Beating the System: RichEdit Controls, Part 2

by Dave Jewell

You will remember that in last month's column I began development of a RichEdit 2.0 control. This control, as you'll recall, requires the presence of the RICHED20.DLL file, a library which contains the underlying implementation code for the API-level control. At the time I wrote the original article, it was my expectation that Microsoft would do the decent thing and make this DLL redistributable, but sadly this has proved not to be the case. With luck, Microsoft will ship RICHED20.DLL as part of the forthcoming Windows 98 release. If not, then the water will continue to get muddier.

I finished up last month by promising to make further enhancements to the new control, and I also indicated that I'd encapsulate the new functionality into properties, methods and events of the `TRichEdit2` component. Let's do that now.

Information Hiding And Encapsulation

To begin with, the testbed client application contained a lot of code that really should have been in the control itself. As always, good component design is dependent upon attention to object oriented principles such as information hiding and encapsulation. Last month's client code knew far more about the rich edit control than was necessary: instead of modifying a property value to change the control's behaviour, it sent a low-level message directly to the window handle. That was bad.

Let's begin by making the automatic URL detection into a property of `TRichEdit2`. I've called this `Boolean` property `URLHighlight` and I've arranged things so that the property defaults to `True`. In order to do this, I've added a private variable, `fURLHighlight`, to the control along with a private method, `SetURLHighlight`, which fields assignments to this property and sends `em_AutoURLDetect` messages to the API-level window as described last month.

As you'll appreciate, when you specify that a property has a certain default value, you are effectively promising Delphi that the property will be assigned this value when the component is created. Normally, one 'makes good' on this promise inside the constructor, but we can't call `Perform` inside the `Create` method since the API-level window handle doesn't exist yet: this is a common error made by many novice Delphi programmers. Instead, we need to override the `CreateWnd` method and initialise the rich text edit control from there.

While we're at it, we need to rationalise the notification mechanism used to inform the client code that a URL link has been clicked. Rather than intercepting `wm_Notify` messages in the application code, a much cleaner, more Delphi-like, approach is to use a custom event such as `OnURLClicked`. In order to implement this, I declared a new event type as below:

```
TURLClickedEvent = procedure (Sender:
  TObject; const TheURL: String;
   Button: TMouseButton) of object;
```

This new event type passes the clicked URL text to the application program and also, as a convenience, provides an indication of which button was clicked. If you're developing an application using this code, you might want to do something completely different when a highlighted URL is right-clicked, for example.

Since we're no longer intercepting notify events at the client level, we need to intercept them within the control. Consequently, I wrote a new routine, `CNNotify`, which responds to notification messages. Inside `CNNotify` I check for an `en_Link` notification (as per last month's code) and call the `URLLinkNotification` method which has now moved into the control. It's important to call `Inherited` for messages that we're not interested in, so the original `TCustomRichEdit` code can do its stuff.

Within the `URLLinkNotification` method, the code retrieves the text corresponding to the URL link and then turns it into a string as before. This time, however, the code checks to see if our custom event handler is assigned and, if so, sends a `OnURLClicked` event to the application. For maximum flexibility, I discriminate between the three different mouse buttons and fill in the final argument of the event type based on the button that was pressed.

Listing 1 shows the changes that I made to the control in order to roll the URL auto-detection capability into the control. This is a partial code listing, a 'delta' of last month's code. Complete source listings are included on the cover disk. You'll notice that in the `URLLinkNotification` routine, I've left the option of you adding your own code to deal with mouse move events occurring over a URL link. Depending on your needs, you might want to define another event type to handle this.

Of Selections, Rows And Columns...

