Programming GTK in Free Pascal: GTK Revisited

Michael Van Canneyt

January 2003

1 Introduction

In a previous series of articles, the possibility of programming in GTK using Free Pascal was examined. The units that were hitherto available provide access to the bare GTK library API. It is therefore possible to create (Free) Pascal programs that used the GTK library, this has been amply demonstrated in the previous series of articles. However, it was also obvious that the raw GTK interface has some drawbacks when programming it in Pascal:

1. C is not an object-oriented language. This means that it has no language support for object inheritance. This makes it necesary to do a lot of typecasts as in the following example:

   ```
   var
     window : PGtkWindow;
     button : PGtkWidget;

   begin
     ...
     gtk_container_add(PGTKContainer(window),button);
   ```

   The typecast to the `PGTKContainer` type is needed to be able to use the `gtk_container_add` functionality of the `window` variable. This functionality is available because the `TGtkWindow` type is implemented as a descendent of the `TGTKContainer` type. However the C compiler cannot check this, so a typecast is needed. Needless to say, this does not make for readable code.

2. C – and hence the GTK interface – uses null-terminated strings. This is not really a problem, but given that most pascal programmers use (ansi)strings it forces the use of a typecast each time a string must be passed to a GTK function:

   ```
   Var
     S : String;
     L : PGtkLabel;

   begin
     S:='Today is '+DateToStr(Date);
     gtk_label_set_text(L,PChar(S));
   ```

   Again, not very pleasing to a pascal programmer.

3. Callbacks must be procedures. This means that object methods cannot be passed as callbacks to GTK signal handlers. For example the following will not work:

```
Type
  TMainForm = Class
    Procedure ButtonClicked(P : PGTKWidget; Data : Pointer);cdecl;
  end;

...

  gtk_signal_connect (PGTKOBJECT (mybutton), 'clicked',
                      GTK_SIGNAL_FUNC (@ButtonClicked), NULL);
```

The call to `gtk_signal_connect` will result in a compiler error. It can be avoided using some tricks, but when the code would be run, it would result in access violations anyway. Note that the typecast using `GTK_SIGNAL_FUNC` is needed even when no objects are used.

That makes the use of objects combined with GTK not very easy. Furthermore, the calling convention for these procedures must be `cdecl`; Again, this is not a real problem, but it is not 'natural' for a pascal programmer, making it easy to forget. The obligatory typecast to `GTK_SIGNAL_FUNC` masks this from the compiler, and run-time errors will occur.

To make GTK more accessible for Free Pascal or Delphi programmers, **FPgtk** was developed by Luk Vandelaer, and is now part of the FPC distribution. It is a very thin object oriented layer around the GTK widgets. It introduces the following things:

1. A class hierarchy that matches the hierarchy in GTK. This removes the need for typecasts.
2. Use of strings for all arguments that require strings. Conversion to and from null-terminated strings is handled transparantly.
3. Support for methods as callbacks for signals. No typecasting is needed, and a procedural type exists for each kind of callback.
4. Most GTK functions or procedures are mapped to a method of a class.
5. Use of properties. All properties of a GTK object which could be set/read using a `gtk_XXX_get_YYY` kind of calls are mapped to properties which can be read or set, as needed.
6. Transparant creation and destruction of the underlying GTK objects. When a GTK object is destroyed (by some callback) the wrapper class is also destroyed.

All GTK (version 1.2) widgets have been wrapped in a corresponding object pascal class declaration with associated properties and methods.

Some small extensions have been introduced which make it easier for a pascal programmer (and especially a Delphi programmer) to use the toolkit. These have been put in a separate unit, to make the OOP layer resemble the original GTK interface as much as possible.

The result is an easy-to-use class library which allows for fast development of GUI programs. It does not aim to be Delphi VCL or CLX compatible, although it is clearly aimed at people who have experience with Delphi programming. People having coded applications in Delphi will feel at home in FPgtk.

2 Using FPgtk

All the widget class functionality has been put into a single unit: fpgtk. It contains the definitions of all classes. The naming scheme used for all classes, calls and types is quite simple:

1. All underscores have been removed from names. Furthermore, the `gtk` prefix for functions that act as methods has been removed. So the function `gtk_signal_connect` becomes the `SignalConnect` method of the `TFPGtkObject` class.
2. Type names are prepended with `FTP`, so the class name of a GTK button becomes `TFPGtkButton`
3. `new_` functions are replaced by constructors.

