The Delphi CLINIC

Edited by Brian Long

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

Enhanced Grids

Q I've noticed that both TStringGrids and TDBGrids have vertical scrollbars when necessary. When you drag the scrollbars to a new position, they update the underlying grid as you release the mouse. This matches the behaviour of Microsoft Word. However, I also notice that the scrollbar on the Delphi code editor dynamically updates the editor as you are dragging the scrollbar. Is it possible to make the VCL grid components act in a similar, more dynamic, fashion?

A To answer this requires some knowledge of what happens when you manipulate a scrollbar. A scrollbar is either set up as an independent entity (eg a TScrollBar component), or as part of another control (as in the case of the VCL grids). Either way, when they are clicked on, or clicked and dragged, they send messages to an underlying window that picks them up and acts upon them.

If the scrollbar is set to be vertical, it sends a wm_VScroll message to its underlying window when it is clicked, and if horizontal it sends a wm_HScroll message. Depending on where the scrollbar is clicked, it sends different accompanying information with the scroll message in the low word of the WParam parameter.

To get some stable terminology for the various sections of the scrollbar, I will refer to Delphi 4's Win32 API help file. (See the boxout on page 62 for more details about Win32 API help and how to get it.) According to the help: 'a scrollbar consists of a shaded shaft with an arrow button at each end and a scroll box (sometimes called a thumb) between the arrow buttons.'

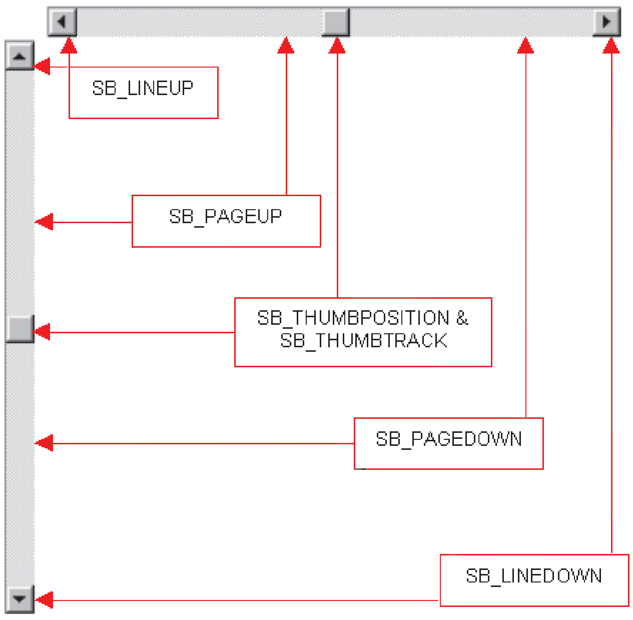
It then goes on to say: 'When the user clicks an arrow button, the application scrolls the content by one unit (typically a single line or column). When the user clicks the shaded areas, the application scrolls the content by one window. The amount of scrolling that occurs when the user drags the scroll box depends on the distance the user drags the scroll box and on the scrolling range of the scrollbar.'

So, as Figure 1 illustrates, when the left arrow button of a horizontal scrollbar, or the top arrow button of a vertical scrollbar is pushed, the WParam value is set as sb_LineUp. The right arrow button and bottom arrow buttons give sb_LineDown. The shaft area to the left, or above the thumb, generates sb_PageUp and the area to the right, or below the thumb, gives sb_PageDown. When you click on the thumb and drag it around, it generates appropriate messages as you do so with sb_ThumbTrack as the associated value. When you release the thumb, sb_ThumbPosition is sent. In addition to all these, when any scroll operation (a click on an arrow button, a click on the shaft area or a drag of the thumb) is completed, another message is sent along with a WParam of sb_EndScroll.

If you have scrollbars that can take the focus, such as TScrollBar components (a scrollbar flashes its thumb when focused), then pressing Home and End take you to the left/top or right/bottom of the scrollbar and generate sb_Top and sb_Bottom WParam values respectively.

