Beating The System: When Worlds Collide

Delphi 5 versus common controls...

by Dave Jewell

For a few months now, I've been planning to do an article on some of the interesting new capabilities in recent versions of Microsoft's Common Controls DLL, COMCTL32.DLL. For example, ever since version 4.70, it's been possible to incorporate custom drawing capabilities into tree views, header controls and so forth. Borland seemed rather slow off the mark at exploiting these new features within the various VCL common control 'wrappers', so I figured that I'd steal a march on them and do the job myself. Alas, too late! Those bounders from Scotts Valley have beaten me to it by adding custom draw support to the Delphi 5 versions of `TTreeView`, `TToolbar` and `TListView`. I figured there was nothing for it but to swallow my pride and describe the new goodies that the latest version of Delphi brings to the party.

Well, that was the theory, but as I worked on this article, I found myself getting increasingly frustrated with the way in which the custom draw capabilities have been implemented. This is a problem which, primarily, relates to the way in which Microsoft have implemented the underlying custom draw architecture. However, let's get positive for a while! I'll save the bad news until later...

Heading Up The Pack With THeaderControl

You don't need much familiarity with Windows Explorer before you notice that, in the Details view, it's possible to grab hold of one of the column headings in the header control and then drag it to a new position, thus rearranging the order in which columns are displayed (see Figure 1). As you drag a column header, a grey outline of the header follows the mouse around, and a thick blue line appears between adjacent column headers to indicate where the 'floating' header would be inserted if the mouse button was released at that point.

Before I became aware of all the new features in COMCTL32.DLL, I reckoned that Microsoft were using a custom version of the header control in Explorer, but this isn't the case, it's just a matter of enabling the capability in the standard control. In Delphi 5, the `THeaderControl` control has a new property, `DragReorder`, which takes care of this functionality. Assuming that you set `DragReorder` to `True`, a new event, `OnSectionDrag`, then comes into play. This event is fired only when the mouse button is released, and has a function prototype like this:

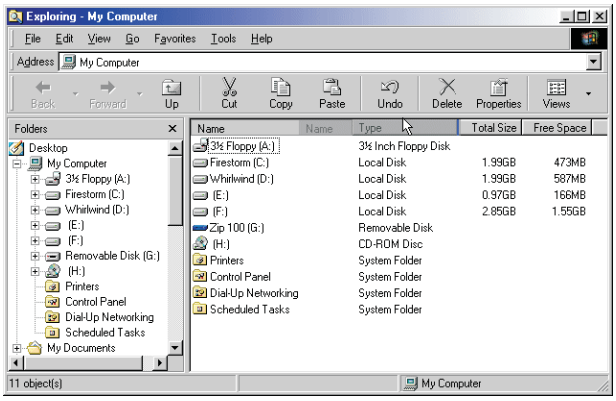
```
type TSectionDragEvent =
  procedure (Sender: TObject;
  Section: THeaderSection;
  var AllowDrag: Boolean)
  of object;
```

The `AllowDrag` argument can be used to specify whether or not the drag is allowed. Because this event handler also receives a reference to the header section which is being dragged (the `Section` argument) you can use it to enable or disable dragging on a per-column basis. This would be particularly appropriate if you wanted, for example, the left-most column to remain where it was, only allowing the user to reconfigure the other columns.

Although this is all good stuff, I would like to have seen another event which enables you to track a section drag *during* the drag operation rather than merely firing the event at the end. If this event handler were supplied with the appropriate arguments, it would be possible to replace the thick blue 'it'll-go-here' indicator with your own custom highlighter, such as those cute lime green arrows used by Developer Express in their `TdxTreeList` control. (In recent versions of `TdxTreeList`, they don't have to be lime green any more, you can have shocking pink if you prefer!) Because there's no simple way of participating in a column drag, it isn't possible to make non-standard things happen during the drag. However, on the positive side, because `OnDragEvent` is fired after the drag operation, it's the perfect place for an application to retrieve the current order of columns at this point, writing the information to a .INI file or the registry so as to make the user's choice persistent.

