired and *Is_Checked* to hold the result of the enquiry.

```
type Check_Enquiry_Data is new Request_Data with
record
   Check_Box  : Gtk.Check_Button.Gtk_Check_Button;
   Is_Checked : Boolean;
end record;
```

The abstract procedure *Synchronous_Service* then has to be defined for the extended type. This procedure contains the code to be executed by the server on behalf of the client.

```
overriding procedure Synchronous_Service
  (Data : in out Check_Enquiry_Data) is
begin
   Data.Is_Checked := Data.Check_Box.Get_Active;
end Synchronous_Service;
```

The synchronisation and passing of data between the client and the server is implemented using a protected type that has two entries, one for the client to call and another that is used to block the client from immediately returning. The protected type also has a state and a means of retaining access to the client data.

```
type Request_Data_Ptr is access all
   Request_Data'class;
protected Gateway is
   entry Synchronous_Request (Data : in out
      Request_Data'class);
private
   entry Serviced (Unused_Data : in out
      Request_Data'class);
   State : Gateway_State := Idle;
   Data  : Request_Data_Ptr;
end Gateway;
```

In order that the client defined synchronous procedure is executed in the context of the server thread, the client needs to create a variable of the extended type, initialise it with the data required by the synchronous procedure and then rendezvous with the server.

At the rendezvous with the server the data will be passed to the server and the client blocked until the server has executed the synchronous procedure associated with the data.

In the following example the client function *Is_Checked* takes a check box as its only parameter. It puts this into a variable of type *Check_Enquiry_Data* that is an extension of *Request_Data* (see previously). The data is then passed to the Gtk server by making a rendezvous at the entry *Synchronous_Request*. When it is released from the entry it

obtains the result from the variable and returns it to the caller.

```
function Is_Checked (The_Check_Box : Check_Box)
return Boolean is
   Data : Check_Enquiry_Data := (Request_Data with
      Check_Box  => The_Check_Box.The_Box,
      Is_Checked => False);
begin
   Gateway.Synchronous_Request (Data);
   return Data.Is_Checked;
end Is_Checked;
```

The entry *Synchronous_Request* within the protected type *Gateway* is implemented as follows:

```
entry Synchronous_Request (Data : in out
   Request_Data'class)
when State = Idle is
begin
   Gateway.Data := Data'unchecked_access;
   State := Busy;
   requeue Serviced;
end Synchronous_Request;
entry Serviced (Unused_Data : in out
   Request_Data'class)
when State = Ready is
begin
   State := Idle;
end Serviced;
```

Callers to *Synchronous_Request* are blocked until the server is ready to process the request by placing a guard on the entry, which is opened when the gateway state is set to *Idle*.

Within the entry a pointer is made to the data passed as the entry's parameter and the state set to *Busy*.

Finally it makes a call to the entry *Serviced* that effectively blocks the call from returning until the state is set to *Ready*.

In this way the client waits for the server to be *Idle*, sets up a pointer to the data, indicates that the data is ready and then waits for the server to indicate that it has processed the data.

Note that the requeue prevents the entry's parameter from being destroyed. Therefore until the state is set to *Ready*, the pointer *Gateway.Data* remains valid.

3.1 Synchronous Server

Making Gtk calls do not, by themselves, result in anything happening. For something to happen a thread must execute *Gtk.Main.Main_Iteration* in a loop.

```
loop
   Unused_Boolean := Gtk.Main.Main_Iteration;
end loop;
```

Consequently a minimum Gtk server must do this as well as process the synchronous requests made by the Gtk clients.

We can do this by modifying the Gtk *Main_Iteration* loop so that *Main_Iteration* is only called whilst there are Gtk events that need to be processed and then making a selective wait with timeout to check if there are any synchronous requests pending.