In last month's code, I demonstrated how to determine whether a rich edit control currently has a text selection through use of the `em_GetSel` message. However, use of this message is now discouraged because it will only work when the current selection is contained within the first 64Kb of text. The

```
type
  TURLClickedEvent = procedure (Sender: TObject;
    const TheURL: String; Button: TMouseButton) of object;
  TRichEdit2 = class (TCustomRichEdit)
  private
    fURLHighlight: Boolean;
    fURLClicked: TURLClickedEvent;
    procedure SetURLHighlight (Value: Boolean);
    procedure WMNCDestroy (var Message: TWMNCDestroy);
      message wm_NCDestroy;
    procedure CNNotify(var Message: TWMNotify);
      message cn_Notify;
  protected
    procedure CreateWnd; override;
    procedure URLLinkNotification (Link: Pointer);
  public
    constructor Create (AOwner: TComponent); override;
  published
    property URLHighlight: Boolean read fURLHighlight
      write SetURLHighlight default True;
    property OnURLClicked: TURLClickedEvent read fURLClicked
      write fURLClicked;
  end;
implementation
{$R *.DCR}
constructor TRichEdit2.Create (AOwner: TComponent);
begin
  Inherited Create (AOwner);
  fURLHighlight := True;
end;
procedure TRichEdit2.CreateWnd;
var
  mask: Integer;
begin
  Inherited CreateWnd;
  mask := Perform (em_GetEventMask, 0, 0) or enm_Link;
  Perform (em_SetEventMask, 0, mask);
  Perform (em_AutoURLDetect, Ord (fURLHighlight), 0);
end;
```

```
procedure TRichEdit2.SetURLHighlight (Value: Boolean);
begin
  if Value <> fURLHighlight then begin
    fURLHighlight := Value;
    Perform (em_AutoURLDetect, Ord (fURLHighlight), 0);
  end;
end;
procedure TRichEdit2.URLLinkNotification (Link: Pointer);
type
  // Need to redefine this - RICHTEXT.PAS gets it wrong!
  TTextRange = record
    chrg: TCharRange;
    lpstrText: PAnsiChar;
  end;
var
  sz: String;
  TextRange: TTextRange;
  pENLink: ^TENLink absolute Link;
begin
  with pENLink^ do begin
    SetLength (sz, chrg.cpMax - chrg.cpMin);
    TextRange.chrg := chrg;
    TextRange.lpstrText := Pointer (sz);
    Perform (em_GetTextRange, 0, Integer (@TextRange));
    case Msg of
      wm_MouseMove   :  ;
      wm_LButtonDown :  if Assigned (fURLClicked)
        then fURLClicked (Self, sz, mbLeft);
      wm_MButtonDown :  if Assigned (fURLClicked)
        then fURLClicked (Self, sz, mbMiddle);
      wm_RButtonDown :  if Assigned (fURLClicked)
        then fURLClicked (Self, sz, mbRight);
    end;
  end;
end;
procedure TRichEdit2.CNNotify (var Message: TWMNotify);
begin
  if Message.NMHdr^.Code <> en_Link then Inherited
    else URLLinkNotification (Message.NMHdr);
end;
```

➤ *Listing 1*

new routine to use is em_ExGetSel. To use this routine, you pass a pointer to a TCharRange data structure which contains a couple of 32-bit fields. On return from the call, these fields are set up with the starting and ending character positions for the text selection.

With this in mind, I wrote a new private method, GetGotSelection (!), and used it to implement a new boolean property called GotSelection. This property will always tell the client application whether or not there's a current selection. If you want immediate notification of text selection changes you can just use the OnSelectionChange event in the normal way.

While I was at it, I decided to add Row and Column properties to the control so that the client software could instantly determine the current row and column position within the text control. Much of the time, text processing programs don't really *need* this information, but it's customary to display it in a status bar at the bottom of the screen along with CapsLock status and so forth.

You might suppose that the rich edit control would provide messages such as em_GetLine and em_GetRow, but as ever things are a little more complex than that. In addition, things work differently according to whether or not the text control currently has a selection. At the Windows API level, the row and column numbers associated with RTF controls are zero-based, but this tends to be confusing to most end-users, only programmers and mathematicians start counting from zero! I've therefore arranged that both Row and Column start from 1 in terms of what's reported by the control.

The GetRow method sets the cp (character position) variable to -1 on the assumption that there's no text selection. If there is a text selection, then cp is assigned the character position that corresponds to the beginning of the selection. Once this is done, an em_LineFromChar message is sent to the control in order to obtain the actual line number.