As in previous versions of Delphi, it's possible to set the `Style` property of individual sections to `hsOwnerDraw`, whereupon it becomes possible to use the `OnDrawSection` event to take full responsibility for the drawing of a particular column header. However, this is overkill if all you want to do is draw a predefined image alongside the caption of each

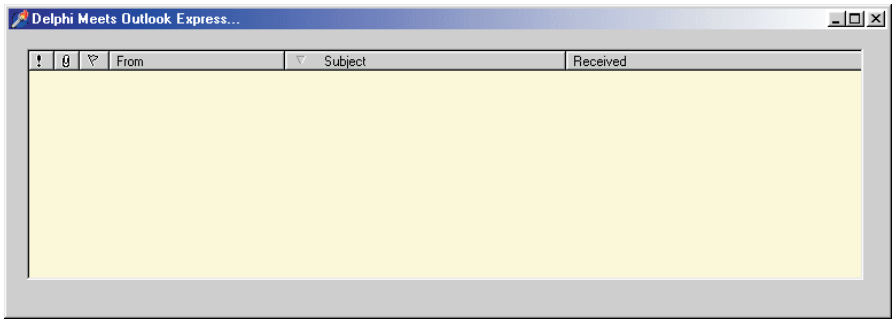
➤ *Figure 1: Windows Explorer is an example of a Microsoft program that allows you to reorder columns simply by dragging from one place to another. The new DragReorder property in THeaderControl makes it easy to do the same.*



column header. Accordingly, the Delphi 5 implementation of `THeaderControl` introduces a new property, `Images`, which you point to an existing `TImageList` component on the same form. Once this is done, you can set the individual `ImageIndex` properties of each section so as to reference the desired image within the image list. You should use this feature with some care, because if you use overlarge bitmaps, you could end up making your header control look much like an Internet Explorer style toolbar, which would doubtless cause some confusion to users of your application!

Emulating Outlook Express...

A better approach is to use the image capability in a discreet, tasteful manner. For example, many modern programs, while displaying a detail view in a listview control, will re-sort the view on a particular column when that column header is clicked, the Windows Explorer being an example of such an application. Some programs, such as Outlook Express, will display a small down-pointing or up-pointing arrow to indicate which column is responsible for sort order, and whether it's an ascending or descending sort.



➤ *Figure 2: Outlook Express, or is it? Well, it isn't of course, but the new Image property makes it possible to associate a small glyph with each section header. With a little extra code (see Listing 1) you can even have 'toggling' sort markers.*

Using the new `Images` property in conjunction with `ImageIndex`, this is now very straightforward to achieve.

An example implementation is given in Listing 1 and you can see the corresponding demo program running in Figure 2. This demo is designed to emulate the look and feel of Outlook Express, at least as far as the behaviour of the `THeaderControl` is concerned! In fact, I shamelessly extracted the Outlook Express toolbar bitmaps from Microsoft's `MSOERES.DLL` file (the various resources used by the program) and stuffed them into a Delphi image-list! I then set up the various `ImageIndex` properties as described above in order to get the effect shown. You'll also notice that I set the `DragReorder` property to `True`, because Outlook Express allows full column reordering.

The real business end of the demo code is the `OnSectionClick` event handler which is called whenever a section header gets clicked. Firstly, it examines the `ImageIndex` and `Text` properties of the clicked section in order to determine which section actually got clicked on. The result is stored as a variable of type `TSortOrder` for subsequent passing to a routine which will do the actual sorting. Next, the code examines the clicked section to see if it is already displaying a sort marker. If so, then the marker is 'toggled' from ascending to descending sort, or vice versa. If there's no previous marker shown, then it defaults to an ascending sort which, again,

➤ *Listing 1*

```
unit OEDemoForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, ImgList, ComCtrls, ExtCtrls;
type
  TSortOrder = ( soPriority, soAttachment, soFlag, soFrom,
                 soSubject, soReceived );
  TForm1 = class(TForm)
    ToolbarImages: TImageList;
    Panel1: TPanel;
    Header: THeaderControl;
    ListView1: TListView;
    procedure HeaderSectionClick(HeaderControl:
      THeaderControl; Section: THeaderSection);
  private
    procedure SortMessages (Order: TSortOrder; Ascending:
      Boolean);
  public
  end;
var Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.HeaderSectionClick (HeaderControl:
  THeaderControl; Section: THeaderSection);
var
  Idx: Integer;
  Sec: THeaderSection;
  SortAscending: Boolean;
  SortOrder: TSortOrder;
begin
  // Just to shut the compiler up!
  SortOrder := soReceived; SortAscending := True;
  with Section do begin
    // Firstly, figure out which section this is.
    if ImageIndex = 4 then
      SortOrder := soPriority
    else if ImageIndex = 5 then
      SortOrder := soAttachment
    else if ImageIndex = 33 then
      SortOrder := soFlag
    else if Text = 'From' then
      SortOrder := soFrom
    else if Text = 'Subject' then
      SortOrder := soSubject
    else if Text = 'Received' then
      SortOrder := soReceived;
    // redisplay sort marker according to selected section
    if Text <> '' then begin
      if ImageIndex = -1 then
        SortAscending := True
      else
        SortAscending := ImageIndex = 2;
      ImageIndex := 2 + Ord(SortAscending);
    end;
    // Next, remove sort marker from any other columns
    for Idx := 0 to Header.Sections.Count - 1 do begin
      Sec := Header.Sections [Idx];
      if (Sec<>Section) and (Sec.ImageIndex in [2,3]) then
        Sec.ImageIndex := -1;
    end;
    // Finally, the sort !
    SortMessages (SortOrder, SortAscending);
  end;
end;
procedure TForm1.SortMessages (Order: TSortOrder;
  Ascending: Boolean);
begin
  // An exercise for the reader....
end;
end.
```

