Getting started with Lazarus
Forms and Controls

Michaël Van Canneyt

July 2, 2006

Abstract

Lazarus is a cross-platform 2-way RAD tool which can be used to develop almost any kind of program for Windows, Linux, Solaris or some of the BSD distributions. The focus is on GUI applications. In this second article, the basics of GUI design will be explained.

1 Introduction

Basically, all lazarus GUI projects have 2 things in common: they use a `TForm` descendent, and many descendents from `TControl`.

In this article, a broad overview of the hierarchy of components is presented, ending with `TForm`, which is the basis for all program. After this, the workings of `TForm` are explained.

Basically, all controls and non-visual components used in Lazarus descend from the following 4 types:

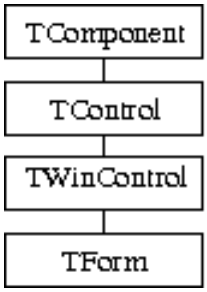
TComponent this is the parent of all objects which can be manipulated in the Lazarus IDE: The streaming system (i.e. the process of writing all published properties of a class to a stream) of Lazarus starts at `TComponent`. `TComponent` introduces the idea of ownership: a component 'owns' it's children: when a component is freed, it's children are freed as well. This is a rudimentary form of automatic memory management. In Lazarus, all components dropped on a form are owned by the form.

TControl All visual components descend from this. `TControl` descends from `TComponent` introduces the idea of position and size, plus the 'parent' relation, which is a visual relation: a control can be dropped on a parent control, meaning that it's visual position is measured relative to the parent control. When a parent control is hidden (made invisible), it's visual children are hidden as well. Ultimately, all controls are parented on a form. Note that the parent-child relation is different from the owner-owned relation, although they coincide for control dropped directly on the form.

TWinControl This is a descendent from `TControl` which has an own window handle associated with it. The window handle is an object which is obtained from the underlying GUI system (Windows, X-Windows). A `TControl` descendent which is not a `TWinControl` (i.e. has no window handle) will draw itself on the `TWinControl` which is it's parent.

TForm This is a descendent from `TWinControl`. It is the basis for all forms made in the Lazarus IDE. It owns all controls that are dropped on it, directly or indirectly, and is the starting point of the streaming system: Lazarus always streams a complete form.

Figure 1: Main items in hierarchy of components



The relation between these 4 components is depicted in figure 1 on page 2

Since `TForm` is the basic of all GUI programs, this will be discussed first.

2 `TForm`: The foundation of GUI programs

All Lazarus programs consist of one or more `TForm` descendents. In difference with all other GUI controls commonly used in Lazarus programs, `TForm` is the only class which is actually subclassed for each type of window: One will never instantiate an instance of `TForm` itself, but always a descendent. For all other controls (buttons and so on) an instance of the standard class (TButton) is instantiated.

However, all this is managed for the developer, by the Lazarus IDE.

For each window in the application, Lazarus creates a `TForm` descendent, which is the form that is designed in the IDE. It also creates a global variable with the same name as the form. Each form is in it's own unit (the name of the unit can be chosen). By default, instances of all forms are created when the program starts: this means that all global form variables point to a valid instance of a form. Only the main form is shown initially, however. All other forms are hidden. The main form form is the first form that is created. When the main form closes, the program exits.

It is possible to tell Lazarus for which forms it should create an instance at project startup. This is done in the 'Project Options' dialog (menu item **'project|Project options'**). There, a choice can be made which forms should be auto-created. See figure 2 on page 3. The topmost form in the list of forms to be auto-created will be the main form of the application. (from this it follows that at least 1 form must be auto-created)

Creating the forms is done in the main program file. The options as set in figure 2 on page 3 will lead to the following code:

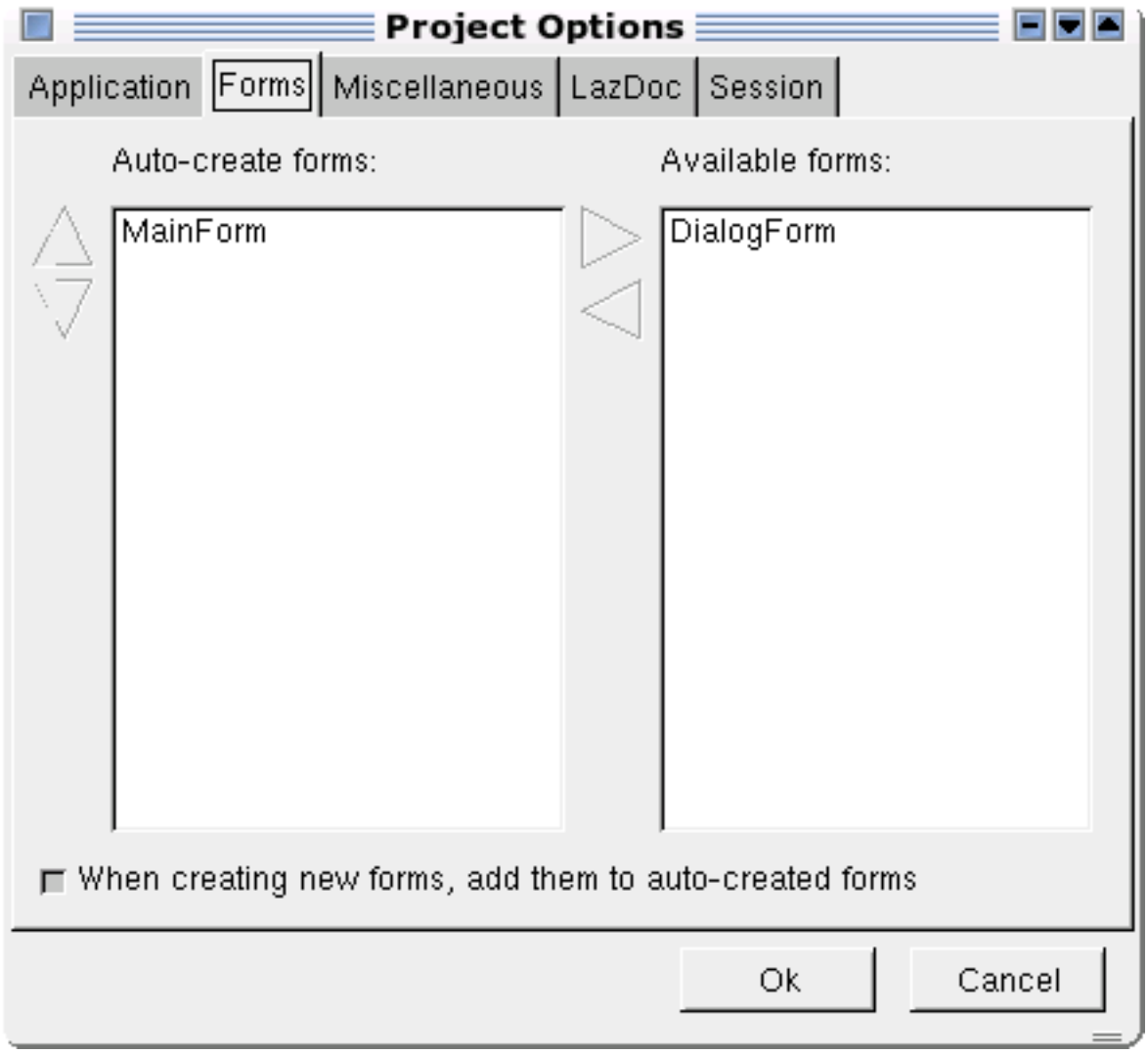
```
program formsdemo;

{$mode objfpc}{$H+}

uses
  Interfaces, Forms
  frmmain,
  frmdialog;

begin
  Application.Initialize;
```

Figure 2: Project options dialog - which forms to create



```
  Application.CreateForm(TMainForm, MainForm);
  Application.Run;
end.
```

The `Run` method of the `Application` class will start the message loop, and will only exit when the `MainForm` form is closed.