To eliminate possibly typos and to make it easier to connect signals, for each signal a method has been introduced to connect a handler to this signal:

1. Connecting to a signal with name `SignalName` happens with a function `ConnectSignalName`, so connecting to the '`clicked`' signal would be done using a function `ConnectClicked`.
2. Most signal handlers are of type `TFPGtkSignalFunction`, which is declared as

   ```
   TFPGtkSignalFunction =
     procedure (Sender:TFPGtkObject; Data:pointer) of Object;
   ```

   which is the FPgtk equivalent of Delphi's `TNotifyEvent` type.

Some extensions have been implemented in a separate unit called fpgtkext ('FPgtk extensions'). These extensions include:

1. Menu item creating functions which allow easy creating of menus.
2. Some functions for accelerator keys.
3. Some scrollable versions of controls. For example, the standard `GtkText` widget (comparable to `TMemo` in Delphi) has no scrollbar support, and the `TFPGtkText` class also does not implement it. The `TFPGtkScrollText` class implements scrollbar support for the text widget.
4. The `MessageDlg` functions of Delphi have been implemented.
5. a `TFPGtkApplication` class, to encapsulate the main GTK event loop. It has some of the functionality that `TApplication` in the VCL offers; Most notably in case of an exception it catches the exception, displays a message dialog with the exception text, and runs the GTK loop again till it exits normally.

With these things kept in mind, a `FPgtk` application can be developed very fast. As an example, some pieces of a rudimentary notepad replacement are presented. The complete source for the application can be found on the CD-ROM accompagnying this issue. The application is developed similar to a Delphi application: The same structure of the application is followed.

The main application will look something like this:

```
{$mode objfpc}
{$H+}
{$apptype gui}
program fpde;

uses FPgtk,fpglib,FPgtkext,frmmain;

begin
  application:=TFPGtkApplication.Create;
  application.MainWindow := TMainForm.Create;
  application.Run;
  application.Free;
end.
```

Looking at the source code of a Delphi project, one will see that this is very similar indeed.

The main form is implemented as a separate class, in a separate unit, as it would be when designed in the Delphi IDE. Its declaration is similar to a declaration of a form as it would appear in Delphi:

```
Type
  TMainForm = Class(TFPGtkWindow)
    FVBox : TFPGtkVBox;
    FFile,
    FFileNew,
    // ...
    FHelpAbout : TFPGtkMenuItem;
    FMainMenu : TFPGtkMenuBar;
    FEditor : TFPGtkScrollText;
    Procedure CreateWindow;
    // ...
    Procedure FileNewClick(Sender : TFPGtkObject; Data : Pointer);
    Procedure FileSaveClick(Sender : TFPGtkObject; Data : Pointer);
  Public
    Constructor Create;
  end;
```

The above is of course only a part of the complete listing of methods and fields. As can be seen, the form descends from `TFPGtkWindow`, which is the GTK window class. When the constructor create is called, the inherited constructor must be called with the correct argument;

```
Constructor TMainForm.Create;

begin
  Inherited create (gtk_window_dialog);
  createwindow;
end;
```

The `gtk_window_dialog` argument will be passed on to the `gtk_window_new` function when the GTK object is created. This deviates from Delphi in that the classes do not descend from `TComponent`, and so no 'owner' instance must be passed. In general, all constructors will have the same arguments as the `gtk_XXX_new` calls which create a new GTK widget.

The `CreateWindow` method populates the window with controls. Streaming is not available, so the complete form must be created in code:

```
Procedure TMainForm.CreateWindow;

Var
  FAccelGroup : Integer;

begin
  FVBox:=TFPGtkVBox.Create;
  FAccelGroup:=AccelGroupNew;
  FFileNew:=NewMenuItem(SMenuFileNew,'','',
                        MakeAccelKeyDef(Self,FaccelGroup,
                                        GDK_N,[amcontrol]),
                        @FileNewClick,Nil);
  // ...
  FHelpAbout:=NewMenuItem(SMenuHelpAbout ,'','',@HelpAboutClick,Nil);
  FHelp := NewSubMenu(SMenuHelp,'','',[FHelpAbout]);
  FMainMenu:=NewMenuBar([FFile,FEdit,FHelp]);
  FEditor:=TFPGtkScrollText.Create;
  Feditor.TheText.ConnectChanged(@EditorChanged,Nil);
  FVBox.PackStart(FmainMenu,False,False,0);
  FVBox.PackStart(FEditor,true, true, 0);
  ConnectDeleteEvent(@OnDeleteEvent,Nil);
  Add(FVBox);
  SetUSize(640,480);
  SetCaption;
  FEditor.TheText.GrabFocus;
end;
```

The above code shows how the various elements of the screen are created and grouped together (most of the menu items have been dropped for clarity).