emulates the behaviour of Microsoft's code.

The next job is to remove the sort marker (if any) from other sections. Since we can only sort on one column at a time, we need to ensure that only one column header displays a sort marker. Finally, the `SortMessages` routine is called to perform the actual sort, the implementation of which is left as an exercise for the reader. Be fair, this article is about new common control features, not about how to write an email or newsgroup reader!

Incidentally, you might be thinking that I've used a rather roundabout way of determining which section header got clicked on. If you are familiar with the `THeaderControl`, you're probably wondering why I don't just look at the `ID` property of the `Section` argument passed to the `OnSectionClick` handler? That would certainly work in this simple case, irrespective of runtime reordering, but you should realise that in a more complex program like Outlook Express, the user has the option not only of rearranging columns, but also of deciding which columns are visible. It might be possible to still use the `ID` property in such circumstances, but you'd probably want another `TCollection` in which to store the 'off screen' items. I decided to adopt the approach used here because it also makes the code relatively independent of changes to the VCL wrapper layer.

Finally, you'll notice that the little up/down sorting markers don't appear in exactly the same positions as they do in Outlook Express. Outlook places the glyphs after the section text, which has the advantage that the text doesn't shuffle up and down when the glyph appears or disappears. If you want your header to look just like the Microsoft offering, and you don't want to centre or right-align the section text, then you'll need to use owner draw to draw the contents of each column header.

The complete code is included on this month's companion disk. You'll also find a 'packaged' EXE file which will only run if you've got the file VCL50.BPL. Since I've only got a pre-release version of Delphi 5 at the moment, there's a possibility that you may need to rebuild the EXE for compatibility with the shipping version of the VCL50 package.

Redmond Strikes Again!

If you take a look at the various events surfaced by the new Delphi 5 implementations of `TTreeView`, `TListView` and `TToolbar`, you might be forgiven for thinking that Borland have taken leave of their senses. The familiar `OnCustomDraw` event has now been joined by another called `OnAdvancedCustomDraw`. Perhaps in Delphi 6, we should look forward to the introduction of an `OnReallyReallyAdvancedCustomDraw` event?

Predictably, the real culprit here is Microsoft, and those API designers at Redmond who have such a talent for extending the API in ways that would make a grown man cry. In order to understand precisely why a new custom draw mechanism was added, it's instructive to review the development of custom draw support in Windows controls. So let's forget about Delphi and the VCL library for a minute and consider things from an API perspective.

The traditional custom-draw mechanism, (actually called owner draw) requires that you create a control using a specific style bit which tells Windows, in advance, that you want to make use of the owner draw facility. At the API level, controls are ultimately created through a call to `CreateWindow` or `CreateWindowEx`, both of which take a style parameter that's made up of a series of style bit flags. Thus, if you wanted to create an owner draw listbox, you'd specify the `LBS_OWNERDRAW` style at the time the listbox was created. Your application would then receive `WM_DRAWITEM` messages every time that Windows wants to draw a listbox item. You'd examine the `WM_DRAWITEM` message, make sure it's been sent by the listbox (it might have been sent by another owner draw control) and then draw whatever you wish onto the device context passed as part of the message.

This, incidentally, is the key reason why (as I've often stated) Visual Basic can't implement owner draw listboxes. The `lParam` argument received in the `WM_DRAWITEM` message is a pointer to an important data structure, and because VB doesn't understand pointers, everything collapses like a pack of cards. If you want an owner draw listbox in Visual Basic, you've really got to use a third party DLL or OCX to help with the meaty stuff.

