The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project? Just email Brian Long, our Delphi Clinic Editor, on clinic@blong.com

Transfer Efficiency

Q I'm making DLLs that are uploaded to a web server for a web app to use. Each time I make a change I have a big file to upload. How can I make the DLL smaller?

A When compiling any DLL or EXE in Delphi, we all normally fall foul of one of the primary benefits of the Delphi development system. This might be a bit like teaching your grandmother to suck eggs (although I've never tried this, so am going on rumours), but Delphi makes Windows development easy by providing the rich and well-formed VCL. The VCL is a class hierarchy which makes use of a lot of inheritance and polymorphism.

Polymorphism is all about deciding which object methods to execute at runtime, based on which object is actually being manipulated. It allows code to be written generically, without having to worry all the time about the specific class types of objects. Treating all objects as if they are of a common, generic class type, rather than their actual class means you do not have to cater for all the special cases that come along. The compiler makes sure that the method from the right class is executed at runtime, even though at compile-time, the compiler will not know which class this will ultimately be.

To allow polymorphism to work, the compiler must compile in all the appropriate polymorphic methods into the executable file. When a number of objects are treated as a generic class type, the polymorphic methods of all the classes inherited from that base class must be present in the executable in order to guarantee that polymorphism will still be able to work under all possible outcomes.

It is this requirement that causes Delphi executables to swell. Delphi has a *smart linker* which can strip out any routines that are *known* not to be called (procedures, functions and non-polymorphic methods), but polymorphism defeats the smart linker, leading to these big files. A do-nothing Delphi 5 application compiles to 286Kb with my default compiler settings.

One way to combat these large files, particularly in an environment where you need to send many updated versions of the files, or have many applications installed on a given machine, is to use *runtime packages*.

Packages were introduced in Delphi 3 and are specialised DLLs that contain code that can be transparently used in multiple applications. Borland supply a number of standard packages that contain the whole VCL and RTL. These can be uploaded onto your web server and can be used from that point on.

Because they must fulfil any application's requirements, the packages have *all* the routines, methods (both polymorphic and not) and variables compiled into them, and so seem quite large in themselves (for example, the main Delphi 5 VCL package, VCL50.BPL, is over 1.9Mb in size). However, if you compile with the option turned on to make use of runtime packages (`Project | Options... | Packages | Build with run-time packages`), your executables and DLLs will shrink markedly.

It should be noted that, due to the way packages work, you are generally advised to use either DLLs only, or packages only. In other words, rebuild your web application to use runtime packages (which will make the executable smaller). Then look into putting the code currently in your DLL into a custom package of your own. Make sure the web application is told about this new runtime package (again, the `Packages` page of the Project Options dialog).

Details of how to build and make use of custom packages is outside the scope of this column (well, to be truthful, it would fill it up and not leave room for any other subjects). Anyway, I'm sure it must have been covered in a past issue (time to make use of your *Collection 2000* CD) so I won't cover it here.

After all this, your web app will be small, and the custom package (which used to be a DLL) will also be very small and much easier to upload.

Transient Stay-On-Top

Q Thanks for your *Drag And Dock* article in Issue 63. It opened up a whole lot of new ideas for me. However, I have noticed that, whilst undocked controls reside in floating windows that stay on top of other forms in the same application, they do not stay on top of windows in other applications.

I desperately want this to be the case and have tried several things to no avail. I gave the floating window (the `TCustomDockForm`) a `FormStyle` of `fsStayOnTop`. I also tried passing its window handle to `SetWindowPos` with a flag of `HWND_TOPMOST` and to `SetWindowLong` with `GWL_EXSTYLE` and `WS_EX_PALETTEWINDOW` flags which should request the same thing. I didn't get anywhere.

A I had a quick look in the VCL, and the `fsStayOnTop` flag is implemented through a call to:

```
SetWindowPos(Handle,
  HWND_TOPMOST, 0, 0, 0, 0,
  SWP_NOMOVE or SWP_NOSIZE or
  SWP_NOACTIVATE);
```

so calling the API directly won't help. In any case, the TCustomDockForm constructor sets its FormStyle to fsStayOnTop, so setting it again will do nothing more.

The reason for the problem is that when the application is deactivated (which means another application is activated and given input focus), the Application object takes steps, which are reversed when the application is re-activated.

Application knows when the program is being activated or deactivated as it is sent a WM_ACTIVATEAPP message with a True or False parameter respectively.

If the application is being deactivated, it turns all stay-on-top windows into normal windows using its NormalizeTopMosts method. It then sends itself a CM_DEACTIVATE message, which ends up triggering the OnDeactivate event handler, if present.

If the app is being activated, it restores all previously stay-on-top windows back to stay-on-top windows with its RestoreTopMosts method. It then sends itself a CM_ACTIVATE message, which ends up triggering the OnActivate event handler, if present.

So the fact that stay-on-top windows are only stay-on-top whilst your application is active is actually by design. The reason, in case you are interested, is that it is not uncommon for a Delphi application to invoke external dialogs (such as the Windows common dialogs using the components on the Dialogs page of the Component Palette).