Incidentally, TScrollBar components translate both wm_HScroll and wm_VScroll messages into OnScroll events, and to indicate what scroll operation took place, the event handler is passed a value from the TScrollCode type as the ScrollCode parameter.

With this information, and what the questioner described as the current behaviour of the VCL grid controls, it should hopefully be



➤ *Figure 1: WParam values for WM_HSCROLL and WM_VSCROLL messages.*

clear that these grid components are programmed to react to sb_ThumbPosition, but do not act upon sb_ThumbTrack. In the case of DBGrid components, this is probably intentional, as the requirement to redraw as you drag the thumb through a set of records brought back from an SQL expression could be time consuming and expensive in terms of network traffic.

So what we need to do to solve the problem is to detect any sb_ThumbTrack messages, indicating that the thumb is being dragged, and send an sb_ThumbPosition message straight to the underlying control, to get the same effect as if we let go of the thumb. In other words, as the thumb track messages come up, we entice the grid to redraw if necessary, by giving it faked thumb position messages.

Listing 1 shows the key parts of a new grid that performs this operation. This code has been added into two new components, TScrollingStringGrid (ScrollingStringGrid.Pas) and TScrollingDBGrid (ScrollingDBGrid.Pas). You can see that the TWMHScroll and TWMVScroll message cracker records give you the low word of WParam as a field called ScrollCode.

If you install these two components, you should find that their horizontal and vertical scrollbars now perform dynamic updating of the grid as the thumb is dragged. To check it out, after installing these two components, you can run the GridTest.Dpr sample project on the disk.

Wm_HScroll/wm_VScroll WParam value	OnScroll ScrollCode parameter value
sb_Bottom	scBottom
sb_EndScroll	scEndScroll
sb_LineDown	scLineDown
sb_LineUp	scLineUp
sb_PageDown	scPageDown
sb_PageUp	scPageUp
sb_ThumbPosition	scPosition
sb_ThumbTrack	scTrack
sb_Top	scTop

➤ *Table 1: Scroll message constants and corresponding VCL enumerated type values.*

➤ *Listing 1*

```
type
  TScrollingStringGrid = class(TStringGrid)
  public
    procedure WMHScroll(var Msg: TWMHScroll);
      message wm_HScroll;
    procedure WMVScroll(var Msg: TWMVScroll);
      message wm_VScroll;
  end;

procedure TScrollingStringGrid.WMHScroll(var Msg: TWMHScroll);
begin
  inherited;
  if Msg.ScrollCode = sb_ThumbTrack then
    Perform(wm_HScroll, MakeLong(sb_ThumbPosition, Msg.Pos), Msg.ScrollBar)
end;

procedure TScrollingStringGrid.WMVScroll(var Msg: TWMVScroll);
begin
  inherited;
  if Msg.ScrollCode = sb_ThumbTrack then
    Perform(wm_VScroll, MakeLong(sb_ThumbPosition, Msg.Pos), Msg.ScrollBar)
end;
```

Incidentally, if you are using DBGrids in Delphi 1 or 2, or using DBGrids against either SQL data or filtered local data in any version, you might be concerned about the behaviour of the vertical scrollbar. The thumb will only ever take up three positions: top middle or bottom.

When the DBGrid was originally designed, it was a generic device for displaying a data set. Whilst Paradox and dBase files have record numbers (which were surfaced in Delphi 2 by the RecNo property), SQL data sets and filtered data sets do not. The BDE has no idea how many records will be in any arbitrary SQL or filtered result set. So with this information in hand, the DBGrid was designed to allow scrolling, but had to cater for the lack of record numbers. Without a record number, you have no idea how far through the result set you are, and so the thumb of the scrollbar cannot be positioned proportionately. As a consequence, the DBGrid has its thumb at the top when the BOF property is True, at the bottom when EOF is True, and right in the centre under all other circumstances.