For the sake of consistency, I've written this code such that if there's a text selection, the Row and Column properties will always return the row and column position of the *beginning* of the selection. According to the Microsoft documentation, if you pass a value of -1 as the wParam field and the control has a text selection, what gets returned is the line index of the beginning of the selection. However, this is wrong. In such cases, what you get back is the line index of the *end* of the selection. It's because of this undocumented behaviour (What? Me, surprised?!) that the GetRow method has to figure out whether there's a selection and adjust things accordingly.

Similarly, the GetColumn routine involves extra complexity if there's a text selection present. In such cases, it sends an em_ExLineFromChar message to the control in order to figure out the line number on which the selection begins. An em_LineIndex message is then sent to convert this line number into the character position that corresponds to the first character on the first line of the selection. Finally (phew!) this character position is subtracted from the character position of the start of the selection. I wouldn't be surprised if there isn't an easier way of figuring this out but it's been a long day, and this is the best I could come up with! The necessary additions to implement the GotSelection, Row and Column properties are summarised in Listing 2. You'll see that

```
  TRichEdit2 = class (TCustomRichEdit)
  private
    function GetRow: Integer;
    function GetColumn: Integer;
    function GetGotSelection: Boolean;
    function GetFirstLine: Integer;
  published
    property GotSelection: Boolean read GetGotSelection;
    property Row: Integer read GetRow;
    property Column: Integer read GetColumn;
    property FirstLine: Integer read GetFirstLine;
  end;
function TRichEdit2.GetGotSelection: Boolean;
begin
  Perform (em_ExGetSel, 0, Integer (@fLastCR));
  Result := fLastCR.cpMin <> fLastCR.cpMax;
end;
function TRichEdit2.GetRow: Integer;
var cp: Integer;
begin
  cp := -1;
  if GetGotSelection then cp := fLastCR.cpMin;
  Result := Perform (em_LineFromChar, cp, 0) + 1;
end;
function TRichEdit2.GetColumn: Integer;
var lp: Integer;
begin
  lp := Perform (em_LineIndex, -1, 0);
  if GetGotSelection then lp := Perform (em_LineIndex, Perform
    (em_ExLineFromChar, 0, fLastCR.cpMin), 0);
  Result := fLastCR.cpMin - lp + 1;
end;
function TRichEdit2.GetFirstLine: Integer;
begin
  Result := Perform (em_GetFirstVisibleLine, 0, 0);
end;
```

➤ *Listing 2*

I've also added another integer property, `FirstLine`. Reading this property sends an `em_GetFirstVisibleLine` message to the edit control and returns the number of the first visible line of text in the control, assuming that you've used the vertical scroll bar to scroll down through the text (if you haven't, then of course this will always return zero). If you want to give the user an indication of the absolute line number within a large file then you should add the `FirstLine` property to the value of the `Row` property.