The big problem with the traditional owner-draw approach is that it represents something of an all-or-nothing approach. If you want to draw the contents of a listbox item using the `LBS_OWNERDRAWITEM` mechanism, then you have to draw the *entire* item, including the background of the item rectangle, the item text, and of course any extra glyphs that you wish to display. Although this gives you maximum flexibility, it's inconvenient if you simply want to (for example) change the font used to display a selected item (but see my later comment on this!).

The newer custom draw mechanism uses a somewhat different approach. With custom draw, the control periodically sends `NM_CUSTOMDRAW` notification messages at crucial points during a drawing operation. Because these are notification messages, they are 'piggy-backed' onto a `WM_NOTIFY` message in the usual way. Custom draw is currently implemented (at the API level) for header controls, track-bars, rebars, tool-tips, as well as listviews, toolbars and tree view controls. However, Delphi 5 only surfaces the custom draw functionality for the last three named control types, so we won't concern ourselves with the others here.

In addition to the above, the custom draw mechanism introduces the concept of a 'paint cycle'. This means that notification mechanisms are potentially sent at up to four different times as follows: immediately before painting the control, immediately after

painting the control, immediately before erasing the control, and immediately after erasing the control.

If you think about it, this makes some sense: getting in on the act *before* painting allows you to select different fonts (for example) into the control's device context whereas getting in on the act after painting allows you to overlay additional graphics on top of whatever the control itself has drawn. Similarly, getting control before and after erasing allows you to replace the background of the control with some non-standard background, or else overlay additional information onto the control's background area. At least, that's the theory.

The above four stages are referred to in the Microsoft documentation as 'global draw stages', and apply to the control as a whole, rather than to any specific item. The Microsoft API documentation uses a modifier bit flag to introduce an additional set of draw stages that apply to items. In other words: immediately before painting an item, immediately after painting an item, immediately before erasing an item, and immediately after erasing an item.

In the Borland VCL wrappers, the item-specific drawing is split off into a separate event handler, as we shall see.

Into The Abyss

So let's take a look at what's been added to the Delphi 5 implementation of `TToolBar`. If you examine the `Events` page in the Property Inspector, you'll find the following new events lurking, none of which are present in the Delphi 4 wrapper:

```
OnCustomDraw
OnCustomDrawButton
OnAdvancedCustomDraw
OnAdvancedCustomDrawButton
```

The `OnCustomDraw` event is specifically for painting the background image of the toolbar. So, if you want to do something as simple as using a non-standard bitmap for the toolbar background, you can do it as easily as the code shown in Listing 2. This assumes that we've already got a `TBitmap` variable called `Clouds` and will produce something like the effect shown in Figure 3. In fact, you can reduce the code to a one-liner if you're sure that your background bitmap is always going to be larger than the toolbar rectangle, in which case you can dispense with the bitmap tiling code. You'll also notice that I haven't touched the value of the `DefaultDraw` argument. By default this is set to `True`, which is almost certainly what you want. The documentation is vague on this subject but if you set it to `False`, the buttons in your toolbar will disappear, which is almost certainly what you *don't* want!

The `OnCustomDrawButton` routine enables you to control the way in which individual toolbar buttons are drawn. Like the `OnCustomDraw` routine, it has a `DefaultDraw` parameter which is `True` by default. While I was playing around with this routine, I happened to discover that the mere act of adding a

do-nothing `OnCustomDrawButton` routine (ie, an event handler containing nothing more than a comment, so as to prevent the IDE from auto-deleting it) was enough to cause the toolbar buttons to lose their raised 3D borders. Compare Figures 3 and Figure 4 to see what I mean. Initially, I thought this was due to a bug in the VCL wrapper, but having examined the VCL source code, it's clear that the wrapper returns different information to the API according to whether or not the application program has assigned to the various custom draw event handlers. This doesn't seem to be a terribly clean way of doing things, but given Microsoft's somewhat idiosyncratic approach here, perhaps this was the only option.

For the technically curious, the problem here is that the API wants to know 'up-front' what message notifications an application wishes to achieve. Thus, when a `CDDS_PREPAINT` notification code is received (see the Windows SDK documentation for more details) the application is expected to respond by indicating whether it wants to receive post-paint notifications, post-erase and item draw notifications. The only way the VCL can reasonably determine this is by checking to see whether the various custom draw event handlers are assigned. It obviously can't distinguish between a 'real' event handler and one that does nothing.

