Drag And Dock

by Brian Long

Docking support was introduced into the Delphi IDE in Delphi 4. As is often the case when a major new piece of functionality is added to the IDE, it was also made available in the VCL. Indeed, Inprise added docking support to the VCL in order to make it available in the IDE.

An unfortunate aspect of this new docking support is that as you move almost any window around in the IDE, it tries to dock with other windows under the mouse. This can get very irritating (although holding the `Ctrl` key down as you drag windows around will stop the problem) and has the side effect of putting people off adding docking support to their own applications. A rumour suggests that customer demand for a global, persistent option to disable IDE docking may be answered in the next version of Delphi.

Another issue with docking is that the sample docking project (found in Delphi's Demos\Docking directory) is overly complex. It is designed to show how to perform advanced docking, as done by the IDE, where one window can be docked into another window in a variety of ways: horizontally, vertically or as a tabbed page. Figure 1 shows the Delphi editor with a number of debugging windows docked in it. Some are docked in a tabbed area at the bottom left and two are docked above each other to the right of the tabbed area.

This article will look at the VCL docking support, introducing it step by step hopefully to make it clearer how to implement docking support in your own programs. The coverage will include as much of the multitude of properties, methods and events exposed by the VCL as space permits.

The VCL docking support was built on top of the original VCL drag and drop support. Consequently, you might find it useful to have a read of my earlier article *Dragging And Dropping Part 1: VCL* from Issue 56 (April 2000).

Simple Undocking

The typical docking requirement is to have some control like a toolbar which can be docked on a form or floating in a window of its own. Fortunately, such straightforward docking is very simple.

In a fresh application, drop a `TToolBar` on the form and set the normal properties as you like (for me this means remove `ebTop` from `EdgeBorders`), then add a few tool buttons to it from the right-click menu.

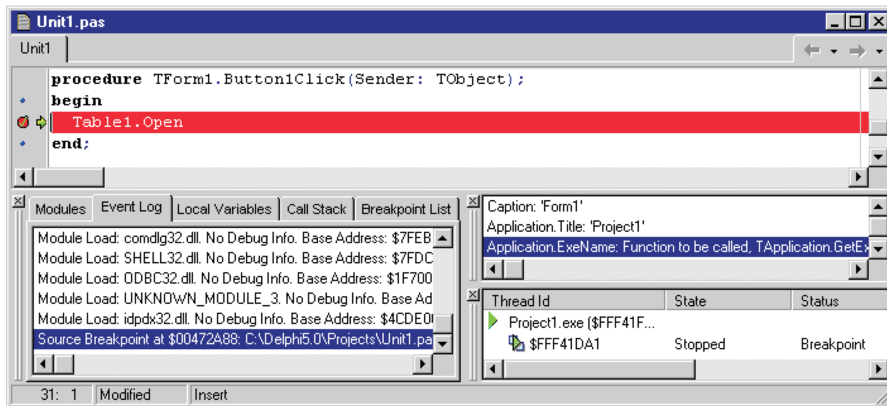
To allow the toolbar to be *ripped off* the form, or *undocked* as we more commonly say, the simplest approach is to set the `DragMode` property to `dmAutomatic` and `DragKind` to `dkDock`.

The `DragMode` setting means that some kind of dragging operation will start automatically when the mouse is clicked down and moved. The `DragKind` setting dictates whether the operation will be a drag and drop operation or a drag and dock operation (we choose the latter).

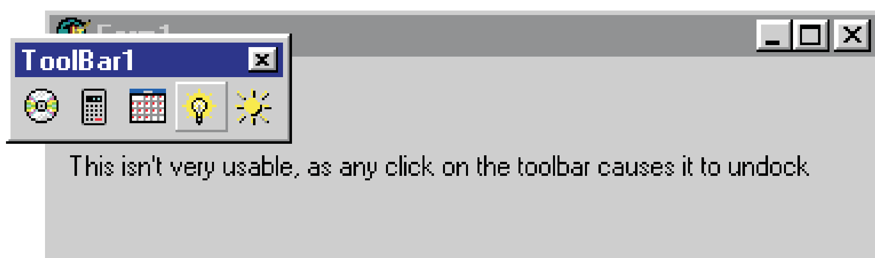
Run the program (on the disk as DockedControls.dpr) and, sure enough, the toolbar can be undocked from the form and left floating on the screen (as shown in Figure 2).

As the toolbar is dragged from the form, a rectangle is drawn to indicate where it would be left if you released the mouse button at any given point (Figure 3). This rectangle is called the *dock image*.

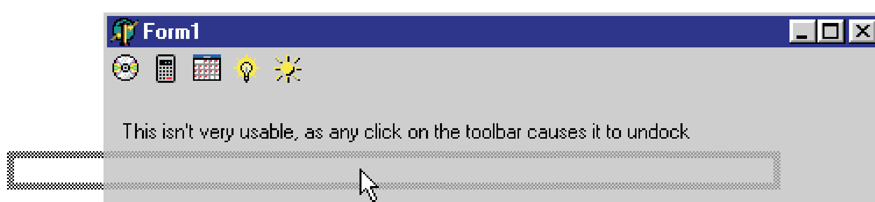
➤ *Figure 1: The editor with various windows docked into it.*



➤ *Figure 2: An undocked toolbar.*



➤ *Figure 3: The dock image being dragged around the screen.*



However, all it takes is the merest mouse click to start the undocking operation and you are left with a floating toolbar. This is part of the problem of the simple solution: the simplest solution is the least flexible.