The DBGrid was improved in Delphi 3 to cater for the existence of a record number in some circumstances, and to display a proportional scrollbar when it is available. But when it is not available, the grid's scrollbar goes back to its old tri-state behaviour.

Restarting Windows

Q My application sometimes makes changes to the Windows registry. In these cases I need to get Windows restarted for the changes to have any effect. Whilst I can reboot using ExitWindowsEx, I want to make my application look consistent with Windows itself. The Control Panel applets all invoke some standard looking dialog to offer the user the choice of rebooting. How do I call this dialog in my application?

A My! Another Windows termination question. You might like to also refer to *Forcing Windows Shutdown* in Issue 36's *Clinic*, as well as *Stopping Windows Shutdown* and *Program Running Upon Windows Start-Up* in Issue 38.

As well as consistency with Windows, the Windows 95 version of this reboot dialog is rather pleasant, in that if you hold the `Shift` key down whilst pressing the `Yes` button (to accept the Windows restart), it will not reboot the machine. Instead, Windows just shuts down and restarts without a reboot. This means the time taken to restart Windows is much reduced.

In researching this question I referred to Dr GUI from the Microsoft Developer Network. In *Microsoft Developer Network News* from March/April 1997, Dr GUI was asked a similar question. The eminent Doctor describes `ExitWindowsEx`, but says this does not help perform a bootless restart. The conclusion reached is that one has to write a thunk to call the 16-bit API routine `ExitWindows` which does support the shorthand restart behaviour.

Now whilst it is true that such a thunk would do the job on Windows 95, our desire here is to gain access to the dialog that Windows already has somewhere, and thereby have all this done for us.

The API that spawns the reboot dialog (as well as the ones that launch the run application dialog, the find file dialog, the choose icon dialog etc) comes from an undocumented portion of SHELL32.DLL. To make things tricky for programmers, the APIs in question are exported from this DLL without any name information, they are exported by ordinal only.

If we can find the relevant information, this poses little problem for Delphi developers. For information about the undocumented portions of SHELL32.DLL, I recommend a trip to www.geocities.com/SiliconValley/4942. Here you will find James Holderness's Undocumented Windows 95 website. This gives pretty full information (using the C syntax) about these APIs and typically highlights differences to be found between Windows 95 and Windows NT implementations. Choosing James's Function Index link will allow you to get to the documentation for the API `RestartDlg`. A Delphi declaration looks like Listing 2.

Windows API Help Files

The Windows API is the set of subroutines, and their associated required types and constants, that Windows itself defines for the programmer to use in order to write Windows applications. All Windows programs use the Windows API in order to work.Tools like Delphi have been developed to shield developers from the complexity inherent in the Windows API. However, when the nice components and support routines in Delphi fail to meet a certain programming task, you can always fall back on direct calls to the Windows API.

Microsoft develops Windows, and hence writes the Windows API. Microsoft also documents the Windows API. Unfortunately, since the API was developed originally in C, and is targeted at C and C++ programmers, Microsoft's documentation is written with C syntax.

Delphi 1 came with a Delphi-ised version of the Win16 API help file. This took the help authors quite a long time to translate and, in cases, there are errors. In Delphi 1, the Windows API information in the help file is hooked into the context-sensitive help system. Place your cursor on an API function name in the editor, press `F1` and the help comes up. However you cannot find all the keywords from the API help file by looking in Delphi's help index.

You can choose `Help | Windows API` to jump to the file directly. Additionally, you are also able to add a Windows API speedbutton onto your speedbar. Right click the speedbar, choose `Configure...`, select the `Windows API help` command from the `Help` category and drag it onto some empty area on the speedbar.

Unfortunately, as Delphi 2 was being developed, it became apparent that translating the Win32 help file was too mammoth a task. The problem was not just the increased size of the file, but also the fact that the Windows API seemed to be ever expanding, with Microsoft updating the original help file on a regular basis.