The code to determine whether there are any pending requests, to obtain the request and to signal that the request has been processed, is implemented by two entries and one function as part of the *Gateway* protected type.

```
protected Gateway is
   entry Check;
   entry Complete_Synchronous_Service;
end Gateway;
function Synchronous_Data return Request_Data_Ptr;
```

The entry *Check* blocks until the state is set to *Busy*. This happens after the client has entered *Synchronous_Request* and has made a pointer to the request data.

```
entry Check
when (State = Busy) is
begin
  null;
end Check;
```

The function *Synchronous_Data* can be used to access the request data.

```
function Synchronous_Data return Request_Data_Ptr
is
begin
   return Gateway.Data;
end Synchronous_Data;
```

The entry *Complete_Synchronous_Service* sets the state to *Ready* which frees the client blocked on the requeue at the entry *Serviced*.

```
procedure Complete_Synchronous_Service is
begin
   State := Ready;
end Complete_Synchronous_Service;
```

A Gtk server for synchronous requests can therefore be implemented as follows:

```
loop
   while Gtk.Main.Events_Pending loop
      Unused_Boolean := Gtk.Main.Main_Iteration;
   end loop;
   select
      Gateway.Check;
      Synchronous_Service
                  (Gateway.Synchronous_Data.all);
      Gateway.Complete_Synchronous_Service;
   or
      delay The_Period;
   end select;
end loop;
```

The server processes any pending Gtk events then checks for any client requests. If there aren't any within a short

period of time (we typically wait for 50ms) the process is repeated.

If *Gateway.Check* is taken then the function *Synchronous_Data* is called to obtain the request data and then the client defined synchronous procedure associated with the data type is called. After which the procedure *Complete_Synchronous_Service* is called to release the client.

4 Asynchronous calls

We could have left it at that. We could have implemented our entire application by processing all our Gtk calls as synchronous requests. However if we had done so, the performance would have been very poor.

This is because the task switch between the client and the server and then back again are both relatively expensive. Using the synchronous mechanism to place a large amount of data into a list view is noticeably and unacceptably slow.

For the sake of efficiency we needed to implement an asynchronous method whereby the client can issue requests to the server to be processed at some future time. The client does not wait for the server to process these requests and therefore does not have to incur the penalty of the task switch back and forth to the server.

The asynchronous method is very similar to the synchronous method described previously, in so far that it relies on an abstract type that is extended to contain the request data and a procedure that overrides the type's abstract procedure.

It differs from the synchronous method in that instead of synchronising with the server it simply places the request into a protected queue ready for the server to process.

Unlike the synchronous method the client is not blocked and so is immediately free to make further requests. In the example of filling a list view with data, the client can first place all the requests into the queue and then, when this is done, the server can process the whole of the queue.

Having a queue of work to process avoids having to continually switch between client and server and is noticeably faster.

The asynchronous interface consists of an abstract type and an abstract procedure based on this type.

```
type Message_Data is abstract tagged null record;
procedure Asynchronous_Service
   (Message : Message_Data) is abstract;
```

The Gtk client makes an asynchronous request to the Gtk server by extending the abstract type to include data that is to be sent to the server.

The following is an example demonstrating how to set a specified check box.

First the abstract type *Message_Data* is extended to make a new type *Set_Check_Data*. This is defined to be a record containing the field *Check_Box* which is used to specify the check box to be set.

```
type Set_Check_Data is new Message_Data with
record
   Check_Box  : Gtk.Check_Button.Gtk_Check_Button;
end record;
```

The abstract procedure *Asynchronous_Service* then has to be defined for the extended type. This procedure contains the code to be executed by the server on behalf of the client.

```
overriding procedure Asynchronous_Service
  (Data : in out Set_Check_Data) is
begin
   Data.Check_Box.Set_Active (True);
end Asynchronous_Service;
```

The data is placed into the queue for the server to process using a protected type that has an entry and an indefinite doubly linked list that is used to implement the queue of requests.

```
package Message_List is new
   Ada.Containers.Indefinite_Doubly_Linked_Lists
   (Message_Data'class);
protected Gateway is
   procedure Asynchronous_Request (Data : in
      Message_Data'class);
private
   The_Messages : Message_List.Item;
end Gateway;
```

In order that the client defined asynchronous procedure is executed in the context of the server task, the client needs to create a variable of the extended type, initialise it with the data required by the asynchronous procedure and then place the data into the server queue.