We really need to allow the toolbar to be clicked without becoming undocked. But that's just one of the problems. Try docking the toolbar back on the form and you won't succeed. Clearly we need to do more to get a realistic docking implementation.

Also, see what happens if you close the floating toolbar form. Unsurprisingly, it closes, but this has the effect of removing the toolbar from the application's user interface. Closing the floating window does not dock the toolbar back again, so we also need to pay attention to that. Anyway, one thing at a time...

Firstly, let's see how we can start the undocking operation in a more usable manner. The problem is caused by automatic drag operations starting immediately upon a left button click, rather than waiting until the mouse has moved a few pixels. To rectify this, set the toolbar's `DragMode` property back to `dmManual`, and make an `OnMouseDown` event handler for it that looks like Listing 1.

```
procedure TForm1.ToolBar1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then
    (Sender as TControl).BeginDrag(False)
end;
```

➤ *Listing 1: A non-immediate undocking operation.*

This makes a drag operation (the operation kind is still dictated by `DragKind`) begin when the left mouse button is pressed, but the `False` parameter ensures the drag operation only starts if the mouse is moved 5 pixels (the value of the global `Mouse` object's `DragThreshold` property).

`BeginDrag` takes an optional second parameter which specifies the number of pixels the mouse must be moved before the drag operation commences, just in case you don't like the default value of 5. You can also change this threshold throughout an application by changing the value of `Mouse.DragThreshold`.

Whilst calling `BeginDrag` is a valid solution for an individual control, a better way of getting deferred undocking throughout the application would be to leave everything as it was (`DragMode` as `dmAutomatic`) and set `Mouse.DragImmediate` to `False` in the form's `OnCreate` event handler (it defaults to `True`).

Simple Docking

To enable a form (or any other `TWinControl` derivative) to support controls being docked in it, thereby being known as a *dock site*, you must set its `DockSite` property to `True`. This allows (by default) any undocked control to be docked in it. Now we have a scheme for acceptable docking/undocking.

```
procedure TForm1.ToolBar1EndDock(Sender, Target: TObject; X, Y: Integer);
begin
  //Make sure toolbar's Align property is reset when it gets docked in the form
  if (Sender is TToolBar) and (Target is TCustomForm) then
    TToolBar(Sender).Align := alTop
end;
```

➤ *Listing 2: Ensuring the toolbar stays aligned to the top of the form.*

First, set `Mouse.DragImmediate` to `False` in the main form's `OnCreate` event handler (or in the project source, noting that `Mouse` is declared in the `Controls` unit).

Then, controls that should be dockable need their `DragKind` property set to `dkDock` and `DragMode` set to `dmAutomatic`.

Lastly, items that should be dock sites need their `DockSite` property set to `True`.

The only real issue with setting the form's `DockSite` property to `True` is that now the toolbar (and any other control) can be docked anywhere in the form. When the toolbar is undocked, and becomes a floating window, its `Align` property is set to `alNone` (whereas it started as `alTop`).

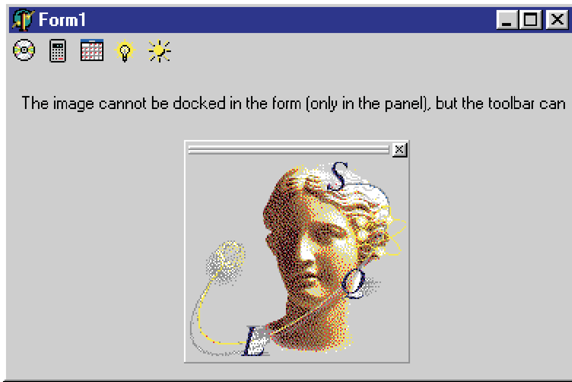
Fortunately, controls have `OnStartDock` and `OnEndDock` events. `OnStartDock` is triggered when any docking operation (undocking or docking) starts, and `OnEndDock` is triggered when the operation ends. You can make a toolbar `OnEndDock` event handler to reset its `Align` property when it is docked back in a form, as in Listing 2. The test program as it stands can be found in DockedControls2.dpr.

➤ *Listing 3: Manually resetting the docking rectangle for a rejected control.*

```
procedure TForm1.FormDockOver(Sender: TObject; Source: TDragDockObject;
  X, Y: Integer; State: TDragState; var Accept: Boolean);
var
  ARect: TRect;
begin
  Accept := Source.Control is TToolBar;
  if not Accept then begin
    //Make mouse position top left of dock rectangle
    ARect.TopLeft := (Sender as TWinControl).ClientToScreen(Point(X, Y));
    //Bottom right is based on dragged control's size
    ARect.BottomRight := TWinControl(Sender).ClientToScreen(
      Point(X + Source.Control.Width, Y + Source.Control.Height));
    //Control was dragged with mouse somewhere other than its top left
    //so use MouseDeltaX/Y fractions to work out the X and Y offset
    OffsetRect(ARect,
      -Trunc(Source.Control.Width * Source.MouseDeltaX),
      -Trunc(Source.Control.Height * Source.MouseDeltaY));
    //Reset dock image to look as though docking will not occur
    //(which it won't) as we rejected the control
    Source.DockRect := ARect;
  end
end;
```

Permission To Dock

We can now start to think bigger. Maybe, for example, you have a dock site, such as a form, that is



➤ *Figure 4: The image component with a grabber added.*

only intended to host toolbars and nothing else. Drag and drop connoisseurs familiar with `OnDragOver` and `OnDragDrop` who have browsed the available events will have seen `OnDockOver` and `OnDockDrop` and will probably be thinking that the `Accept var` parameter in `OnDockOver` is the answer.

