Docking in Lazarus

Michaël Van Canneyt

March 29, 2014

Abstract

Docking is a feature, available in many applications. Basically, this means that parts of the applications (usually toolbars or menus) can be moved to other locations in a window, or even moved to a separate window. This article discusses the mechanisms to provide docking in an LCL application.

1 Introduction

Traditionally, an application shows available toolbars on the top of the main window. However, when many toolbars are available, the top of the window becomes crowded. On top of this, each user has his own preferences when it comes to toolbar placement: For example, the formatting toolbar on the right, the alignment toolbar on the left, and the drawing toolbar on the bottom of the window, or even having the various toolbars float around. Not only toolbars can be placed, but also other aids when working with documents or data can be moved around on the screen.

To accomodate for all this, docking was invented: this technology allows a user to drag certain parts of the user interface to another location on the screen. Depending on where it is dropped, it is nicely integrated in the window where it was dropped.

The LCL also allows to do this, and in this article the

2 Allowing controls to be dragged

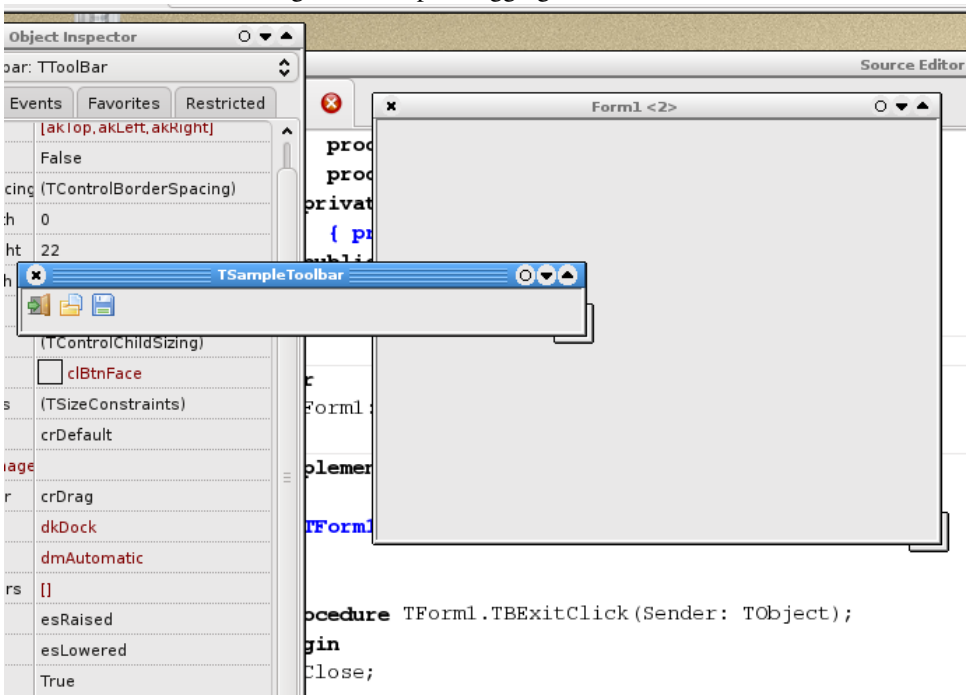
The support for drag and dock is similar to the support for drag & drop: instead of dragging the contents of a control (dragging an item from a listview to e.g. a treeview), the control itself is dragged along the screen, and dropped somewhere else. Therefor, the starting point of drag & dock is similar to the starting point of drag & drop operations.

To make a control dockable, 2 properties are important: 'DragKind' and 'DragMode'. The former is by default set to 'dkDrop', and should be set to 'dkDock', to indicate that when a drag operation starts, the control must be dragged instead of the control contents. (although it is possible to do both, this will be explained below). DragMode should be set to 'dmAutomatic'. This means that the LCL will automatically start a drag operation as soon as it sees that the mouse is clicked and dragged.