In the following example, the client procedure *Set* takes as its only parameter the check box that should be set.

It copies the parameter into a variable of type *Set_Check_Data* that is an extension of *Message_Data* (see previously). The data is then placed into the Gtk server queue by making a call to the entry *Asynchronous_Request*.

```
procedure Set (The_Check_Box :
                  Gtk.Check_Button.Gtk_Check_Button)
is
  Data : Set_Check_Data
        := (Message_Data with
            Check_Box => The_Check_Box);
begin
   Gateway.Asynchronous_Request (Data);
end Is_Checked;
```

The entry *Asynchronous_Request* within the protected type *Gateway* is implemented as follows:

```
protected body Gateway is
   procedure Asynchronous_Request (Data : in
      Message_Data'class) is
   begin
      The_Messages.Append (Data);
   end Asynchronous_Request;
end Gateway;
```

4.1 Asynchronous Server

In order to process asynchronous requests, in addition to synchronous requests, the server needs to be extended. The *Check* entry needs to block until either a synchronous request is made or the queue of asynchronous work becomes not empty and for it to return what type of request is available.

```
type Data_Type is (Synchronous, Asynchronous);
entry Check (The_Data_Type : out Data_Type)
when not (State = Busy) or else
   (The_Messages.Count > 0) is
begin
   If The_Messages.Count > 0 then
      The_Data_Type := Asynchronous;
   elsif State = Busy then
      The_Data_Type := Synchronous;
   end if;
end Check;
```

The main processing loop can then use the request type to decide how to process the request.

```
loop
   while Gtk.Main.Events_Pending loop
      Unused_Boolean := Gtk.Main.Main_Iteration;
   end loop;
   select
      Gateway.Check (The_Data_Type);
      case The_Data_Type is
      when Synchronous =>
         Synchronous_Service
            (Gateway.Synchronous_Data.all);
         Gateway.Complete_Synchronous_Service;
      when Asynchronous =>
         Asynchronous_Service
            (Gateway.Next_Message);
         Gateway.Delete_First_Message;
      end case;
   or
      delay The_Period;
   end select;
end loop;
```

The function *Next_Message* is a function to return the asynchronous request at the head of the asynchronous request queue and the procedure *Delete_First_Message* removes it from the queue.

```
function Next_Message return Message_Data'class is
   The_Message : constant Message_Data'class :=
      The_Messages.First_Element;
begin
   return The_Message;
end Next_Message;

procedure Delete_First_Message is
begin
   The_Messages.Delete_First;
end Delete_First_Message;
```

5 Callbacks

Most GUI implementations will require some form of callback in order that they can be notified of user interaction. Some callbacks need only identify the object (for example the button when a button is clicked) whilst others will require additional information (for example which row within a list view has been clicked).

GtkAda provides bindings to the Gtk mechanism however it is important to realise that only a very limited amount of work should be performed within these callbacks otherwise the responsiveness of the windowing system will be adversely affected.

To prevent this type of degradation, the Gtk callbacks in our GUI implementation are kept as simple as possible – any large amount of work is placed into a protected queue to be processed by a separate dedicated task.

For example, the Gtk callback called as a result of clicking on a button would add an action to the callback handler queue. The callback handler task processing this queue eventually processes the action; which invariably results in a routine being called that does whatever work is actually required.

By delegating this work to another task, the Gtk server task is released to service Gtk requests (perhaps generated as a result of the button being clicked) as well as processing the main Gtk event loop, thereby keeping windows and mouse tracking up to date.

Although the provision of this mechanism is not a requirement for a functional Gtk server, we found it convenient if the mechanism is provided in the same package as the server.