So Delphi 2 (as well as versions 3 and 4) comes with the original C programmer's Win32 API help file, as licensed to Inprise from Microsoft. They all support context-sensitive help for API routines in the editor, but have hidden the `Help | Windows API` menu item. Fortunately, despite hiding it, you can still place the speedbutton for it on the speedbar. As an alternative you can navigate to Delphi's HELP directory and locate Win32.Hlp from there.

Delphi 4 makes the help file marginally more accessible by adding it into the `Start` menu `Delphi` folder. You can access this by clicking on your task bar's `Start` button, choosing `Programs`, `Borland Delphi 4`, `Help`, `MS SDK Help Files`, `Win32 Programmer's Reference`.

Having found the API help file, it may not be particularly useful unless you know how to speak the C language. But examination of it, and appropriate sections of the relevant import unit, can be educational. The source for these import units can be found in Delphi's SOURCE\RTL\WIN directory in the Professional and Client/Server versions. The interface sections of these units can be found as .INT files in the DOC directory in the Standard version.

From examining these, you can learn how Inprise have represented various C parameters, parameter types and data structures in Pascal. The primary import unit is `Windows` in 32-bit and the `WinProcs` and `WinTypes` pair in Delphi 1.

You may also wish to investigate the work being done by the Jedi group. They are attempting to translate all the additional Windows APIs into Delphi Pascal that Inprise have not yet done themselves. You can find the Jedi group at www.delphi-jedi.org.

Because the Win32 SDK is ever growing, and because Inprise only have access to reasonably old versions of the API documentation, Delphi 4's README suggests referring to the Microsoft Developer Network (MSDN) website http://msdn.microsoft.com/developer. This will always have the most up-to-date SDK documentation available.

```
function RestartDialog(hwndOwner: HWnd; lpstrReason: PChar; uFlags: UINT):
  Integer; stdcall; external 'Shell32.Dll' index 59;
```

➤ *Listing 2*

```
function FindWindowA(lpClassName, lpWindowName: PAnsiChar): HWND;
  stdcall; external 'user32.dll' name 'FindWindowA';
function FindWindowW(lpClassName, lpWindowName: PWideChar): HWND;
  stdcall; external 'user32.dll' name 'FindWindowW';
```

➤ *Listing 3*

The first parameter is the window that owns the dialog. The second parameter is an optional string (we will come back to this). The last parameter can take the same values supported by both `ExitWindowsEx` and `ExitWindows`, including `ew_RestartWindows`. In fact it is the `ew_RestartWindows` flag that is used to get the `Shift` key to do its thing. The function returns `mrYes` or `mrNo`, dependent on the button pressed. If `mrYes` is returned, Windows will attempt to restart (but may not be successful due to another application objecting).

The dialog always has a caption of `System Settings Change`. Assuming the `lpstrReason` parameter is `nil`, the dialog displays a default message. If `uFlags` is `ewx_ShutDown` the text is ‘Do you want to shut down now?’ All other values use the message ‘You must restart your computer before the new settings will take effect. Do you want to restart your computer now?’

If `lpstrReason` has a textual value then it is written as the first piece of text in the dialog, followed immediately by whatever text would be written anyway. Because it is immediately followed by the other text, you should finish the parameter with a space, or a carriage return character. If `lpstrReason` starts with a # character, the normal text is not written in the dialog box.