3 Showing other forms

So, if an application has multiple forms, how to display a second form to the user ? This depends on whether the form is auto-created by Lazarus or not. To illustrate this, an application with 2 forms is created: `TMainForm` and `TDialogForm`. Both forms will be by default autocreated, but only the `TMainForm` instance is shown. The former is obviously the main form, and the second is the one which will be shown additionally to the user. On the first form, a button is dropped (let's call it `BDefault`), and it's 'OnClick' event handler is filled with the following code:

```
procedure TMainForm.BDefaultClick(Sender: TObject);
begin
  DialogForm.Show;
end;
```

That's it. For this code to compile, the unit of the `DialogForm` must be included in the uses clause of the main form unit. Note that the `Show` call returns immediatly, i.e. there

After compiling and running the application, the main form is shown. When the `BDefault` button is clicked, the dialog form is shown.

Now, obviously, this way, it is not possible to create multiple instances of a form. For this, a separate technique is needed. The multiple instances must be created on an as-needed basis. To demonstrate this, a second button is placed on the mainform (`BPrivate`). It's `OnClick` method is filled with the following code:

```
procedure TMainForm.BPrivateClick(Sender: TObject);

begin
  With TDialogForm.Create(Application) do
    Show;
end;
```

Here, a new instance of the `TDialogForm` is created, and it's `Show` method is called. The `Application` component is given as the owning component of the dialog form. If the application stops executing, the application component is removed from memory, and so are all dialog instances.

The `BPrivate` button can be clicked as often as needed: each time the button is clicked, a new dialog form will be shown.

The `Show` method returns at once. This means that the main form is again in a waiting state (waiting for GUI events), while the dialog is also being shown.

Supposing that the dialog should be shown, and the main form should not process any events any more, or accept any input? In that case the `ShowModal` function should be used. The `ShowModal` does the same as `Show`, but does not return until the user has closed the form. What is more, `ShowModal` is a function that returns an exit code similar to the one of the standard `MessageDLG` function (described in the previous article).

To demonstrate this, a checkbox is dropped on the main form (`CBShowModal`). If checked, clicking the buttons will show the dialog form using `ShowModal` instead of `Show`. The OnClick handlers of the buttons would look like this:

```
procedure TMainForm.BDefaultClick(Sender: TObject);
begin
  If CBShowModal.Checked then
    DialogForm.ShowModal
  else
    DialogForm.Show;
end;

procedure TMainForm.BPrivateClick(Sender: TObject);
begin
  With TDialogForm.Create(Application) do
    If CBShowModal.Checked then
      ShowModal
    else
      Show;
end;
```

To verify that the `ShowModal` function actually returns only after the dialog form is closed, a call `ShowMessage` could be inserted after the `ShowModal` call.

4 Closing a form

Closing a form can be done programatically (calling the `Close` method), but can also be done by the user if he/she clicks the 'close window' icon of the window manager. In both cases, two events are fired:

OnCloseQuery This event can be used to prevent the form from closing. For this, it has a parameter (`CanClose`) which can be used to signal to the LCL that the form should not be closed:

```
procedure TDialogForm.FormCloseQuery(Sender: TObject;
                                     var CanClose: boolean);
begin
  ShowMessage('OnCloseQuery');
  CanClose:=True;
end;
```

If `CanClose` is set to false, then the form will not be closed, even if closed programatically.

OnClose This event is fired after the `OnCloseQuery` event, and only if `CanClose` was True. The `OnClose` event can be used to control what happens to the form when it is closed.

The `CloseAction` parameter of the `OnClose` event handler determines what happens to the form when it is closed. It can be set to any of the following values:

caNone The default action is executed. What this action is depends on the kind of form.

caHide The form is simply hidden (made invisible). This is the default action for most forms which are not the main form.

caMinimize The form is minimized.

caFree The form is destroyed and removed from memory. This is the default action for the main form.

If the form was the default form instance created by lazarus, it is inadvisable to use the `caFree` result: after the form was destroyed, the reference to the form in the global variable is no longer valid.