That's all there is to it. To demonstrate this, a small demo application can be created: a simple form with a toolbar on it, containing 3 buttons: one to exit the application (TBExit), 2 others (TBOpen,TBSave) which simply show a message:

```
procedure TForm1.TBExitClick(Sender: TObject);
begin
```

Figure 1: Simple dragging of the toolbar




```
  Close;
end;

procedure TForm1.TBopenClick(Sender: TObject);
begin
  ShowMessage('You clicked button '+(Sender as TComponent).Name);
end;
```

The abovementioned properties can be set on the toolbar, and the application can be run. The toolbar can now be dragged off the main form, as shown in figure 1 on page 2. Note that the window on which the toolbar appears has the name of the toolbar component. This will come back later.

When the floating window on which the toolbar resides is closed, the user finds himself without toolbar. This can be easily remedied with a popup menu, or a 'View' menu, in which the toolbar can be shown or hidden. For the simple demo a button is placed on the form, with the following code in the 'OnClick' handler:

```
procedure TForm1.BShowToolbarClick(Sender: TObject);
begin
  TSampleToolbar.Parent:=Self;
  TSampleToolbar.Visible:=True;
end;
```

Clicking the button after the floating toolbar window was closed, will place the toolbar on the top of the window again. In a real application, the menu item would be disabled as long as the toolbar is visible, of course.

3 Docking a control on another control

It is simple to let a control be dragged outside a form and have it floating in a window. So how to dock it along the side of a form, or indeed anywhere on a form? To allow this, many `TWinControl` descendents can be made a Dock site: that means that it can have a control docked on it. By default, controls are not dock sites. They can be made one by setting the 'DockSite' property to 'True'.

The main form itself should not be made a docksite: this would allow the user to drop the toolbar anywhere on the form, which is most likely not the intention of the programmer; Common locations for a toolbar are along the edges of the main window. To provide this, 4 panels are placed on the form, and aligned along the top, left, bottom and right edges of the form: They will get their `Docksite` property set to `True`, and also their `AutoSize` property set to `True`: this will make sure they'll adapt their size to the size of the control docked on them. Note that this will make the panels disappear in the designer. This behaviour is currently normal but different from Delphi. To keep the panels from disappearing in the designer, leave the `AutoSize` property set to `False`, and set it to `True` in the `OnCreate` event of the form:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  PTop.AutoSize:=True;
  PBottom.AutoSize:=True;
  PLeft.AutoSize:=True;
  PRight.AutoSize:=True;
end;
```

Now, the toolbar can be dragged from one edge of the form to another. To make sure that the toolbar adapts it's button placement to match the side of the form to which it is docked, the toolbar's `align` property is set to match the alignment of the panel on which it is dropped. This can be done in the `OnDockDrop` event, which is executed when the control is docked:

```
procedure TForm1.SetDocksiteSize(Sender: TObject;
                                 Source: TDragDockObject;
                                 X,Y: Integer);