Note the use of the `NewMenuItem` function of the fpgtkext unit, which creates and fills a GTK menu item with all needed labels, with optional support for shortcut keys. Likewise, the `NewSubMenu` and `NewMenuBar` functions come from the FPgtkext unit, and are meant to make creation of menus easier.

The `HelpAboutClick` and `EditorChanged` callback functions are normal event handlers as they would appear in any Delphi program:

```
Procedure TMainForm.EditorChanged(Sender : TFPGtkObject;
                                  Data : Pointer);
begin
  If FModified<>True then
    begin
    FModified:=True;
    SetCaption;
    end;
end;

Procedure TMainForm.HelpAboutClick(Sender : TFPGtkObject;
                                   Data : Pointer);
begin
  With TAboutForm.Create do
```

```
    Execute(Nil,Nil,Nil);
end;
```

What makes these method different from their Delphi counterparts is the presence of the `Data` pointer. The `Sender` argument can be used just as in Delphi: in the case of the `HelpAboutClick` handler, the `Sender` argument will contain the `FHelpAbout` object.

3 Modal dialogs

In the previous series of GTK articles it was shown that GTK does not provide an easy method for handling modal windows. Likewise, and as can be seen from the previous listing, a `ShowModal` call for the `TFPGtkWindow` class does not exist - instead the `Execute` method exists. The reason to implement this method is the behaviour of GTK: When the user clicks the 'close' button of the window manager, GTK will immediatly destroy the Window object - including the associated FPgtk class.

This means that if the `ShowModal` call would be implemented, the window would have been destroyed if the call returns and would no longer available be in memory to read for instance the contents of an edit widget on the form. Since this is not desirable nor useful, the `Execute` method was implemented instead for the `TFPGtkWindow` class. It's declared as follows:

```
function Execute (anOnDialogInit:DialogInitCallBack;
                  anInitData:pointer;
                  anOnDialogResult:DialogResultCallBack) : integer;
```

The function accepts 3 functions: An initialization callback, a pointer to initialization data, and a result callback. The first callback is called when the form is shown. The last callback is shown when the form is closed, before it is destroyed and removed from memory. This callback must be used to retrieve any needed data from the form.

When the function returns, the form is destroyed and the function result is the 'modalresult' of the form. The modal result is also passed on to the resultcallback.

For an example how this should be used in an application, see the `GetFileName` method in the application source. It will open a GTK filename dialog and retrieve the filename through the result callback:

```
Function TMainForm.GetFileName(ATitle : String) : String;

var
  FS : TFPGtkFileSelection;

begin
  Result:='';
  FS := TFPGtkFileSelection.Create (gtk_window_dialog);
  with FS do
    begin
    Title:=ATitle;
    OKButton.ConnectClicked (@(CloseWithResult), inttopointer(drOk));
    CancelButton.ConnectClicked (@(CloseWindow), nil);
    if Not execute (nil, @Result, @DialogSetFilename) = drOk then
      Result:='';
    end;
```

```
end;
```

The `OKbutton` property of the fileselection dialog refers to the `OK` button on the file selection form. It is connected to the standard `CloseWithResult` method of `TFPGtkWindow`. This method will close the form, and will set the return value to `drOK`, a value which must be passed to the `ConnectClicked` method.

The dialog result callback looks as follows:

```
Procedure TMainForm.DialogSetFilename(Sender : TFPGtkWindow;
                                      Data : Pointer;
                                      Action : Integer;
                                      Initiator : TFPGtkObject);

type
  PString = ^AnsiString;

begin
  PString(Data)^:=(Sender as TFPGtkFileSelection).Filename;
end;
```

The `Sender` will be the filename dialog, `Data` is what has been passed in the second argument to the `Execute` call. In this case, it is the address of the string where the filename should be stored. Other methods could be used to store the filename of course, but this is a case which will occur quite often. The `Initiator` will be set to the object that initiated the callback (by default the form itself). `Action` will contain the modalresult of the form.