If stay-on-top windows were left as stay on top when an external dialog was invoked, the dialog may get irritatingly obscured by them.

To overcome the problem, programmatically set up an OnDeactivate event handler for the Application object and reverse the situation by calling Application.RestoreTopMosts from it (see Listing 1).

If you are using Delphi 5 or later, you will find it easier to drop a TApplicationEvents component on the form (from the Additional page of the Component Palette) and use the Object Inspector to make an OnDeactivate event handler from where you can call the Application method.

Listing 1: Overcoming the stay-on-top problem.

```
type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
  public
    procedure ApplicationOnDeactivateHandler(Sender: TObject);
  end;
...
procedure TForm1.FormCreate(Sender: TObject);
begin
  //Manually set up event handler for Application event
  Application.OnDeactivate := ApplicationOnDeactivateHandler
end;
procedure TForm1.ApplicationOnDeactivateHandler(Sender: TObject);
begin
  //When app is deactivated, fix stay-on-top windows
  Application.RestoreTopMosts
end;
```

Drag And Drop And Scrolling

Q When dragging from a listview to a treeview, how do you get the treeview to scroll like Windows Explorer does?

A The answer is that you have to do it manually, by checking where the mouse is and scrolling as appropriate. Let's look at a solution that will work just as well with memos, rich edits and other scrollable controls as it will with treeviews.

The goal is that when the user is dragging from some control over your treeview (or whatever it is), if the mouse position is somewhere near the edge of the control, the control starts scrolling in the appropriate direction. This means

the code to perform the scrolling task must be placed in the treeview's OnDragOver event handler.

Firstly, let's set up a simple drag and drop application (a finished version can be found on the disk as DragScroll.dpr). The question mentioned dragging from a listview to a treeview, but to keep things straightforward, our application will allow dragging from a label. Place a label component on the form and set the DragMode property to dmAutomatic.

Now place a treeview component on the form and use the Items property editor to give it lots of nodes. In the form's OnCreate event handler, call the treeview's FullExpand method to expand all the nodes in the tree.

Next, place a memo component on the form, set the ScrollBars property to ssBoth and enter lots of text into the Lines property of the memo. Now make an event handler for the treeview's OnDragOver event handler and make the memo's OnDragOver event share the event handler. If you want, you can rename the event handler in the Object Inspector to make it look more generic.

Finally, drop a timer component on the form, with its Enabled property set to False. The timer will perform the actual scroll operations when needed.

OnDragOver has a State parameter that specifies whether the mouse is entering the control's screen area, moving over the control or leaving the control's area. Each time the mouse is moved around the control, the event handler must check where it is.

If the mouse is close to one of the edges of the control (or to two edges, if the mouse is near a corner), then this information must be recorded. Another way of saying this is that it must be noted if the mouse is in a small *inset region* inside the border of the control, shown in light blue in Figure 1. Assuming the mouse is over the target control's inset region, periodically a scroll operation must be invoked in that direction. That's the short version. Now let's look at the details.

Windows defines some constants for the default values that are used in this type of operation. The default width of the inset region is 11 pixels, as defined by the DD_DEFSCROLLINSET constant in the ActiveX unit. The default delay before scrolling starts is 50ms (DD_DEFSCROLLDELAY) and the default gap between each successive scroll, once scrolling has started, is also 50ms (DD_DEFSCROLLINTERVAL).

These defaults can apparently be updated by modifying the DragScrollInset, DragScrollDelay and DragScrollInterval entries in the [windows] section of WIN.INI, according to *Inside OLE* by Kraig Brockschmidt, but I have a suspicion this might be a 16-bit Windows thing as I can find no other references to them.

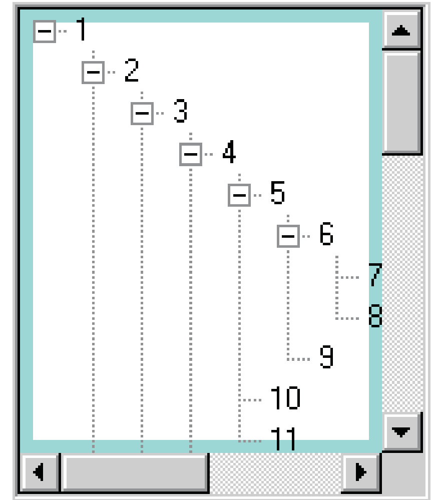

Figure 1: The scrolling inset region of a control.