For example, causing *The_Action_Routine* to be executed whenever *The_Button* is clicked could be coded as follows.

```
type Action_Routine is access procedure;
package Action_Callback is new
   Gtk.Handlers.User_Callback (
      Widget_Type => Gtk.Widget.Gtk_Widget_Record,
      User_Type => Action_Routine);
procedure Action_Handler (
   Unused : access
      Gtk.Widget.Gtk_Widget_Record'class;
   The_Action_Routine : Action_Routine) is
begin
   Callback_Handling.Put (The_Action_Routine);
end Action_Handler;
Gtk.Button.Gtk_New (The_Button, “Button”);
Action_Callback.Connect (
   The_Button,
   "clicked",
   Action_Callback.To_Marshaller(
      Action_Handler'access),
   The_Action_Routine);
```

When *The_Button* is clicked, Gtk calls the procedure *Action_Handler* in the context of the Gtk server thread. All

this does is place *The_Action_Routine* into the callback queue.

A dedicated task *Callback_Handler* created by the Gtk server serially executes procedures placed in this queue. This can be coded as follows:

```
package Callback_List is new
   Definite_Doubly_Linked_Lists (Action_Routine);

protected Callback_Handling is
   procedure Put (The_Action : Action_Routine);
   procedure Finish;
   entry Get (The_Callback : out Action_Routine);
private
   Is_Enabled : Boolean := True;
   The_Callback_List : Callback_List.Item;
end Callback_Handling;

protected body Callback_Handling is
   procedure Put (The_Action : Action_Routine) is
   begin
      if Is_Enabled then
         The_Callback_List.Append (The_Callback);
      end if;
   end Put;
   procedure Finish is
   begin
      Is_Enabled := False;
   end Finish;
   entry Get (The_Routine : out Action_Routine) when
      (not Is_Enabled) or (The_Callback_List.Count > 0)
   is
   begin
      if Is_Enabled then
         The_Routine :=
            The_Callback_List.First_Element;
         The_Callback_List.Delete_First;
      else
         The_ Routine := null;
      end if;
   end Get;
end Callback_Handling;
task body Callback_Handler is
   The_Routine : Action_Routine;
begin
   loop
      Callback_Handling.Get (The_ Routine);
      exit when The_ Routine = null;
      The_Routine.all;
   end loop;
   The_Termination_Handler.Finalize;
end Callback_Handler;
```

5.1 Qualified callbacks

A qualified callback is a variation on the callback idea. It works in a similar fashion as the simple callback described previously but in addition returns client supplied data. This type of callback is used when it is insufficient just knowing which widget has been the subject of an event. For example when the row of a list view has been clicked the application invariably wants to know which row was clicked.

To support qualified callbacks we need to base the callback queue on a record that may contain different information depending on the type of the callback.

```
type Callback is (Simple, Qualified);
type Quaified_Routine is
   access procedure (Item : Information);
type Callback_Data (The_Callback : Callback := Action)
is record
   case The_Callback is
   when Simple =>
      Simple_Action : Action_Routine;
   when Qualified =>
      Qualified_Action : Qualified_Routine;
      The_Information : Information;
   end case;
end record;
package Callback_List is new
Definite_Doubly_Linked_Lists (Callback_Data);
```

and the protected type *Callback_Handling* extended accordingly.

```
protected Callback_Handling is
   procedure Put (The_Action : Action_Routine);
   procedure Put (The_Action : Qualified_Routine;
                  The_Information : Information);
   procedure Finish;
   entry Get (The_Callback : out Callback_Data);
end Callback_Handling;
task body Callback_Handler is
   The_Callback : Callback_Data;
begin
    loop
      Callback_Handling.Get (The_Callback);
      case The_Callback.The_Callback is
      when Simple =>
         exit when The_Callback.Simple_Action = null;
         The_Callback.Simple_Action.all;
      when Qualified =>
         The_Callback.Qualified_Action.all
            (The_Callback.The_Information);
      end case;
   end loop;
   The_Termination_Handler.Finalize;
end Callback_Handler;
```

The following is an example of how this extended callback mechanism could be used to indicate which row of a list view has been clicked.

```
package Qualified_Callback is new
Gtk.Handlers.User_Callback
(Gtk.Widget.Gtk_Widget_Record, Qualified_Routine);
Gtk.Tree_View.Gtk_New (The_View);
Qualified_Callback.Connect (
   The_View,
   "row-activated",
   Qualified_Callback.To_Marshaller
```

```
      (List_Click_Handler'access),
      The_Routine);
```

The procedure *List_Click_Handler* and the client supplied qualified routine are connected to the row activation event of the list view.