Things get even messier when we look at the tree-view and listview controls. Suppose I've got a listview control and I want to draw the currently selected item using a non-standard colour such as red. Doesn't exactly sound like rocket science, does it? Naively, I tried adding the code in Listing 3 to a `OnCustomDrawItem` handler.

➤ *Figure 3: It's equally easy to add a background image to a toolbar using the OnCustomDraw event, but if you try to get much more ambitious than this, you'll end up grappling with the API...*

➤ *Figure 4: The mere act of adding an empty OnCustomDrawButton handler will radically affect the appearance of the toolbar buttons, even though, strictly speaking, they should look just the same as Figure 3 at this point.*

This produced the result shown in Figure 5. As you can see, the two sub-items have been correctly drawn in red, but the item itself has been drawn using the default highlight colour, which isn't what we want. If you try making use of the newer `OnAdvancedCustomDrawItem` event handler, or even the snappily named `OnAdvancedCustomDrawSubItem`, you'll get the same result.

So what is going wrong? The key to the problem is the fact that the two sub-items are being correctly drawn, whereas the selected item is not. The code inside COMCTL32.DLL will attempt to draw the selected item using the default highlight colour (which in this case is navy blue) irrespective of the change we made to the `Canvas` property. The two sub-items, on the other hand, wouldn't normally be part of the 'selection bar' anyway, and consequently they are drawn using the colour that we specified.

To put things another way, the selected item isn't being drawn the way we want simply because, ironically, it `is` the selected item! If it wasn't selected, we could draw it anyway we like, but because it is selected, the common controls code will ignore the background colour and draw it the way it believes selected items should be drawn.

From this, you'll maybe guess that there is a workaround. If we could simply fool the COMCTL32.DLL code into thinking that the item isn't selected (even though it is) then it would draw the item using the background colour we specify which, in effect, is the new selection colour for this control.

The bad news is that the workaround is only accessible at the API level. The C code snippet in Listing 4 is taken from a recent newsgroup message which illustrates how easy it would be to fix the problem if we were coding at the API level.

Well, you get the idea. Because these data structures are directly accessible to the API-level programmer, he/she can easily sidestep a problem which is a right royal pain in VCL land.

➤ *Listing 2*

```
procedure TForm1.ToolBar1CustomDraw (Sender: TToolBar; const ARect: TRect;
  var DefaultDraw: Boolean);
var
  X, Y: Integer;
begin
  Y := ARect.Top;
  // Tile the bitmap over the toolbar
  while Y < ARect.Bottom do begin
    X := ARect.Left;
    while X < ARect.Right do begin
      Sender.Canvas.Draw (X, Y, Clouds);
      Inc (X, Clouds.Width);
    end;
    Inc (Y, Clouds.Height);
  end;
end;
```

Conclusions

Perhaps you're surprised to hear me talking in this way? As you know, I'm a great fan of Delphi, and a staunch supporter of the VCL framework. Almost always, VCL

programming is far simpler and more enjoyable than would be the case with MFC or straight API level programming. Almost always: but not always. Borland have a great talent for taking awkward API programming interfaces and simplifying them while adding extra power and flexibility. But when the underlying architecture is as messed up as this, there are limits to what even Borland can do.

After spending the last couple of days grappling with the vagaries of Microsoft's custom draw scheme, and wading through the cries for help in the various API-related newsgroups, I'm forced to the conclusion that the software giant has really excelled itself this time in terms of bad API design. Unfortunately, the problems are made worse for VCL developers (both Delphi and C++Builder) because we are abstracted further away from the underlying API messages and data structures. An unpleasant but quick 'tweak' like the one I discussed above becomes difficult, if not impossible, when you've got 20,000 lines of COMCTRLS.PAS interposed between you and the API.

```
procedure TForm1.ListView1CustomDrawItem(Sender: TCustomListView;
  Item: TListItem;
  State: TCustomDrawState;
  var DefaultDraw: Boolean);
begin
  if cdsSelected in State then Sender.Canvas.Brush.Color := clRed;
end;
```