Var
  C : TControl;
begin
  C:=(Sender as TControl);
  if Source.Control is TToolbar then
    Source.Control.Align:=C.Align
  else
    Source.Control.Align:=alNone
end;
```

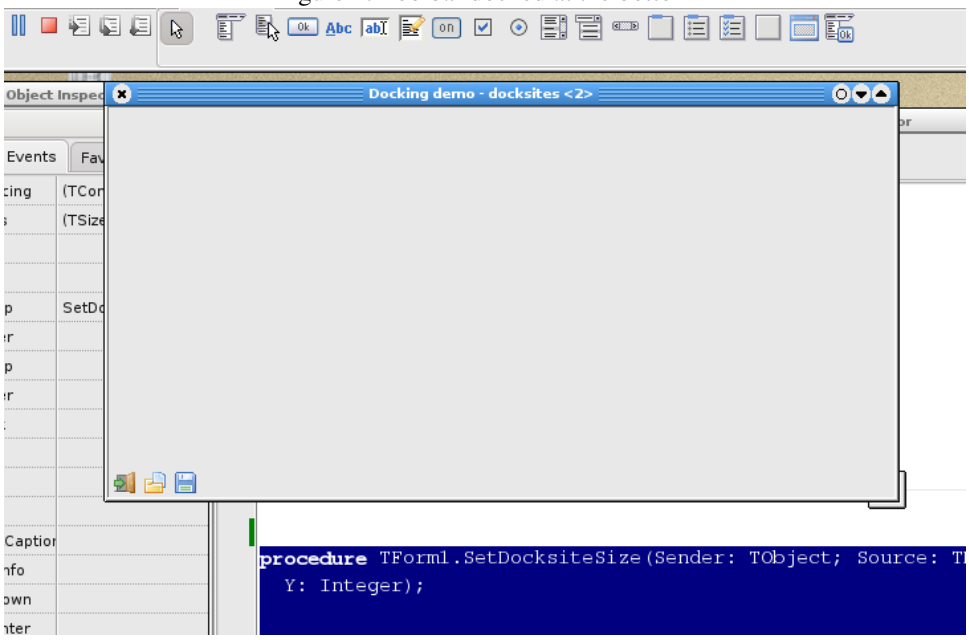
For other controls, the `Align` property is set to `alNone`, so they keep their natural size.

Note the `Source` parameter to this event: it is of type `TDragDockObject`, and describes the drag& dock operation. It has many properties, of which the following are the interesting ones for docking operations:

Control The control which is dragged.

DragPos The starting point of the drag operation.

Figure 2: Toolbar docked at the bottom



DragTarget the target control over which we are currently dragging.

DockRect A rectangle indicating the area where the control will be docked if the mouse is released.

DropAlign the alignment to use when docking the control relative to `DropOnControl`

DropOnControl A control already docked in the docksite, relative to which the dragged control will be docked.

Floating is the control currently floating ?

The result is shown in figure 2 on page 4.

4 Providing feedback to the user

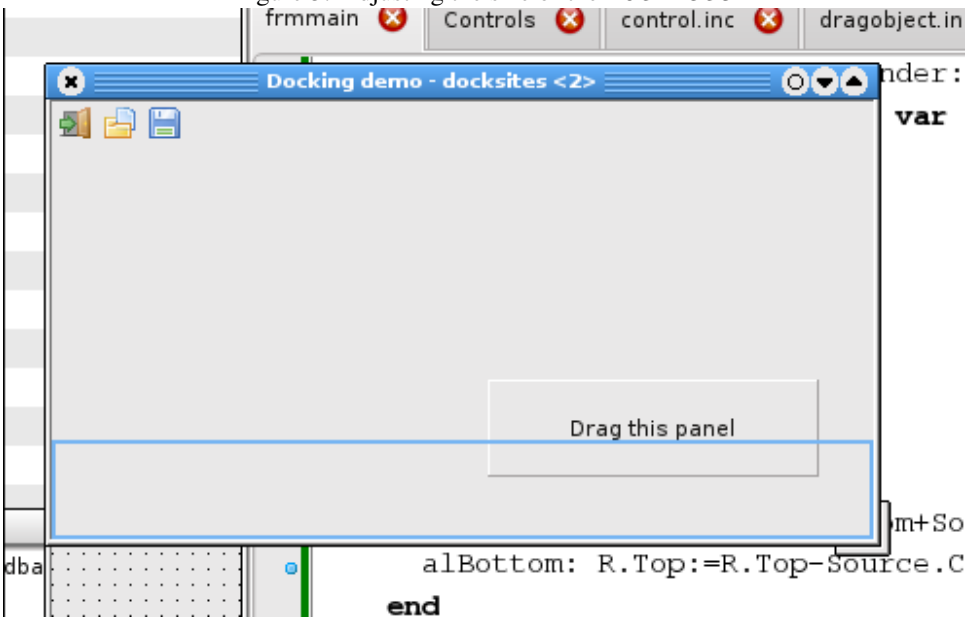
When playing with the sample docksites implementation, an annoying side-effect of the `Autosize` property can be noticed: the docking rectangle - which is usually the contours of the docksite, has width or height zero, because the docksite has no width or height (depending on the side of the form to which the docksite is aligned).

This can be remedied in the `OnDockOver` event:

```
procedure TForm1.PTopDockOver(Sender: TObject;
                              Source: TDragDockObject;
                              X,Y: Integer;
                              State: TDragState;
                              var Accept: Boolean);

Var
  R : TRect;
  C : TControl;
```

Figure 3: Adjusting the size of the `DockRect`