A dock site's `OnDockOver` event is triggered as a control (which has been set up for docking) is dragged across it. `OnDockDrop` is triggered when such a control is dropped on a dock site.

Well, the `OnDockOver` event's `Accept` parameter will indeed allow any specified control to be accepted or rejected but, as you drag the control over the dock site, the dock image will automatically snap to its perimeter, suggesting the control can be docked there even though it will be rejected. To remedy this, you would need to modify the docking rectangle (a field of one of the event handler parameters) for all rejected controls, which would be tedious.

Listing 3 shows some code that seems to do the job. As you can see, it's a bit of a chore, particularly when you find that the `MouseDeltaY` field of the `TDragDockObject` object is wrongly implemented in Delphi 4 and 5, and returns the same value as `MouseDeltaX` (this has been reported and will be fixed in Delphi 6).

Instead, a better approach is to use another event handler `OnGetSiteInfo`, which is called before `OnDockOver`. This event has a `var` parameter called `CanDock`, which does the job as you would expect, as in Listing 4. The test program (now DockedControls3.dpr) has a dockable image component that defaults to being a child of a panel. If you try and dock it into the form, it will not succeed, but you can dock it back in the panel.

Note that when you do dock it back in the panel, the image component has a *grabber* (a pair of vertical lines) and a close button added above it (see Figure 4). The grabber allows the image component to be undocked easily, and is designed to make it obvious that the image supports docking.

These UI widgets are automatically added upon docking thanks to the panel's `UseDockManager` property being `True`. The form also has this property, but it defaults to `False`, so these adornments aren't automatically given to controls that dock to the form. For the time being, a discussion of the dock manager, as enabled by the `UseDockManager` property, will be left to one side, but rest assured we will come back to it later.

Listing 4 also shows the `InfluenceRect var` parameter of the `OnGetSiteInfo` event. This `TRect` indicates where the mouse can be released in order for the dragged control to be docked. It defaults to being the bounding rectangle of the dock site expanded by 10 pixels in each direction (this value is hardcoded in the `Controls` unit method that calls `OnGetSiteInfo`, in a constant which is called `DefExpandoRect`).

The implication of this is that, as you drag a control across to a possible dock site, you are able to release the mouse whilst it is 10 pixels outside the dock site and still have the control docked. If you wanted to be choosier about where the mouse should be, you can adjust the rectangle in the event handler.

Control Bars And Docking

At this point, before moving on to more involved areas of docking, some comments should be made about the `TControlBar`. This component is found on the `Additional` page of the Component Palette, and acts a bit like a purely Delphi-written simplified version of the `TCoolBar` Win32 common control component (from the `Win32` page).

The control bar component is really designed as a handy dock site for toolbars. Any component dropped on a control bar at design-time is automatically drawn in a draggable band. Each band has a grabber on the left that is used to drag the bands around. The Delphi IDE main window uses a control bar to house all its toolbars, the menu and the Component Palette.

`TControlBar` initialises its `DockSite` property to `True`, and has an `AutoDrag` property that dictates whether controls can be undocked simply by dragging them off the control bar with their grabbers (also `True` by default). By placing toolbars in a control bar, you can forget about setting the `DragMode` property or setting `Mouse.DragImmediate`, although you must still set `DragKind`.

The test program is now in DockedControls4.dpr and has a control bar on the form with the toolbar in it. `DragKind` is the only docking-related property that is set in the toolbar, and all the docking-related event handlers have been removed. The `OnCreate` event handler is all that remains.

Closing Undocked Controls

Earlier, I highlighted the issue of closing a floating control (a control in an undocked state). The actual effect of closing the floating window is that the control's `Visible` property is set to `False`. So to rectify the problem, you need to

➤ *Listing 4: Rejecting a control more concisely than in Listing 3.*

```
procedure TForm1.Panel1GetSiteInfo(Sender: TObject; DockClient: TControl;
  var InfluenceRect: TRect; MousePos: TPoint; var CanDock: Boolean);
begin
  CanDock := DockClient is TToolBar
end;
```

have some way on the form of setting `Visible` back to `True`.

The Delphi IDE has the same issue with its toolbars. If you undock any of them and then close them, you need some way of making them visible. Right clicking on any of the existing IDE toolbars or the control bar brings up a popup menu with an entry for each toolbar. All visible toolbars have a checkmark next to their corresponding menu item. Selecting any of these menu items toggles the checkmark, and also the toolbar's `Visible` property.

To replicate this behaviour, the test project (DockedControls5 .dpr) has a menu on it and an action list. The action list defines an action whose job is to toggle the state of the toolbar's visibility. The action's event handlers are shown in Listing 5 and you can see a menu item that is connected to the action in Figure 5.