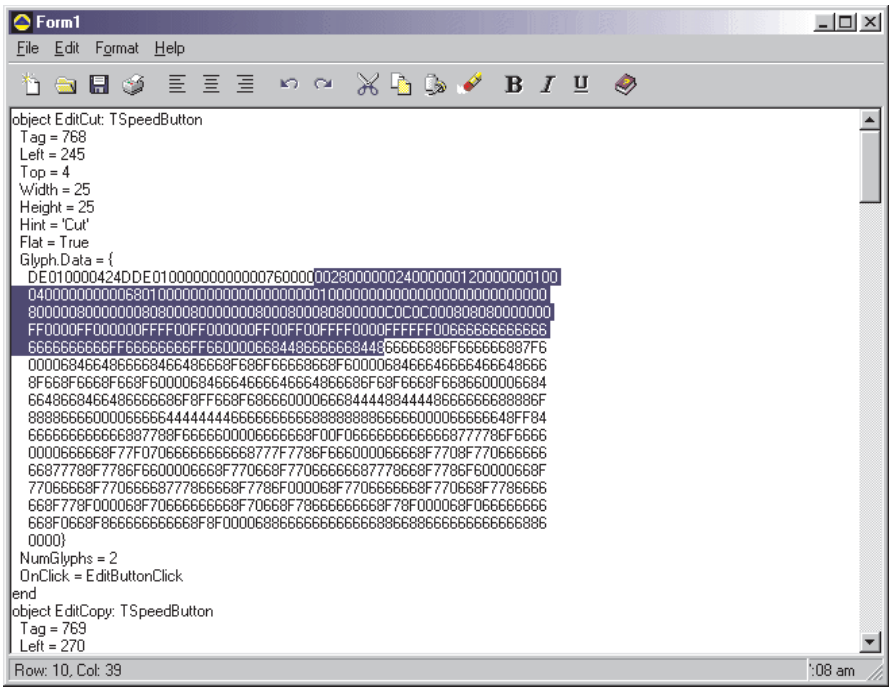
Extended Support For Undo And Redo

Last month I mentioned that the version 2.0 rich text edit control includes enhanced support for undo/redo. In version 1, you've only got a simple one-step undo facility. In version 2, you've got a full multi-step undo/redo facility. The great thing about this is that your application doesn't have to manage its own undo/redo buffer, everything is handled internally by the control *[If want to know how to implement your own multiple undo/redo, check out Warren Kovach's detailed article in the May 1998 issue, #33. Ed].*

So how do we implement this functionality? If you look at this month's project (see the disk and Figure 1) you'll see that I've added a redo button to the toolbar. In last month's code, the `Tag` property of the undo button was set to `$304` (`wm_Undo`) meaning that a `wm_Undo` message was sent to the control whenever this button was pressed. For better consistency with the rich text control, I've changed this to `$C7` (`em_Undo`) which is the correct notification message for use with edit controls. Additionally, the new redo button has a tag value of `em_Redo` and fires this message to the text control when clicked. With just these few trivial changes, you've automatically got a multi-level undo/redo facility: try it and see! It really is that simple.

However, the rich text control provides a number of facilities for adding a little more 'spit and polish' to the final result. For example, how do we know whether to enable the undo/redo buttons? In last month's code, I sent an `em_CanUndo` message to the control in order to determine whether or not to enable the undo button. It turns out that rich edit 2.0 implements another message, `em_CanRedo`, whose purpose should require no further elaboration! I packaged up these two messages into a couple of boolean properties, `CanUndo` and `CanRedo`, and incorporated them into the edit control. The revised `EnableDisableToolbar` code now references these properties to enable or disable the undo/redo buttons.

➤ *Figure 1: Here's our testbed application now sporting a row and column indication in the status bar, along with a multiple undo/redo facility accessible via the taskbar.*



```
object EditCut: TSpeedButton
  Tag = 768
  Left = 245
  Top = 4
  Width = 25
  Height = 25
  Hint = 'Cut'
  Flat = True
  Glyph.Data = {
    DE010000424DDE01000000000000760000002800000024000000120000000100
    0400000000006801000000000000000000001000000000000000000000000000
    80000080000000808000800000008000800080800000C0C0C000808080000000
    FF0000FF000000FFFF00FF000000FF00FF00FFFF0000FFFFFF00666666666666
    6666666666FF66666666FF66000066844866666684486666886F666666887F6
    000068466486666684664866668F686F66668668F6000068466646666466648666
    8F668F6668F668F60000684666466664666486668 6F68F6668F66866000006684
    6648668466486666686F8FF668F686660000666844448844448666666668886F
    8888666600006666444444446666666666888888886666000066666648FF84
    66666666666887788F6666600006666668F00F066666666666668777786F6666
    0000666668F77F07066666666668777F7786F666000066668F7708F770666666
    66877788F7786F660000666 8F770668F770666666687778668F7786F60000668F
    77066668F77066666877786668F7786F000068F7706666668F770668F7786666
    668F778F000068F70666666668F70668F7866666668F78F000068F066666666
    668F0668F86666666666 8F8F00006886666666666666886688666666666666886
    0000}
  NumGlyphs = 2
  OnClick = EditButtonClick
end
object EditCopy: TSpeedButton
  Tag = 769
  Left = 270
```