Listing 2: Some useful types.

```
type
  TScrollDir =
    (sdUp, sdLeft, sdDown, sdRight);
  TScrollDirs = set of TScrollDir;
```

Listing 3: The shared OnDragOver event handler.

```
// Data fields defined in the form
ScrollDirs: TScrollDirs;
Ctrl: TWinControl;
...
procedure TForm1.SharedDragOver(Sender, Source: TObject; X,
  Y: Integer; State: TDragState; var Accept: Boolean);
var
  // Control's windows style and extended window style
  Style, ExStyle, HorzSclHt, VertSclWd, //Scroll bar sizes
  Left, Right, Top, Bottom: Integer;
begin
  Ctrl := Sender as TWinControl;
  case State of
    dsDragEnter, dsDragLeave: Timer1.Enabled := False;
    dsDragMove:
    begin
      // Get window styles to see if there are scroll
      // bars/borders
      Style := GetWindowLong(Ctrl.Handle, GWL_STYLE);
      ExStyle := GetWindowLong(Ctrl.Handle, GWL_EXSTYLE);
      // Record scroll bar size, taking into account they
      // might not be there
      HorzSclHt := 0;
      VertSclWd := 0;
      if Style and WS_HSCROLL <> 0 then
        HorzSclHt := GetSystemMetrics(SM_CYHSCROLL);
      if Style and WS_VSCROLL <> 0 then
        VertSclWd := GetSystemMetrics(SM_CXVSCROLL);
      // Record bounding dimensions of control's area,
      // taking into account borders and scroll bars
      Left := 0;
      Top := 0;
      Right := Ctrl.Width - 1 - VertSclWd;
      Bottom := Ctrl.Height - 1 - HorzSclHt;
      if (Style and WS_BORDER <> 0) or (ExStyle and
        WS_EX_CLIENTEDGE <> 0) then begin
        Left := GetSystemMetrics(SM_CXEDGE);
        Top := GetSystemMetrics(SM_CYEDGE);
        Dec(Right, Left);
        Dec(Bottom, Top);
      end;
      // Check if over a scroll bar, in which case reject
      // drop
      if ((X >= Right) and (X <= Right + VertSclWd)) or
        ((Y >= Bottom) and (Y <= Bottom + HorzSclHt))
        then begin
        Accept := False;
        Exit;
      end;
      //Initialise to no scrolling direction
      ScrollDirs := [];
      //See if in scroll region
      if (X >= Left) and (X < Left + DD_DEFSCROLLINSET) then
        ScrollDirs := [sdLeft];
      if (X >= Right - DD_DEFSCROLLINSET) and
        (X < Right) then
        ScrollDirs := [sdRight];
      if (Y >= Top) and (Y < Top + DD_DEFSCROLLINSET) then
        ScrollDirs := ScrollDirs + [sdUp];
      if (Y >= Bottom - DD_DEFSCROLLINSET) and
        (Y < Bottom) then
        ScrollDirs := ScrollDirs + [sdDown];
      // If so, reset timer tick and record which region
      if ScrollDirs <> [] then begin
        Timer1.Interval := DD_DEFSCROLLDELAY;
        Timer1.Enabled := True;
      end
    end;
  end
end;
```

Some simple comparisons of the co-ordinates passed in the X and Y parameters of the OnDragDrop event handler will tell us if the mouse is in the inset region, and if so, whereabouts. To record which part it is in, a couple of type definitions are useful. An enumerated type defines the four primary scroll directions, but since the mouse might be, for example, at the top-left of the control, warranting an upwards and leftwards scroll, a set type is also defined to help store multiple values.

The main code is in the shared OnDragOver event handler (see Listing 3). It starts off trying to work out the bounding area of the control that is appropriate for the mouse to be over in the first place. This must not include any borders (introduced with the BorderStyle property) or scroll bars that may be present. The GetSystemMetrics and GetWindowLong APIs are useful tools in these calculations.

Once this area is known, the position of the mouse can then be examined. If it is in the inset region, appropriate values are added into a set variable. If the mouse is over a scroll bar, the drag operation is rejected, causing the mouse cursor to turn into a *No Entry* sign.

Once the comparisons have been made, the set variable will be empty if the mouse is not in the inset region. If, however, the set is not empty, then a timer component has its Interval property set to DD_DEFSCROLLDELAY and is enabled. Admittedly, the value of 50ms is smaller than the granularity of Windows timers (which is 55ms at best), but we are fairly close.

The timer's code can be seen in Listing 4. Depending which values are found in the set, the control is scrolled in that direction by one unit. The timer's interval is then set to the other time constant, DD_DEFSCROLLINTERVAL. The timer is eventually disabled when the mouse is moved off the control or the drag operation is cancelled in some way (the OnDragOver event handler's State parameter will be dsDragLeave).

For more information on VCL drag and drop see my articles on the subject in Issues 56 and 57. For details on inter-application drag and drop, as implemented by a variety of Windows applications, such as Windows Explorer, see my article in Issue 58.

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  //Depending which region, scroll as appropriate
  if sdLeft in ScrollDirs then
    Ctrl.Perform (WM_HSCROLL, SB_LINELEFT, 0);
  if sdRight in ScrollDirs then
    Ctrl.Perform (WM_HSCROLL, SB_LINERIGHT, 0);
  if sdUp in ScrollDirs then
    Ctrl.Perform (WM_VSCROLL, SB_LINEUP, 0);
  if sdDown in ScrollDirs then
    Ctrl.Perform (WM_VSCROLL, SB_LINEDOWN, 0);
  Timer1.Interval := DD_DEFSCROLLINTERVAL
end;
```

➤ *Listing 4: A timer does the scrolling.*