```
begin
  R:=Source.DockRect;
  C:=Sender as TControl;
  If (R.Bottom-R.Top)<=1 then
    Case C.Align of
      alTop    : R.Bottom:=R.Bottom+Source.Control.Height;
      alBottom: R.Top:=R.Top-Source.Control.Height;
    end
  else  If (R.Right-R.Left)<=1 then
    Case C.Align of
      alLeft   : R.Right:=R.Right+Source.Control.Width;
      alRight : R.Left:=R.Left-Source.Control.Width;
    end;
  Source.DockRect:=R;
end;
```

The `DockRect` property of the `Source` object can be modified to provide feedback to the user over the area where the control will be docked. The above code simply increases the rectangle so it has non-zero size.

The toolbar has the width of the form, hence the rectangle will be increased to the size of the form when hovering over the left or right edges. To demonstrate the effect of the above code better, a panel is dropped on the form, and made draggable by setting it's `DragKind` and `DragMode` properties. When dragging the panel over one of the left or right edges, a rectangle is drawn as in figure 3 on page 5.

More feedback can be given: it is, for instance, possible to increase the sensitivity of the docking mechanism: by default, if the mouse moves within 10 pixels of the docking zone (this is a hardcoded value), the docking feedback will be triggered. This zone can be increased (or decreased) or, indeed, the zone can be removed at all: for instance, it is possible to indicate that a docksite doesn't accept panels. This can be done in the `OnGetSiteInfo` event, which is triggered as the control is being dragged. For the panels along the edges of the form, the event can be implemented as follows:

```
procedure TForm1.PLeftGetSiteInfo(Sender: TObject;
                                  DockClient: TControl;
                                  var InfluenceRect: TRect;
                                  MousePos: TPoint;
                                  var CanDock: Boolean);
begin
  CanDock:=DockClient is TToolbar;
  If CanDock then
    Case (Sender as TControl).Align of
      alLeft,alRight : InflateRect(InfluenceRect,20,0);
      alBottom,alTop : InflateRect(InfluenceRect,0,20);
    end;
end;
```

The `CanDock` parameter can be used to indicate that the docksite will accept the control being dragged. The `InfluenceRect` is the bounding rectangle of the dock site, increased with 10 Pixels. The above code adds another 20 pixels to this. If the `CanDock` parameter is `True` (its value on entry) and the control is dragged within the `InfluenceRect`, the docking rectangle will be shown.

Even if the `CanDock` parameter is true, and the docking rectangle is shown, it is still possible to reject the docking of the control in the `OnDockOver` event: if the `Accept` parameter of that event is set to `False`, the docking will be rejected. This can be used to fine-tune the docking: where `OnGetSiteInfo` merely indicates whether a docksite is accepting a control for docking, the `OnDockOver` event can restrict the docking to certain areas of the dock site (as happens, for instance in the Delphi IDE when one tries to dock multiple toolwindows on top of each other).

5 Controlling the start of the drag operation

Playing with the draggable panel on the form will quickly reveal that clicking on the panel will cause it to float. This is because when `DragMode` is set to `dmAutomatic`, the mouse-down event starts the drag operation, and this is rather annoying. Fortunately, this can be remedied with the help of 2 properties of the global `Mouse` object:

DragImmediate If set to `True`, the mouse-down event on any draggable control will start the drag operation. If set to `False`, the drag operation will only start after the mouse has been dragged for `DragTreshold` pixels.

DragTreshold The distance the user must drag a mouse before starting a drag operation when `DragImmediate` is `False`. By default, this is 5 pixels.

To make the panel clickable again, the following code in the form's `OnCreate` event is therefor sufficient:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Mouse.DragImmediate:=False;
  Mouse.DragThreshold:=50;
end;
```

6 Manual drag and dock operations

There is another way in which the panel can be made clickable without tinkering with the `Mouse` object. This is also a more powerful mechanism. The `DragMode` property can be set to `dmManual`, this indicates that drag operations will be started manually using the `BeginDrag` method of `TControl`. This method looks as follows:

```
procedure BeginDrag(Immediate: Boolean;
                    Threshold: Integer = -1);
```

When invoked, this method will start a drag operation if `Immediate` is `True`. If `Immediate` is `False`, then the drag operation will start after the mouse has been dragged for `Treshold` pixels. if `Treshold` is -1, then the `Mouse.DragTreshold` value is used.