Whilst this approach works just fine, it is not the only option for keeping a floating control accessible. We will come back to this subject later in the article.

Programmatic Docking

Sometimes you will want to take control of the docking/undocking operation in code, rather than leaving it to the user. Controls have `ManualDock` and `ManualFloat` methods to help out here.

`ManualDock` takes a dock site to dock the control into. `ManualFloat` floats a control and takes a screen rectangle in which the floating control will be displayed. There is also a read-only `Floating` property to tell you whether the control is currently docked in a control or in a floating window.

```
procedure TForm1.actToggleToolbarExecute(Sender: TObject);
begin
  ToolBar1.Visible := not (Sender as TAction).Checked
end;
procedure TForm1.actToggleToolbarUpdate(Sender: TObject);
begin
  (Sender as TAction).Checked := ToolBar1.Visible
end;
```

➤ *Listing 5: An action to keep toolbars accessible.*

```
procedure TForm1.btnToggleFloatClick(Sender: TObject);
begin
  if ToolBar1.Floating then
    ToolBar1.ManualDock(ControlBar1)
  else
    //ToolBar1.ManualDock(nil)
    ToolBar1.ManualFloat(Rect(Left, Top, Left+ToolBar1.UndockWidth,
      Top+ToolBar1.UndockHeight))
end;
```

➤ *Listing 6: Manually docking or undocking a control.*

Listing 6 shows some code from a button in DockedControls6.dpr. If the toolbar is already floating it is docked into the control bar. If it is not floating, it is told to float at the top left of the main form. Notice the `UndockWidth` and `UndockHeight` properties that record how wide the control was the last time it was floating. Also notice the comment shows that passing `nil` to `ManualDock` is another way to float the control.

This current version of the project also manually docks the image component in the panel in the form's `OnCreate` event handler to force the grabber and close button to appear immediately.

Manufacturing Dock Zones

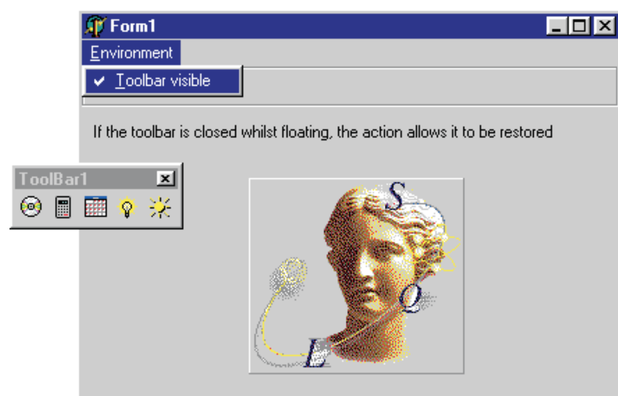
It is common for controls such as toolbars to be docked at the edge of the form. The normal starting place would be the top of the form, but you might want to support docking it to the left, right and bottom of the form, but not anywhere else. This is achieved in Delphi by aligning some 'hidden' dock site controls to the left, top, right and bottom of the form.

For example, four panels would work as dock sites. Drop them on the form, set `BevelOuter` to `bvNone`, set `Align` appropriately and both the `DockSite` and `AutoSize` properties to `True`. This makes the panels effectively disappear as they have no child controls to accommodate. The top and bottom aligned panels attain a zero height, but have a width that matches the form's client area. The left and right aligned panels have a zero width, but have a height that matches the form's client height. These panels will now be referred to as *dock panels*.

You might think that this would make it difficult to dock anything in the dock panels, but remember the point made earlier. The default influence rectangle is the dock site's bounding rectangle expanded by 10 pixels in all directions. Therefore, as any toolbar is dragged within 10 pixels of the form's border (and therefore within 10 pixels of one of the dock panels), it can be successfully docked in the panel.

DockedControls7.dpr has these changes made, along with a few more. Firstly, the form's `OnCreate` event handler manually docks the toolbar in the top panel (`TopDockPanel`).

Also, all the dock panels share a couple of event handlers. Their `OnGetSiteInfo` event handler ensures only toolbars can be docked in them. Their `OnDockDrop` event handler sets the toolbar's `Align` property appropriately depending on which dock panel it is dropped (in fact, the



➤ *Figure 5: Ensuring the toolbar can be retrieved if closed.*

panel's `Align` property is copied to the toolbar). This means that when the toolbar is docked at the left or right side of the form it lays out its buttons vertically.

Whilst the program works, it has an aesthetic downside. When dragging the toolbar near a panel, the dock image snaps to the boundaries of the panel in question. Unfortunately, the panel either has no height (in the case of the top and bottom dock panels) or no width (left and right panels) and so the dock image just looks like a thick grey line.

What is needed is for it to suggest the outline of where the toolbar would be if it were docked. To do this we need code in each panel's `OnDockOver` event handler to modify the `TRect` passed as a field of the `TDragDockObject` object parameter.

DockedControls8.dpr has been modified to include this sort of code (see Listing 7). The code first works out how much the dock image will need to be extended by, one way or another. This is either the height of a horizontal toolbar or the width of a vertical one. Depending which dock panel is being dragged over (or near), the dock image is extended in an appropriate way.