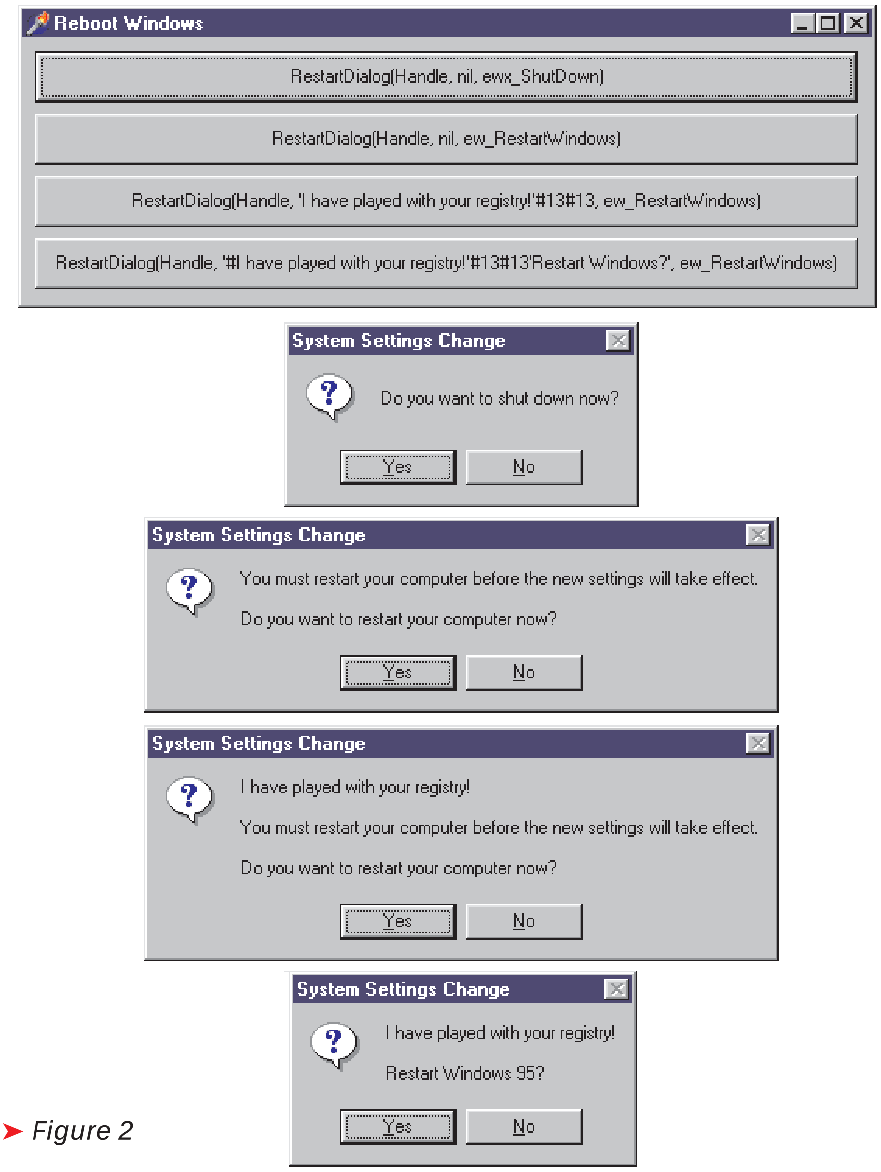
In the documented portion of the Win32 API, any routines that take textual versions have two implementations. Take `FindWindow` for example. There is not really any such routine as `FindWindow`. Instead, Windows implements the routines `FindWindowA` and `FindWindowW` (see Listing 3).

`FindWindowA` takes ANSI strings (`PAnsiChars` actually) and `FindWindowW` is defined to take Unicode strings (wide strings, or `PWideChars` in C lingo). This allows developers to write applications using normal ANSI strings or Unicode applications. The Pascal function name `FindWindow` is defined in the `Windows` import unit to map onto `FindWindowA`:

```
function FindWindow(
  lpClassName,
  lpWindowName: PChar):
  HWND; stdcall;
  external ‘user32.dll’
  name ‘FindWindowA’;
```

`RestartDialog` is not documented and so is slightly different. On Windows 95 it always takes ANSI strings and on Windows NT it always takes Unicode strings. As a consequence, when passing textual information to this API you need to check which platform you are running on to ensure that you pass the right type of characters along.