This method can now be used to initiate the drag operation in the `OnMouseDown` event of the panel:

```
procedure TForm1.Panel1MouseDown(Sender: TObject;
                                 Button: TMouseButton;
                                 Shift: TShiftState;
                                 X, Y: Integer);
begin
  If (Button=mbLeft) and (ssCtrl in Shift) then
    Panel1.BeginDrag(False,10);
end;
```

As such, there is not much difference with the previous situation, where `DragMode=dmAutomatic`.

It becomes only interesting in case one wishes to switch for example between a Drag & Drop operation and a Drag & Dock operation: The following code will start a Drag & Dock operation only if the `Ctrl` key is pressed while the mouse is dragged. If the `Ctrl` key is not pressed, then a Drag & Drop operation is started:

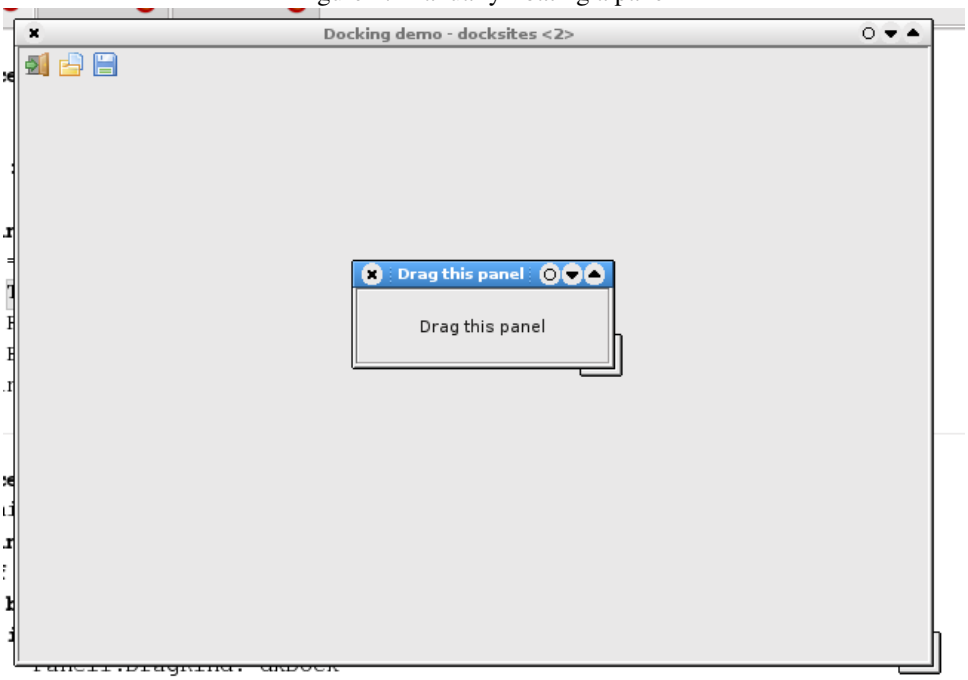
```
procedure TForm1.Panel1MouseDown(Sender: TObject;
                                 Button: TMouseButton;
                                 Shift: TShiftState;
                                 X, Y: Integer);
begin
  If (Button=mbLeft) then
    begin
    if (ssCtrl in Shift) then
      Panel1.DragKind:=dkDock
    else
      Panel1.DragKind:=dkDrag;
    Panel1.BeginDrag(False,10);
    end;
end;
```

Obviously, for a panel there is not much to drag, but code such as the above can be interesting in grids, listboxes, listviews or treeviews where the user can simply drag items, or can drag and dock the control using the `Ctrl` key.

It is not only possible to start a drag operation manually, a control can also be made floating manually, using the `ManualFloat` operation:

```
function ManualFloat(TheScreenRect: TRect;
                     KeepDockSiteSize: Boolean): Boolean;
```

Figure 4: Manually floating a panel




```
else
  Panel1.DragKind:=dkDrag;
Panel1.BeginDrag(False,10);
```

The `TheScreenRect` parameter is the bounding rectangle (relative to the screen) for the control when it is floating; The (optional) `KeepDockSiteSize` parameter determines whether the current docksite should be resized or not (the latter being the default).