You could leave it at that, but there is still an imperfection. Because the dock panel currently occupied by the toolbar has a certain height (or width), it means that the two dock panels aligned to each side of it do not stretch along the entire sides of the form. For example, when the form starts, the left and right dock panels stretch from the bottom of the form's client area to the bottom of the top dock panel housing the toolbar. If we leave it like this, the dock image will look a little small as you drag the toolbar around the form.

The remainder of the event handler caters for this by ensuring that the dock image in question definitely goes from top to bottom or left side to right side of the form's client area.

The form now appears to have a toolbar that can be docked in one of four *dock zones*. Figure 6 shows what happens when you drag the toolbar near the right hand dock zone. Notice the dock image stretches from the bottom to the top of the form's client area, overlapping the toolbar and current dock panel.

Custom Docking/Undocking

As a component user, there are three useful events that can be used to customise the behaviour exhibited when controls are dragged over, docked in and undocked from a control. These are `OnDockOver`, `OnDockDrop` and `OnUnDock`.

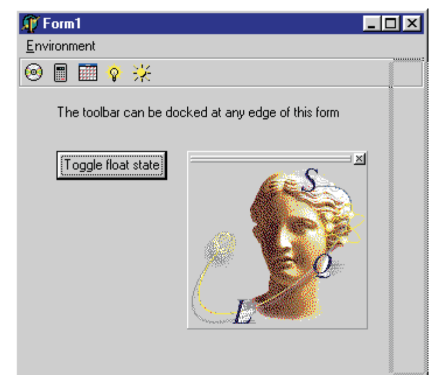
Incidentally, the `OnUnDock` event is only triggered if the control is considered to be docked in a host when undocked. For example, if a toolbar is placed in a control bar at design-time and undocked at runtime, the `OnUnDock` event will not trigger as it will not be considered to be docked to begin with. To overcome this, add a statement in the form's `OnCreate` event handler to manually dock the control to the client in advance (even though it is already a child control).

These events are useful for component users; however, for component writers there are polymorphic methods more appropriate for customising this behaviour. Certainly, each event is called from such a method (`DoDockOver`, `DockDrop` and `DoUnDock`) but there are better methods provided.

To customise behaviour when a control is docked in a `TWinControl`, override its `DoAddDockClient` method, which is passed the control and a `TRect` describing the region where it was docked as parameters. To customise behaviour when a control is undocked, override `DoRemoveDockClient`, which is passed the control.

`TPageControl` is an example of a control that overrides both of these methods to support specialised docking. When a control is docked on a page control, it becomes a new page (a

➤ *Figure 6: Edge dock zones.*



➤ *Listing 7: Enlarging a dock rectangle.*

```
procedure TForm1.DockPanelDockOver(Sender: TObject;
  Source: TDragDockObject; X, Y: Integer; State: TDragState;
  var Accept: Boolean);
var
  DockBar: TToolBar;
  InflateSize: Integer;
  ARect: TRect;
  ClientTL: TPoint;
begin
  DockBar := Source.Control as TToolBar;
  //Get current height if horizontal or width if vertical
  if DockBar.Height > DockBar.Width then
    InflateSize := DockBar.Height
  else
    InflateSize := DockBar.Width;
  //Dock rect will be 0 width/height as per panel dimensions
  //To make it look realistic, increase rectangle
  //to same width/height as toolbar
  ARect := Source.DockRect;
  case (Sender as TPanel).Align of
    alTop: Inc(ARect.Bottom, InflateSize);
    alLeft: Inc(ARect.Left, InflateSize);
    alBottom: Dec(ARect.Top, InflateSize);
    alRight: Dec(ARect.Right, InflateSize);
  end;
  //Two of the four aligned dock panels will not stretch
  //along edge of form client area (because of the currently
  //visible one's size). Make sure dock rectangles do
  ClientTL := Point(0, 0);
  ClientTL := ClientToScreen(ClientTL);
  case (Sender as TPanel).Align of
    alTop, alBottom:
      //Make horizontal panels stretch across form's client
      // width
      begin
        ARect.Left := ClientTL.X;
        ARect.Right := ClientTL.X + ClientWidth;
      end;
    alLeft, alRight:
      //Make vertical panels stretch across form's client
      // width
      begin
        ARect.Top := ClientTL.Y;
        ARect.Bottom := ClientTL.Y + ClientHeight;
      end;
  end;
  Source.DockRect := ARect
end;
```

new TTabSheet). The control can be undocked either by dragging it with the tab or by dragging the control itself.

DockedControls9.dpr now starts off with the image component manually docked in the page control (see Figure 7). It can be undocked and left floating and then docked back in the page control.

The extra code in this version of the program is minimal (see Listing 8). It makes sure that the tab that contains the docked image has a caption, and ensures that toolbar controls cannot be docked in the page control.

More Customisation

In addition to the events and methods I've already discussed, the VCL docking architecture has more up its sleeve for customisation. The OnStartDock and OnEndDock event handlers of a control can be used to set up and free a custom dock object. This is quite like the custom drag objects that are managed for customised drag and drop operations in OnStartDrag and OnEndDrag event handlers.

Without our doing anything, the VCL creates a TDragDockObject when a dockable control starts being dragged. This object is passed to various event handlers, such as OnDockOver (see Listings 3 and 7) and OnDockDrop (Listing 8).