To show what happens, the `DialogForm` is equipped with a radio group component, which allows to choose between these 4 possibilities: in the `OnClose` event, the chosen `CloseAction` is set:

```
procedure TDialogForm.FormClose(Sender: TObject;
                                var CloseAction: TCloseAction);
begin
  ShowMessage('OnClose');
  Case RGOnClose.ItemIndex of
    0 : CloseAction:=caNone;
    1 : CloseAction:=caHide;
    2 : CloseAction:=caFree;
    3 : CloseAction:=caMinimize;
  end;
end;
```

To show the order of events, the `OnClose`, `OnCloseQuery`, `OnDestroy` and `OnHide` events also get an event handler, which simply shows which handler is being called:

```
procedure TDialogForm.FormDestroy(Sender: TObject);
begin
  ShowMessage('Destroying Dialog Instance');
end;

procedure TDialogForm.FormCloseQuery(Sender: TObject;
                                     var CanClose: boolean);
begin
  ShowMessage('OnCloseQuery for dialog');
  CanClose:=True;
end;

procedure TDialogForm.FormHide(Sender: TObject);
begin
  ShowMessage('Hiding Dialog Instance');
end;
```

Experimenting with the dialog and the various settings will give insight in what events are called, and in which order they are called, depending on the settings.

5 Closing a modal form

If a modal form is closed by the `Close` method or using the close icon of the window manager, the `ShowModal` call which was used to show the form, will return `mrNone` as

result.

To return an other result than `mrNone`, the `ModalResult` property of the form can be set. If the form is shown modally, setting the property will close the form, and will set the result of the `ShowModal` function which was used to show the form.

To illustrate this, a new form (`TModalForm`) is made with a radiogroup which allows to select the value for the `ModalResult` property of the form. The value is set in the `OnClick` method of a button (`BClose`) :

```
procedure TModalForm.BCloseClick(Sender: TObject);
begin
  ModalResult:=RGModalResult.ItemIndex;
end;
```

The code makes use of the fact that `mrNone` equals 0.

In the main form, a button is dropped which executes the following code when clicked:

```
procedure TMainForm.BTestModalClick(Sender: TObject);

Var
  S : String;

begin
  With TModalForm.Create(Application) do
    If CBShowModal.Checked then
      begin
      case ShowModal of
        mrNone     : S:='mrNone';
        mrOK       : S:='mrOK';
        mrCancel   : S:='mrCancel';
        mrAbort    : S:='mrAbort';
        mrRetry    : S:='mrRetry';
        mrIgnore   : S:='mrIgnore';
        mrYes      : S:='mrYes';
        mrNo       : s:='mrNo';
        mrAll      : S:='mrAll';
        mrNoToAll  : S:='mrNoToAll';
        mrYesToAll : S:='mrYesToAll';
      end;
      ShowMessage('Modalresult = '+S);
      end
    else
      Show;
end;
```

Experimenting with this button and the various possibilities will show that if the `Show` method is used, setting `ModalResult` does not close the form at all: the form is only closed when it was shown using `ShowModal`.

6 Positioning a form

When a form is created, it will be positioned on screen by default on the position where it was designed. Obviously, this is not always what is desirable: screen sizes differ, other

windows may already be visible, and dialogs should usually be shown in the middle of the screen, where they attract attention.

To control this, the `Position` property of `TForm` can be used. It can have the following values:

poDesigned The default: the form is positioned where it was designed.

poDefault Lets the window manager decide where to position the form.

poDefaultPosOnly Lets the window manager decide where to position the form, but the size is as designed.

poDefaultSizeOnly Lets the window manager decide how to size the form, but the position is as designed.

poDesktopCenter Positions the form at the center of the desktop.

poMainFormCenter Positions the form at the center of main application form.

poOwnerFormCenter Positions the form at the center of the form that owns it. This is only useful if the form is created in the following way:

```
procedure TMainForm.BPrivateClick(Sender: TObject);
begin
  With TDialogForm.Create(Self) do
    If CBShowModal.Checked then
      ShowModal
    else
      Show;
end;
```

Here, the dialog would be positioned at the center of the main form, because the main form owns the dialog form (it is passed as the owner to the constructor)