This method can be used for instance in a double-click event for the panel:

```
procedure TForm1.Panel1DblClick(Sender: TObject);

Var
  R : TRect;

begin
  R:=Panel1.BoundsRect;
  R.TopLeft:=ClientToScreen(R.TopLeft);
  R.Right:=R.Left+Panel1.Width;
  R.Bottom:=R.Top+Panel1.Height;
  Panel1.ManualFloat(R);
end;
```

Most of this code serves to calculate the size and position of the bounding rectangle for the panel, relative to the screen. The result of this code is shown in figure 4 on page 8

Finally, it is also possible to dock a control on a dock site manually. This can be done with the `ManualDock` method of the control:

```
function ManualDock(NewDockSite: TWinControl;
              DropControl: TControl = nil;
              ControlSide: TAlign = alNone;
```

```
                KeepDockSiteSize: Boolean = true): Boolean;
```

The `NewDockSite` parameter tells the LCL on which docksite control the control should be docked, and is the only required parameter of this call. The `DropControl` and `ControlSide` parameters are optional: if the docking site has already some controls docked on it, then these two parameters can be used to specify a relative position: `DropControl` is the control relative to which the new control will be placed, and `ControlSide` determines where exactly the control will be docked. These parameters correspond to the `DropOnControl` and `DropAlign` properties of the `TDragDockObject` described earlier.

Finally, the `KeepDockSiteSize` can be used to determine whether the docking mechanism will resize the dock site or not. By default, it will keep the size of the dock site.

To demonstrate the use of this, the toolbar can be manually docked in the top panel of the sample program: initially, the toolbar is not docked at all, but is simply placed above the top panel. The following code in the `FormCreate` call will dock the toolbar in the top panel:

```
  Toolbar1.ManualDock(PTop);
```

7 Some docking events

There is another good reason for manually docking the toolbar: Each docking control has an event: `OnUndock`: this event is triggered when a drag and dock operation starts: it is triggered from the `TWinControl` dock site where the control is currently docked. It can be used for instance to prevent a drag&dock operation, as in the following example:

```
procedure TForm1.PTopUnDock(Sender: TObject;
                                                    Client: TControl;
                                                    NewTarget: TWinControl;
                                                    var Allow: Boolean);
begin
   Allow:=(NewTarget<>PBottom)
end;
```

The `Allow` parameter tells the LCL whether the undock operation is allowed or should be prohibited, and the `NewTarget` parameter is the new dock site on which the user wants to dock the control. Note that this events is only triggered when the user actually drops the control somewhere, i.e. when the drag & dock operation is finished, and the LCL attempts to actually dock the control.

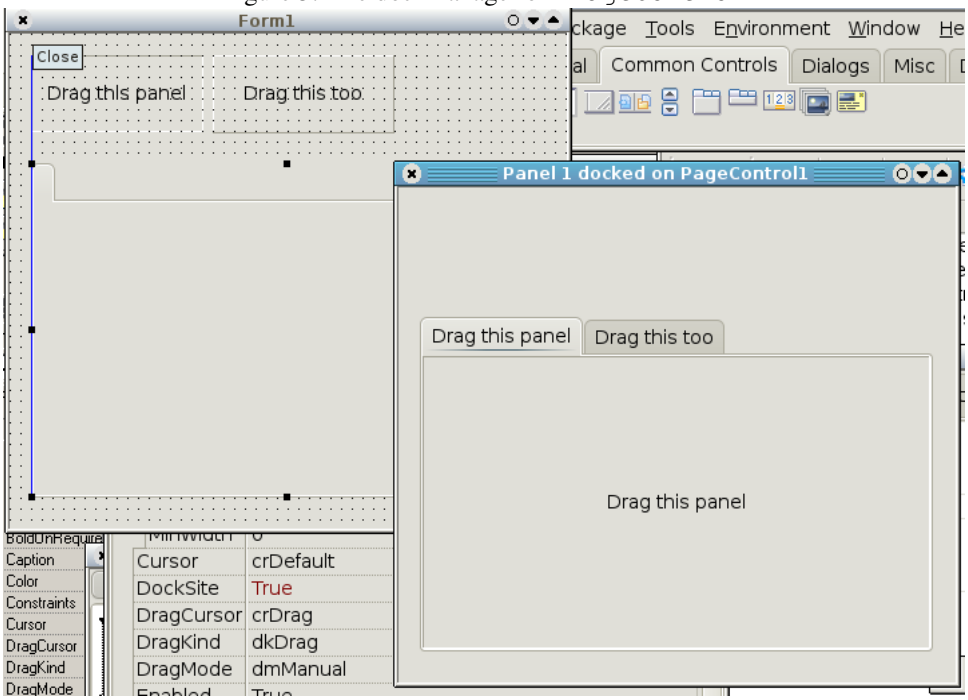
Since the toolbar initially is not docked, no `OnUnDock` event would be triggered when it is docked for the first time. Manually docking it in the top panel will make sure that a `OnUnDock` event is also triggered when the toolbar is re-docked for the first time.