The final result is displayed in figure **??**. It shows the editor after the user clicked the 'close' button: The displayed message is put on the screen using the `MessageDlg` function implemented in the fpgtkext unit, and which works identically to the version implemented by Delphi.

4 Accessing the GTK layer

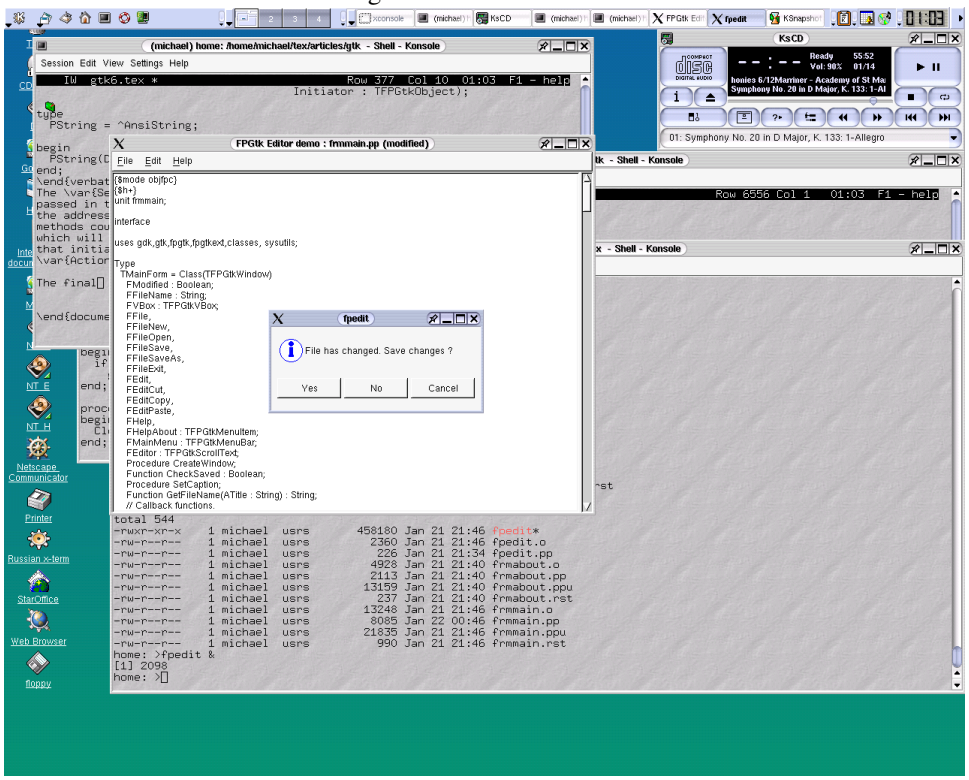
Each class implements a wrapper around a GTK widget. Care has been taken to convert most (if not all) GTK calls to methods of the classes. However if something was omitted, or the programmer thinks the way it was wrapped is not comfortable, the underlying GTK object is still available: Each class has a function called `TheGTKObject` which returns a typed pointer to the underlying GTK object, for example:

```
TFPgtkAdjustment = class (TFPgtkData)
  // ...
Protected
  function TheGtkObject : PGtkAdjustment;
```

So the class can be subclassed, and the pointer can be used to manipulate the underlying GTK object if this should be necessary.

There may be cases when this is needed: The GDK layer has not been wrapped in a OOP layer, since the focus lies mainly on fast development of GUI programs using the everyday GTK widgets. However, for some programs it may be needed to drop down to the GDK layer and do some custom drawing. For this the GTK widget pointer may be needed, so it was made available. It is not excluded that the GDK API itself will be wrapped in a OOP layer as well in a later stadium, although no plans for thsi exist at the moment.

Figure 1: Editor in action.



5 Conslusion

The FPgtk OOP layer around the GTK toolkit makes the programming of portable GUI programs in Free Pascal even more easy. While it definitely does not claim to provide the functionality of the VCL, it does allow to develop complex GUI applications in a very short time. Since its initial inclusion in the Free Pascal supported packages, it has been used to create an GUI editor for the free pascal source documentor (fpdoc). The FPgtk layer itself is generated from source templates: the editor for these templates itself is a fpGTK application, rewritten from an initial Delphi program. Lastly, a debug message server similar to the one that comes with the gexperts toolkit for Delphi has been written in FPgtk. Writing all these programs required little more effort than would have been required when writing them in Delphi with the VCL.