The behaviour of the dock operation can be customised by inheriting a new class from TDragDockObject and assigning an instance of the class to the DragObject parameter of the OnStartDock event handler. You must also ensure that you free this custom dock object in the OnEndDock event handler, despite the online help suggesting this is not necessary.

➤ *Figure 7: The image docked in a page control.*

This custom drag object can do various things such as alter the dock image drawn whilst a control is being dragged around. The DrawDragDockImage and EraseDragDockImage virtual methods simply call the same-named methods in the dragged control by default, to get the standard rectangle (as in Figures 3 and 6). Given this information, you can also customise the dock image of any control by inheriting a new class and overriding the appropriate methods.

The VCL already has one custom dock object class defined for use with dockable toolbars. You might notice that when you undock an IDE toolbar there is no dock image at all. Instead, the toolbar gets physically dragged around the screen as the mouse is moved. This is thanks to the IDE using a TToolDockObject, defined in the ToolWin unit, in the toolbar OnStartDock and OnEndDock handlers.

We can replicate this behaviour ourselves, but it works best when the toolbars normally reside in a control bar. The project DockedControls10.dpr gets rid of all the docking panels and associated code and uses a single control bar instead, as we did earlier.

The control bar has Align set to alTop and AutoSize set to True to make it collapse to zero height

➤ *Listing 8: Ensuring the new tab sheet gets a caption and the page control does not accept toolbars.*

```
procedure TForm1.PageControl1DockDrop(Sender: TObject;
  Source: TDragDockObject; X, Y: Integer);
begin
  if Source.Control = Image1 then
    PageControl1.ActivePage.Caption := 'Athena'
end;
procedure TForm1.PageControl1GetSiteInfo(Sender: TObject;
  DockClient: TControl; var InfluenceRect: TRect; MousePos: TPoint;
  var CanDock: Boolean);
begin
  CanDock := not (DockClient is TToolBar)
end;
```

➤ *Listing 9: Custom toolbar dock objects.*

```
type
  TForm1 = class(TForm)
  ...
  private
    CustomDockObject: TDragDockObject;
  end;
...
procedure TForm1.ControlBar1GetSiteInfo(Sender: TObject; DockClient: TControl;
  var InfluenceRect: TRect; MousePos: TPoint; var CanDock: Boolean);
begin
  CanDock := DockClient is TToolBar;  //Only accept toolbars
end;
procedure TForm1.ControlBar1DockDrop(Sender: TObject; Source: TDragDockObject;
  X, Y: Integer);
begin
  with (Sender as TControlBar) do
    BevelEdges := [beLeft, beRight, beTop, beBottom]
end;
procedure TForm1.ControlBar1UnDock(Sender: TObject; Client: TControl;
  NewTarget: TWinControl; var Allow: Boolean);
begin
  with (Sender as TControlBar) do
    // When last control is being undocked,remove bevelled edges
    if ControlCount = 1 then
      BevelEdges := []
end;
procedure TForm1.ToolBar1StartDock(Sender: TObject;
  var DragObject: TDragDockObject);
begin
  DragObject := TToolDockObject.Create(ToolBar1);
  CustomDockObject := DragObject
end;
procedure TForm1.ToolBar1EndDock(Sender, Target: TObject; X, Y: Integer);
begin
  CustomDockObject.Free;
  CustomDockObject := nil
end;
```

across the top of the form when the toolbar is undocked. Its `OnGetSiteInfo` event handler accepts only toolbars. It also has an `OnUnDock` event handler that removes the bevelled edges when the toolbar is undocked, so as it collapses, it completely disappears. An `OnDockDrop` event handler adds the bevelled edges back when the toolbar is docked again. As mentioned before, to ensure the `OnUnDock` event handler triggers the first time the toolbar is undocked, the form manually docks the toolbar in the control bar when it is created.

Now the `OnStartDock` and `OnEndDock` event handlers can be made for the toolbar. They rely on a `TDragDockObject` data field defined in the form. Listing 9 shows the code discussed so far. When you test out this version of the project, you can see the improvement. As you drag the toolbar out of the control bar, the toolbar instantly undocks and gets dragged around, rather than showing the usual dock image rectangle.

Floating Forms

When a control is undocked from its original parent, we have seen that it becomes a child of a new form that is automatically created (as visible in Figure 2). This form is normally an instance of class `TCustomDockForm`, defined in the `Forms` unit, although toolbars reside in forms of type `TToolDockForm` (from the `ToolWin` unit).

Any given component knows what type of form it will reside in through its public `FloatingDockSiteClass` property. This defaults to `TCustomDockForm` but can be assigned any class inherited from `TWinControl`. Again, you can customise the behaviour of how controls dock in the floating dock site class and undock from it by overriding the `DoAddDockClient` and `DoRemoveDockClient` methods.

Note that controls have a `HostDockSite` property that tells you which windowed control they are docked in (typically an object of the type specified in `FloatingDockSiteClass`). If the control is not docked, `HostDockSite` will be `nil`. If the control is docked, `HostDockSite` will be the same as `Parent`. `HostDockSite` can be written to, but you are advised to call `ManualDock` for better results with programmatic docking.