When a control is docked somewhere, the `OnEndDock` event is triggered. This can for example be used to provide some feedback to the user:

```
procedure TForm1Toolbar1EndDock(Sender, Target: TObject;
                                                          X, Y: Integer);
begin
  Caption:='Toolbar docked on '+TComponent(Target).Name;
end;
```

Or one could save the new dock location in a .ini file and restore it in a new program session with `ManualDock`.

Figure 5: The dockmanager of `TPageControl`



8 The DockManager

When playing with the sample program, one will quickly notice a limitation of the default drag & dock mechanism: only one control at a time is visible in a dock site: the LCL just places one control on top of the next in the dock site, and so the last dropped control wil usually be the only visible one.

When a `TWinControl` is a dock site, and a control is docked on it, it uses a Dock manager (an instance of abstract class `TDockManager` to manage the positioning of the docked controls. This behaviour can be disabled with the `UseDockManager` property: if it is set to `False`, no docking manager is used and the docked control is simply made a child control of the dock site.

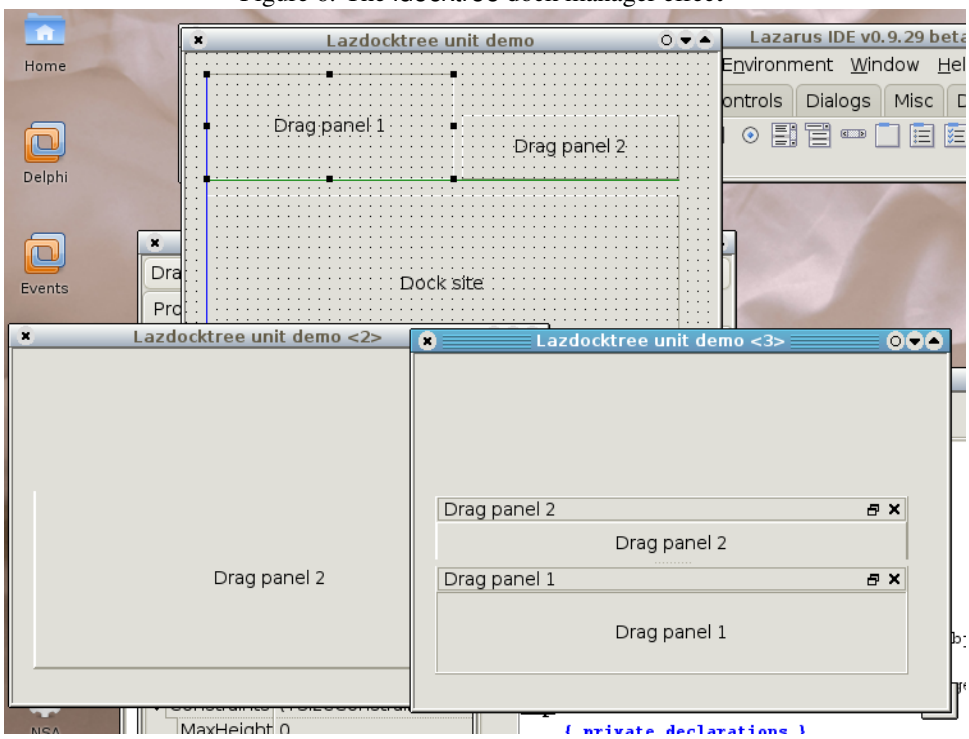
If the `UseDockManager` property is `True` (the default for all controls except `TForm`) then a dockmanager instance is created as soon as a control is docked (if none existed yet). By default, the actual class of the dockmanager instance is determined by the `DefaultDockManagerClass` global variable in unit controls:

```
var
  DefaultDockManagerClass: TDockManagerClass;
```

The standard dockmanager implementation (in class `TDockTree`) does not contain any logic for repositioning of controls on a docksite. Some descendents of `TControl` such as `TPageControl` implement their own dockmanager class to provide customized behaviour: in the case of `TPageControl` the dock manager will dock each dropped control on a new page of the page control. This can easily be demonstrated by creating a form with 2 dockable panels on it, and a `TPageControl` instance which is a docksite. After docking both panels on the dockform, a situation like in figure 5 on page 10 will be created: The original form as in the designer is also visible, to show the difference.