Figure 2 shows a form with 4 buttons, each of which executes a line of code similar to that written



```
RestartDialog(Handle, nil, ewx_ShutDown)
RestartDialog(Handle, nil, ew_RestartWindows)
RestartDialog(Handle, 'I have played with your registry!'#13#13, ew_RestartWindows)
RestartDialog(Handle, '#I have played with your registry!'#13#13'Restart Windows?', ew_RestartWindows)
```

System Settings Change

Do you want to shut down now?

Yes No

System Settings Change

You must restart your computer before the new settings will take effect.

Do you want to restart your computer now?

Yes No

System Settings Change

I have played with your registry!

You must restart your computer before the new settings will take effect.

Do you want to restart your computer now?

Yes No

System Settings Change

I have played with your registry!

Restart Windows 95?

Yes No

➤ *Figure 2*

in its caption. Below the form are the four dialogs generated. The last two buttons do send text to `RestartDialog`, as you can see, and so in fact the implementations of their button handlers are not quite as straightforward as they could be. Listing 4 shows how to write platform dependent code for these buttons. You can see that two import declarations for `RestartDialog` have been written, one for use in Windows 95, one for use in Windows NT. If no text is being passed to the API, it is irrelevant which one you choose.

As you can see, when you define a Pascal string constant it doesn't matter what string data type a subroutine call is expecting, your constant can be passed along quite happily. In fact, the same string constant can be passed along to several routines, as a `String`, `ShortString`, `WideString`, `PChar`, `PAnsiChar` and `PWideChar`, and the compiler deals with the requirements sensibly, translating the string when needed.

```
function RestartDialog(hwndOwner: HWnd; lpstrReason: PChar; uFlags: UINT):
  Integer; stdcall; external 'Shell32.Dll' index 59;
function RestartDialogW(hwndOwner: HWnd; lpstrReason: PWideChar; uFlags: UINT):
  Integer; stdcall; external 'Shell32.Dll' index 59;
procedure TForm1.Button3Click(Sender: TObject);
const
  Msg = 'I have played with your registry!'#13#13;
begin
  if Win32Platform = VER_PLATFORM_WIN32_WINDOWS then
    RestartDialog(Handle, Msg, ew_RestartWindows)
  else
    RestartDialogW(Handle, Msg, ew_RestartWindows)
end;
procedure TForm1.Button4Click(Sender: TObject);
begin
  if Win32Platform = VER_PLATFORM_WIN32_WINDOWS then
    RestartDialog(Handle, '#I have played with your registry!'#13#13 +
      'Restart Windows 95?', ew_RestartWindows)
  else
    RestartDialogW(Handle, '#I have played with your registry!'#13#13 +
      'Restart Windows NT?', ew_RestartWindows)
end;
```

➤ *Listing 4*

TStringGrid Coordinates

Q Do you know how I can find out what cell on a string grid I am selecting on a single mouse click?

A The `TStringGrid` class implements a convenient `MouseToCell` method. This takes the mouse coordinates and translates them into grid coordinates. So if you have a string grid called `Grid`, then you could use an `OnClick` event handler or `OnMouseDown` handler as shown in Listing 5.

```
procedure TForm1.GridClick(Sender: TObject);
var
  Pt: TPoint;
  Col, Row: Longint;
begin
  GetCursorPos(Pt);
  Grid.MouseToCell(Pt.X, Pt.Y, Col, Row);
  Caption := Format('Cell clicked was (%d, %d)', [Col, Row])
end;
procedure TForm1.GridMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  Col, Row: Longint;
begin
  Grid.MouseToCell(X, Y, Col, Row);
  Caption := Format('Cell clicked was (%d, %d)', [Col, Row])
end;
```

➤ *Listing 5*

Code Insight Customisation

Q A Delphi 3 or 4 question. Is there any way of forcing an instance's methods/properties to be sorted alphabetically in the pop-up code completion listbox?

A Right click on the kibitz window (as the Code Completion window is called internally), which can be pulled up on demand with `Ctrl+Space`, and choose `Sort By Name` instead of `Sort By Scope`.