As an example of a custom floating dock site class, let's go back to a problem from earlier on. Recall that when a floating control is closed, it is hidden. An action was used earlier to allow a menu item to re-show the hidden control.

An alternative solution is to modify the floating dock site class so that when it is closed, the control is automatically docked back in the control bar. Listing 10 shows the implementation of the new class and shows the toolbar's `FloatingDockSiteClass` property being assigned the new value. This code can be found in the DockedControls11.dpr project.

```
TToolBarDockForm = class(TToolDockForm)
protected
  procedure WMClose(var Message: TWMClose);
    message WM_CLOSE;
end;
procedure TToolBarDockForm.WMClose(var Message: TWMClose);
begin
  //When form is closed, dock control back in old dock site
  if (DockClientCount = 1) and (DockClients[0] is TToolBar) then
    TToolBar(DockClients[0]).ManualDock(Form1.ControlBar1);
  inherited
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  Mouse.DragImmediate := False;
  Image1.ManualDock(PageControl1);
  ToolBar1.ManualDock(ControlBar1);
  ToolBar1.FloatingDockSiteClass := TToolBarDockForm;
end;
```

➤ *Listing 10: A custom floating dock site class.*

Multiple Docked Clients

Once you start playing with docking you will see that a lot of thought went into the design and implementation. Here's an example that might make you think about what can be done.

```
procedure TMainForm.FormCreate(Sender: TObject);
const
  Colors: array[1..6] of TColor =
    (clWhite, clBlack, clBlue, clGreen, clRed, clYellow);
var I: Integer;
begin
  for I := Low(Colors) to High(Colors) do
    with TForm.CreateNew(Self) do begin
      Caption := 'Dock me in the main form';
      Color := Colors[I];
      DragKind := dkDock;
      DragMode := dmAutomatic;
      Position := poDefaultPosOnly;
      Width := 100;
      Height := 100;
      Visible := True;
    end;
end;
```

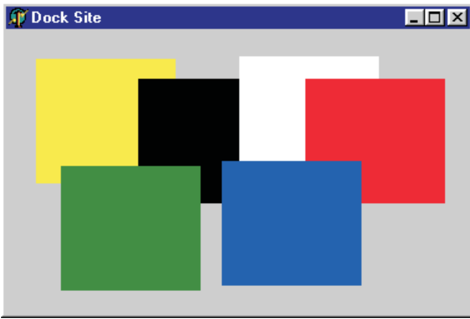
➤ *Listing 11: Creating multiple dockable forms.*

Start a fresh project and set the form's `DockSite` property to `True`, to allow dockable controls to be docked in it. Use the `OnCreate` event handler from Listing 11 to create a number of additional blank, coloured forms that can be docked into the main form. Each of these forms has a caption that suggests docking it into the main form.

Figure 8 shows what happens when you run the program and dock all the coloured forms in the main form. Not that impressive as it stands right now.

The Dock Manager

But if you set the main form's `UseDockManager` property `True` and dock the forms, you get behaviour akin to that demonstrated in the IDE windows, as shown in Figure 9. Each docked control (or form in this case) gets a grabber and close button to start with. Additionally, as you drag a control around the dock site, the dock image snaps to various positions relative to the already docked controls, and all



➤ *Figure 8: Multiple docked forms.*

the docked controls are arranged appropriately based upon the dock image.

The pattern of docked forms shown in Figure 9 is an arbitrary pattern based upon where the mouse cursor was (and hence what shape the docking rectangle was) when each secondary form was docked in the main form. The program used to generate Figure 9 can be found on the disk as MultipleClients.dpr.

Clearly, this `UseDockManager` property proves to be quite useful to know about. It tells the `TWinControl` derivative whether to use a *dock manager* for dealing with controls docked in it.

The dock manager's job is twofold. It is responsible for moving and positioning docked controls within a dock site (the regions occupied by the docked controls are called *dock zones*), and also for drawing the grabber and the miniature close button on the docked control's frame (visible in Figure 9, for example).

When `UseDockManager` is set to `True`, the dynamic protected `CreateDockManager` function method is called and the resultant `IDockManager` interface reference is stored in the protected `DockManager` property (assuming that `DockManager` was `nil`). `DockManager` is also an `IDockManager` interface reference (`IDockManager` is defined in the Controls.pas unit).

When any control requires a dock manager to manage its dock zones, the `CreateDockManager` method uses the `DefaultDockTreeClass` class reference variable, which by default refers to the undocumented `TDockTree` class.

`TDockTree` is by no means the only choice for the dock manager. You can write replacement dock managers by writing a class that implements `IDockManager` (defined in the `Controls` unit). Any individual control can have a custom dock manager by assigning an instance of the class to the `DockManager` property before setting `UseDockManager` to `True`, or by returning an instance of it from an overriden `CreateDockManager` method.

Implementing `IDockManager` from scratch would, however, be a tall order so you could alternatively inherit from `TDockTree` and add or modify behaviour there. If you assign such a class to `DefaultDockTreeClass`, you can replace the dock manager globally throughout your application.

An example of a customised dock manager was given in *The Delphi Clinic* in Issue 51 (November 1999), in an entry called *Customised Docking*. The dock manager was inherited from and two methods were customised to prevent the grabber and close button being drawn at all. It went on to explain how you could either customise one control's dock manager with this new class, or use it as the new default dock manager for the whole application.