When the row of the list view is clicked, the procedure *List_Click_Handler* is called with both the list view widget and the user supplied qualified routine passed as parameters.

The *List_Click_Handler* procedure can then retrieve the information associated with the row that has been activated and then schedule a callback with this information.

```
procedure List_Click_Handler (
   Widget : access
      Gtk.Widget.Gtk_Widget_Record'class;
      The_Routine : Qualified_Routine)
is
  Iter  : Gtk.Tree_Model.Gtk_Tree_Iter;
  Model : Gtk.Tree_Model.Gtk_Tree_Model;
  Value : Glib.Values.GValue;
begin
   Gtk.Tree_Selection.Get_Selected
      (Gtk.Tree_View.Get_Selection
         (Gtk.Tree_View.Gtk_Tree_View(Widget)),
       Model, Iter);
   Gtk.Tree_Model.Get_Value (Model, Iter, 0, Value);
   Callback_Handling.Put (The_Routine,
      Information(Glib.Values.Get_Ulong(Value)));
end List_Click_Handler;
```

6 Closing the server

As previously described, the Gtk server sits in a loop either executing Gtk events or processing synchronous or asynchronous requests. However this is insufficient, we need a method to exit the loop and return control back to the thread that called the server procedure.

This is achieved by calling the *Gateway* entry *Quit*. The *Gateway* is the protected type used by clients to make synchronous or asynchronous requests. This needs to be enhanced so that a new type of request can be made.

This new type is the *Killed* request.

```
type Data_Type is (Killed, Synchronous, Asynchronous);
```

The gateway procedure *Quit* sets the gateway state to *Killed* and the *Check* entry is enhanced to return the *Killed* request if called in the *Killed* state.

```
procedure Quit is
begin
   State := Killed;
end Quit;
entry Check (The_Data_Type : out Data_Type)
when (State = Busy) or else
   (State = Killed) or else
   (The_Messages.Count > 0) is
begin
   if State = Killed then
      The_Data_Type := Killed;
   elsif The_Messages.Count > 0 then
      The_Data_Type := Asynchronous;
   elsif State = Busy then
      The_Data_Type := Synchronous;
   end if;
end Check;
```

The server can then use this request type as an indication that it should exit the otherwise infinite processing loop.

```
loop
   while Gtk.Main.Events_Pending loop
      Unused_Boolean := Gtk.Main.Main_Iteration;
   end loop;
   select
      Gateway.Check (The_Data_Type);
      case The_Data_Type is
      when Killed =>
         Gtk.Widget.Destroy
            (Gtk.Widget.Gtk_Widget(The_Main_Window));
         exit;
      when Synchronous =>
      …
      when Asynchronous =>
      …
      end case;
   end select;
end loop;
```

Note that this is also where the server destroys the main window that it created just before calling the start-up procedure.

7 The OSX restriction

Under MS-Windows and Linux, the server can be a standard Ada task; usually created in the package body. Unfortunately under OSX the thread that executes the Gtk calls **must** be the main thread of the process.

This tedious restriction means that the server has to be implemented as a procedure (which we call *Execute* – see below for details) that is called from the main thread and returns only when the GUI has been closed down.

To help synchronise start-up and shutdown we implemented our server procedure to include two procedures passed as parameters, one that is called immediately after the server is able to accept requests and the other that is called in response to the GUI being closed down.

Both of these procedures are executed in their own dedicated tasks so that they can better interact with the Gtk server.