The standard docking class does not do anything special with the controls docked on a

Figure 6: The ldocktree dock manager effect



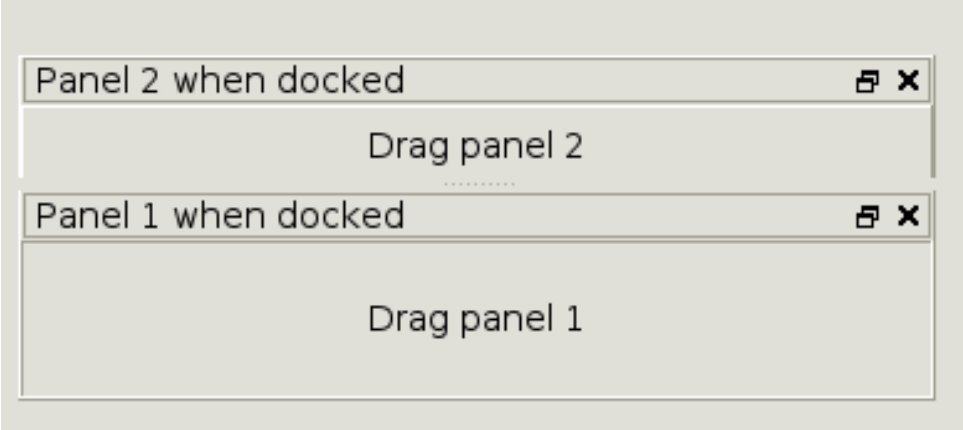
docksite. However, the ldocktree unit implements a dockmanager class (and installs it as the `DefaultDockManagerClass`) which does a lot more than simply parenting the docked controls on the docksite.

To demonstrate this, a small project is made with 2 draggable panels, and a third panel which is a dock zone. The panels are different in size and have a different caption. The program is compiled and run (it is started outside the IDE). Then the source is modified: the ldocktree unit is added to the uses clause, and the program is run again. The result is visible in figure **??** on page **??**. It shows 3 times the same form: At the top is the form in design mode. The bottom-left form is the form with the 2 panels docked, but no docking manager installed: panel 2 was docked last, and is the only visible one. The bottom-right form is from the application which uses the ldocktree unit. It looks markedly different: the docking manager has placed the panels in some kind of mini window, and has placed them in the docksite.

The caption used when creating the mini-docking window is obtained through the 'Text' property of the docked control, but this can be customized: the `OnGetDockCaption` event of the dock site can be used to create custom captions:

```
procedure TMainForm.CreateCaption(Sender: TObject;
   AControl: TControl;
   var ACaption: String);
begin
  If AControl=DragPanel1 then
    ACaption:='Panel 1 when docked'
  else if AControl=DragPanel2 then
    ACaption:='Panel 2 when docked';
end;
```

Figure 7: Custom captions for docking managers



The effect of this can be seen in figure 7 on page 12. It is of course possible to install other docking managers: the examples/dockmanager directory in the Lazarus source tree contains an implementation of a docking manager that implements even other effects; It comes with some demonstration programs which can be compiled with different compiler directives, to show the differences. It also contains a study for adding support for docking to the Lazarus IDE itself. The interested reader should definitely try these examples, a lot about docking and dock managers can be learnt from them.

9 Conclusions

This article has attempted to show that docking in Lazarus is easily implemented: simple docking requires almost no code, while more complicated configurations are possible, but will require some coding: especially sizing of docked controls needs attention. Restoring a docking layout has not yet been covered: this is left for a future contribution. While Lazarus has docking support since quite some time, the sizing and docking mechanisms have been under serious reconstruction: part of the examples given here were compiled with the latest sources of Lazarus, which will be made available on the DVD accompagnying this issue.