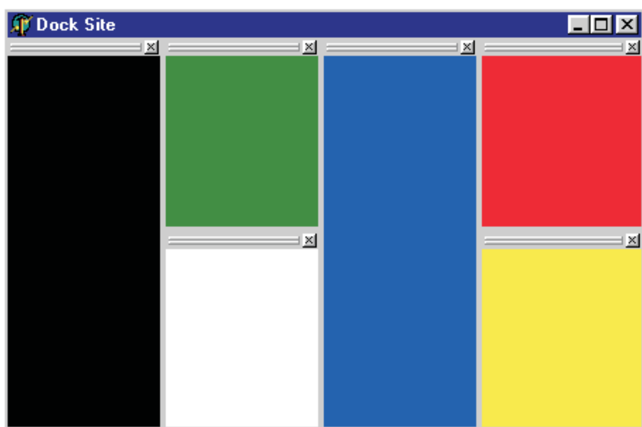
The default dock manager is not without its little problems, though, as I discovered when testing this application for a while. For example, `TDockTree` does not always rearrange the already docked clients correctly when another client is docked to it, as you can see in Figure 10. The windows were docked in the following order: green, blue, black and then white (this fourth one caused the problem).

This problem has been reported (it can be reproduced in the Delphi editor by docking debugger windows into the bottom of the editor in a similar order), but fortunately it can be avoided very easily. Any control that can have multiple controls docked in it (the main form in this case) should call `DockManager.ResetBounds(True)` in the `OnDockDrop` event handler. The updated project MultipleClients2.dpr on the disk contains this change.

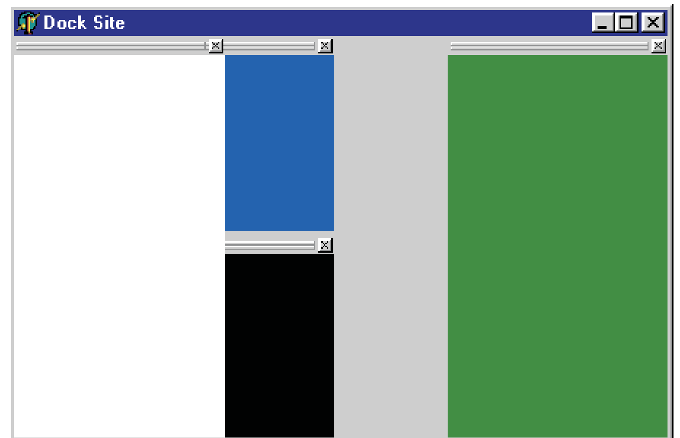
More Docking Subjects

That pretty much fills the space I have been allotted for this issue, but I haven't exhausted the subject of VCL docking yet. Things still to be investigated include custom

➤ *Figure 9: Multiple forms docked using a dock manager.*



➤ *Figure 10: The dock manager going awry. Beware!*



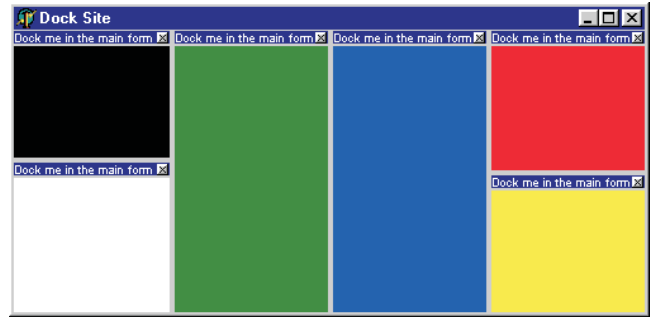
dock managers which, for example, can draw docked clients differently (see Figure 11 for an example). Also, the fancy docking as demonstrated by both the IDE and Delphi's docking demo, where multiple docked clients can reside in a page control. There is also the issue of saving information about docked windows away to a file of some sort, so that when the program restarts, everything can be restored as it was.

I haven't researched these topics yet but, when I do, you can be sure I will write up my findings in another article.

Summary

There is a large portion of the VCL architecture set aside to support drag and dock in Delphi (and C++Builder) applications. Much of this goes unnoticed due to the lack of documentation on the subject in the Delphi product, and the overly complex demo application supplied with Delphi.

This article has looked at a number of issues to do with



➤ *Figure 11: A custom dock manager in action.*

docking and discussed a variety of the properties, methods and events that can be used to build docking support into your programs in a variety of ways.

Whilst researching the content of this article I have found numerous errors in the help, and also many issues which are either undocumented, or unhelpful. For example, I had fully implemented a custom dock object to mimic the IDE toolbar behaviour (no dock image, just immediate undocking) before bumping, completely by chance, into the `TToolDockObject` in the `ToolWin` unit which does the same thing. Whilst this class is indeed documented, there is no reference to it in either the `TToolBar` or `TControlBar` help pages.

Every issue I have found has been reported to Borland, and so far I have been informed that most of the reports will be acted upon for Delphi 6.

Brian Long is a freelance trainer and problem-solver specialising in Delphi and C++Builder work. Visit his website at www.blong.com or email him on brian@blong.com

TDMWeb
Your website in safe hands www.TDMWeb.com