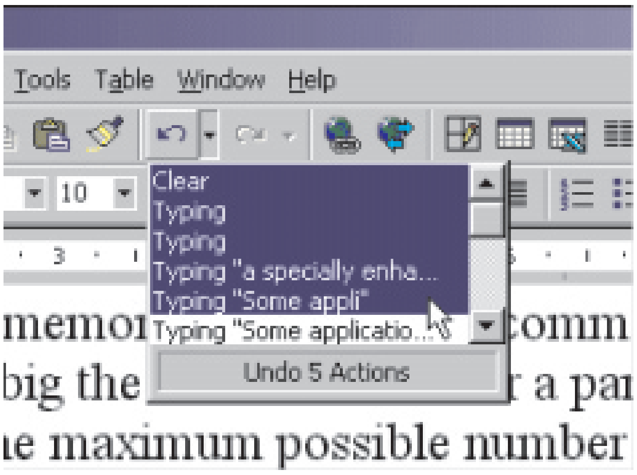
Row: 10, Col: 39 | :08 am

When examining the code, you'll notice that I've implemented both the `boolean` properties using a common access routine called `GetBoolProp`. If you create a control that has a large number of properties of the same type, all of which require very similar code, you can greatly reduce the size of the generated code by using indexed properties and a common access routine as illustrated here. Moreover, you can often add another property simply by adding a one line property declaration to the class declaration, and another one line clause to the associated `case` statement. However, Object Pascal can go even better than this, as we shall see.

So far, so good. But how many actions can be stored in the control's built-in undo/redo buffer? The answer is 100, by default. However, it is possible to use a new message, `em_SetUndoLimit`, to increase or decrease the size of the buffer. Microsoft point out that if you increase the buffer, there must be sufficient memory available to accommodate the resized buffer, but they don't give any indication of how big the buffer will be for a particular number of 'actions', so to speak. Nor is there any guidance on the maximum possible number of actions that can be assigned to the undo buffer. Finally, although there's a `em_SetUndoLimit` message for resizing the undo buffer, there is no corresponding `em_GetUndoLimit` message for determining the current undo limit.

Despite these shortcomings, I decided to bite the bullet and add an `UndoLimit` property to the edit control. After all, the write-only nature of the undo limit isn't a problem because we can just store the current value in a private member variable. However, in terms of the allowable values for the undo limit, I decided to set the minimum allowable value to 10 and the maximum allowable value to 400. Any property assignment outside of this range is politely ignored.

Some applications provide a specially enhanced user interface for undo/redo actions, according to the type of actions that are available. For example, take a look at Figure 2, which shows part of Microsoft Word 97. The drop-down combo box lists the available actions, with some indication of what each specific action does. As you drag the cursor towards the bottom of the combo, more and more actions are scheduled for undoing.



➤ *Figure 2: Word 97 provides a fancy undo/redo facility which indicates the different categories of actions available on the undo/redo 'stack'.*

The rich text edit 2.0 control doesn't give us as much functionality as this (there is obviously some custom book-keeping of undo and redo actions being performed inside Word itself), but it does offer some useful help. For instance, there are two new messages called `em_GetUndoName` and `em_GetRedoName`. These two messages return an integer value which indicates the type of action (if any) which will be undone or redone in response to an `em_Undo` or `em_Redo` message. In effect, these messages let us 'peek' at the topmost message in the undo/redo buffer. The C/C++ declaration for the returned integer value looks like this:

```
typedef enum _undonameid {
  UID_UNKNOWN    = 0,
  UID_TYPING     = 1,
  UID_DELETE     = 2,
  UID_DRAGDROP   = 3,
  UID_CUT        = 4,
  UID_PASTE      = 5
} UNDONAMEID;
```

I turned this enumeration type into the equivalent Pascal type (`TUndoRedoType`) and implemented a couple of properties, `UndoType`, `RedoType`, to return values from this enumeration type. As before, I wrote a common access routine, `GetUndoRedoType`, which is used by both of these properties: the passed index value simply tells the access routine which message to send to the control. Like the other read-only properties that I've introduced, do bear in mind that these properties won't appear in the Property Inspector because it will not display read-only properties.

As a quick philosophical aside, if you find this use of indexed properties interesting, you might not realise that you can take things a stage further in the interests of efficiency and tight code. For example, consider what would happen if I were to declare the `UndoType` and `RedoType` properties like those in Listing 3.