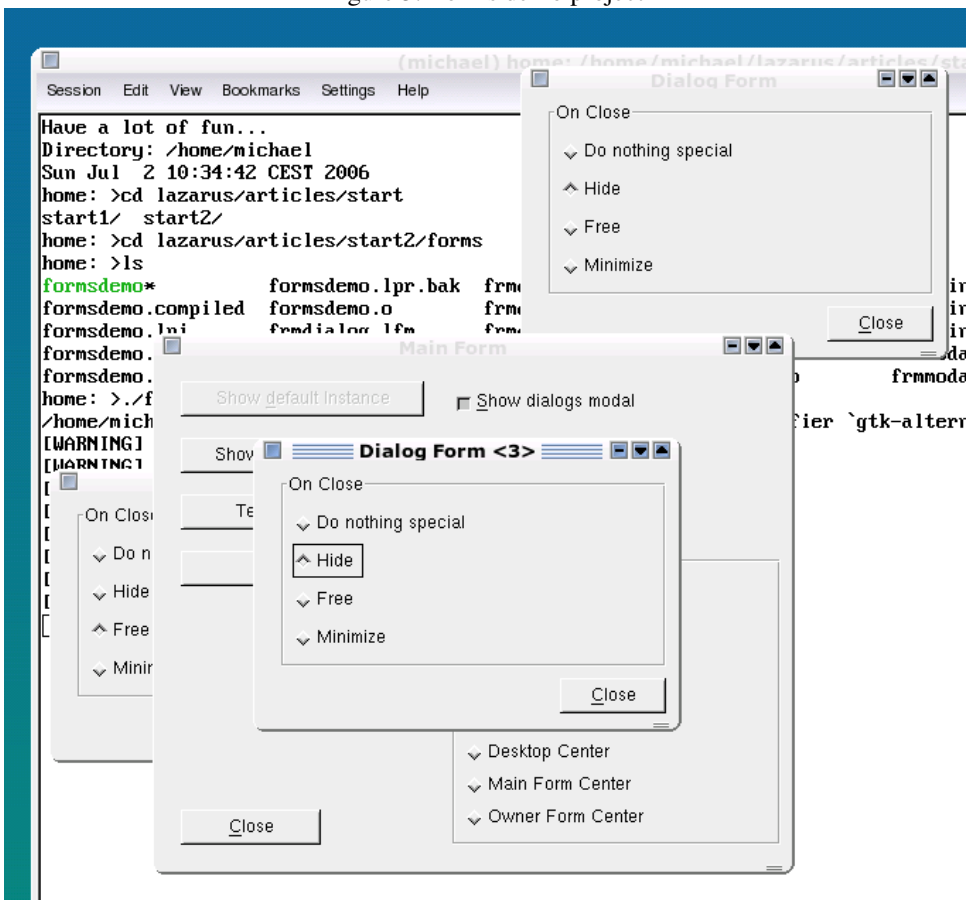
poScreenCenter The form is positioned at the center of the screen, which can be different from the center of the desktop.

The main form of the test application is enhanced with a `BPosition` button, and a radio group which allows to select one of the above possibilities. In the `OnClick` handler of the button, the following code is executed:

```
procedure TMainForm.BTestPositionClick(Sender: TObject);

begin
  With TDialogForm.Create(Self) do
    begin
    Case RGPosition.ItemIndex of
      0 : Position:=poDesigned;
      1 : Position:=poDefault;
      2 : Position:=poDefaultPosOnly;
      3 : Position:=poDefaultSizeOnly;
      4 : Position:=poScreenCenter;
      5 : Position:=poDesktopCenter;
      6 : Position:=poMainFormCenter;
      7 : Position:=poOwnerFormCenter;
    end;
```

Figure 3: Forms demo project




```
    ShowModal;
    end;
end;
```

From this code one can also see that setting the position property can be done just before showing the form, i.e. not just at design time. The final result is shown in figure 3 on page 9.

7 Conclusion

In this second article, the workings of the basic element in all GUI development with Lazarus was handled: `TForm`. It was shown how to create modal and modeless windows, and how to determine how a window was closed. In the next articles, the various controls that can be used on a form, will be examined in more detail.