Typically the start-up procedure is used to start GUI related tasks and to create the GUI objects, whilst the shutdown procedure is used to terminate these tasks.

```
procedure Execute (
   Startup_Routine  : access procedure
      (Window : Gtk.Window.Gtk_Window);
   Termination_Routine : access procedure);
```

7.1 Start-up

The server procedure creates the main Gtk window. It then creates the start-up task with the user supplied start-up procedure passed as its parameter. It then waits for the task to start by making a rendezvous with it, at which time it passes the previously created main Gtk window. After the rendezvous it executes the server code described previously.

```
task type Startup (Startup_Routine : access procedure
   (Window : Gtk.Window.Gtk_Window))
is
   entry Start (Window : Gtk.Window.Gtk_Window) ;
end Startup;
type Startup_Ptr is access Startup;
Startup_Task : Startup_Ptr;

task body Startup is
  The_Main_Window : Gtk.Window.Gtk_Window;
begin
   accept Start (Window : Gtk.Window.Gtk_Window) do
      The_Main_Window := Window;
   end Start;
   Startup_Routine.all (The_Main_Window);
end Startup;
procedure Execute (
   Startup_Routine  : access procedure
      (Window : Gtk.Window.Gtk_Window);
   Termination_Routine : access procedure)
is
   The_Main_Window : Gtk.Window.Gtk_Window;
begin
   Gtk.Window.Gtk_New (The_Main_Window);
   Startup_Task := new Startup (Startup_Routine);
   Startup_Task.Start (The_Main_Window);
   while Gtk.Main.Events_Pending loop
      Unused_Boolean := Gtk.Main.Main_Iteration;
   end loop;
   …  -- remainder of server
```

7.2 Termination

Early in the execution of the server procedure, a termination task is created.

```
task type Termination_Handler is
   entry Start;
   entry Finalize;
end Termination_Handler;
```

The server then connects a function to the delete-event of the main window.

```
package Window_Callback is new
   Gtk.Handlers.Return_Callback
   (Gtk.Window.Gtk_Window_Record, Boolean);
   Window_Callback.Connect (The_Main_Window,
                            "delete-event",
                            Close_Window'access);
```

The connected function is called when the main window of the GUI is closed.

```
function Close_Window ( Unused : access
   Gtk.Window.Gtk_Window_Record'class)
return Boolean is
begin
   The_Termination_Handler.Start;
   return True; -- Don't destroy the main window.
end Close_Window;
```

This function starts the server termination task by making a rendezvous at its *Start* entry. Note that the function returns *True* to indicate to Gtk that the window should **not** be destroyed. This is so that the termination routine can still access the window in order that it can retrieve information before the window is actually closed and the information lost. For example its size and position on screen.

```
task body Termination_Handler is
begin
   accept Start;
   The_Termination_Routine.all;
   Callback_Handling.Finish;
   accept Finalize;
   Gateway.Quit;
end Termination_Handler;
```

The termination task waits to be started then executes the termination routine supplied by the client. When this is finished it causes the callback task to terminate. It waits for the callback task to rendezvous at its *Finalize* entry to make sure that all the action routines have been executed before it finally closes the window and terminates the Gtk server by calling the Gateway *Quit* entry.

8 Downloads

A working example of a Gtk server package as described in this article may be downloaded from our web site www.white-elephant.ch

We cordially invite readers to comment and suggest improvements and/or corrections. We do not consider ourselves to be in any way knowledgeable with regard to Gtk or GtkAda and so would very much appreciate feedback.

Acknowledgements

We would like to acknowledge the work done by Dmitry A. Kazakov and Maxim Reznik in their GtkAda Contributions.
(http://www.dmitry-kazakov.de/ada/gtkada_contributions)

The work presented in this article was inspired by and derived from ideas implemented by Dmitry and Maxim in the afore-mentioned work.

However our implementation differs from theirs in that our Gtk server blocks (with timeout) waiting for requests rather than periodically processing pending requests using a timer and that our implementation of asynchronous requests uses a queue.

In our experience using a queue and a select with timeout greatly increases the performance of the GUI.