➤ *Above: Listing 3*

➤ *Below: Listing 4*

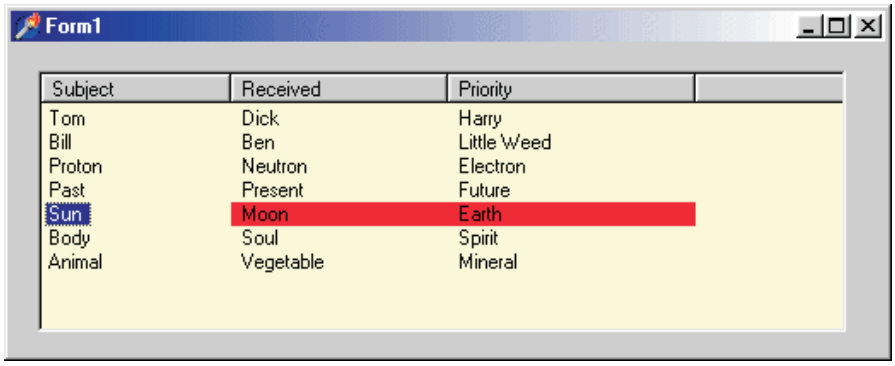
```
case CDDS_ITEMPREPAINT:
  // We only want to change things for the selected items
  if (pCustom->>nmcd.uItemState & CDIS_SELECTED) {
    // Change the background color to deep red.
    pCustom->>clrTextBk = RGB(255,0,0);
    pCustom->>nmcd.uItemState &= ~CDIS_SELECTED;
  }
```

In case you think I'm over-stating the case here, let me just refer you to a comment I made earlier when I introduced the new custom draw mechanism. I stated that the traditional owner draw mechanism gave maximum flexibility but was inconvenient if you simply wanted to change the font of a selected item. Ironically, after writing that, I discovered that *'Custom draw does not allow you to change the font attributes for selected items'* (taken from the MSDN introduction to owner draw). So the bottom line is that, even when used at the API level, custom draw won't let you do such a simple thing as changing the font of the selected item, and it will even fight you for trying to change the background colour of the selected item! Amazing! How's that for state of the art user interface design?

Ok, clever clogs, so how would *I* implement custom draw? At the VCL level, I'd like to simply be able to set a `TPicture` property as the background image of a control. I'd like to see separate event handlers for `OnDrawControlBackground`, `OnDrawItemBackground` and so forth.

Rather than worrying about pre-paint versus post-paint, pre-erase versus post-erase, the only issue should be whether you do your own custom stuff before or after you call `Inherited`. What I'm really arguing for is a much more *elegant* implementation of custom draw, rather than compromising the inherent elegance of the VCL library by surfacing Microsoft's underlying API klunkiness as part of the VCL programming interface. If Borland want to port the VCL across to Linux as part of the ongoing 'Delphi for Linux' project, they would do well to avoid sacrificing the VCL's elegance on the altar of Microsoft's increasingly terminally broken API.

Now, some folks will argue that it's impossible to build an elegant programming interface on top of an inelegant can of worms. Borland have done pretty well with the rest of the Windows API, but it may well be the case that what I'm asking for here isn't practical. If so, can I gently suggest that Borland point their web browsers at www.teemach.com and take a leaf out of David Berneda's book? The excellent TeeChart and TeeTree controls simply run rings around Microsoft's treeview and listview components, simply because they're not limited by the underpowered, badly designed COMCTL32.DLL. What we need are a few more native VCL controls of this calibre, rather than trying to provide support for something which, to be blunt, really ought to be put out of its misery.



➤ *Figure 5: Problems also arise when we try something as simple as changing the highlight colour of a list-view item. How about a property called HighlightColor? If only life were that simple.*

Even if Borland don't feel motivated to do so, recent initiatives such as the excellent Wine project (www.winehq.com) have provided an impetus which, I've no doubt, will bring forth an increasing number of alternatives to Microsoft's controls over the next year or so. And again, I've no doubt that many of these alternatives will be written using native VCL code. If these developers are careful to use the Windows API as little as possible, then there's a good chance that these controls will be easily portable to other platforms as Borland's 'Delphi for Linux' and other VCL-based systems become available. COMCTL32.DLL is dead. Long live the revolution!

Dave Jewell is a freelance consultant/programmer and technical journalist specialising in system-level Windows and DOS work. He is Technical Editor of *Developers Review* which is also published by iTec. You can email Dave at TechEditor@itecuk.com

➤ *Figure 6: You want a tree view? Now that's a tree-view! David Berneda's excellent TeeTree utility is an excellent example of the sort of flexible, native VCL control that I want to see more of, rather than grappling with the brain-dead COMCTL32 library. And yes, TeeTree can look a lot more conventional, if you want it to.*