In this case, the required message type is directly encoded as the index value. This effectively means that we can completely eliminate the `case` statement from the access routine, and the code for `GetUndoRedoType` collapses down to that shown in Listing 4.

This is an extremely powerful technique that relatively few Delphi programmers seem to be aware of. In this case, we're effectively using the `Index` attribute of a property in just the same way that we can use the `Tag` property of a group of related push buttons to discriminate between them, as in my sample application that accompanies this article. Just as the `Tag` property can discriminate between a number of similar controls, the `Index` property can be used to discriminate between a number of similar properties whose implementations are bundled into the same access routine. While not applicable in all circumstances, such techniques really can help shrink your source and object code and are a real testimony to Ander's genius for language design. The code additions that relate to undo/redo are summarised in Listing 5.

```
property UndoType: TUndoRedoType index em_GetUndoName read GetUndoRedoType;
property RedoType: TUndoRedoType index em_GetRedoName read GetUndoRedoType;
```

➤ *Listing 3*

```
function TRichEdit2.GetUndoRedoType (Index: Integer): TUndoRedoType;
begin
  Result := TUndoRedoType (Perform (Index, 0, 0));
end;
```

➤ *Listing 4*

```
type
  TundoRedoType = ( uidUnknown, uidTyping, uidDelete, uidDragDrop, uidCut,
    uidPaste );
  TRichEdit2 = class (TCustomRichEdit)
  private
    fUndoLimit: Integer;
    function GetBoolProp (Index: Integer): Boolean;
    procedure SetUndoLimit (Value: Integer);
    function GetUndoRedoType (Index: Integer): TUndoRedoType;
  published
    property CanUndo: Boolean index 0 read GetBoolProp;
    property CanRedo: Boolean index 1 read GetBoolProp;
    property UndoLimit: Integer read fUndoLimit write SetUndoLimit default 100;
    property UndoType: TUndoRedoType index 0 read GetUndoRedoType;
    property RedoType: TUndoRedoType index 1 read GetUndoRedoType;
  end;
function TRichEdit2.GetBoolProp (Index: Integer): Boolean;
begin
  Result := False;  { Stop compiler whinging }
  case Index of
    0 : Result := Perform (em_CanUndo, 0, 0) <> 0;
    1 : Result := Perform (em_CanRedo, 0, 0) <> 0;
  end;
end;
procedure TRichEdit2.SetUndoLimit (Value: Integer);
begin
  if (fUndoLimit <> Value) and (Value >= 10) and (Value <= 400) then begin
    fUndoLimit := Value;
    Perform (em_SetUndoLimit, Value, 0);
  end;
end;
function TRichEdit2.GetUndoRedoType (Index: Integer): TUndoRedoType;
begin
  Result := uidUnknown;  { Stop compiler whinging }
  case Index of
    0 : Result := TUndoRedoType (Perform (em_GetUndoName, 0, 0));
    1 : Result := TUndoRedoType (Perform (em_GetRedoName, 0, 0));
  end;
end;
```

➤ *Listing 5*

Conclusions

There's a great deal more that could be done with this control, as you can discover if you read through Microsoft's SDK documentation and the RICHEDIT.PAS file that accompanies Delphi 3.0 and 4.0. However, you will undoubtedly have your own ideas about how you want to extend the control and how you want to implement properties. As a general rule, I don't like to burden a control with dozens of different properties, all of which do nothing more than (for example) return some `boolean` value. For example, you might want to implement a set of properties that indicate whether the current selection (or the text at the current caret position) is bold, underlined, italic, embossed, link text, and so forth. Rather than implementing a host of `boolean` properties, one for each attribute, I'd encourage you to implement a related set of controls as (for example) a set type which shows which bits are set and which are cleared. Take a look at how Inprise implemented the `Style` sub-property in the `Font` property and you'll soon see what I'm talking about.

This month and last, I've tried to demonstrate some of the fun things you can do with version 2.0 of the rich edit control and I hope I've succeeded in that. The only fly in the ointment (and it's a fairly substantially sized fly) is the uncertain redistribution status of RICHED20.DLL. I've spoken to Redmond about this and nobody was able to give me a straight answer as to whether or not you could ship the DLL with your product, so effectively you're on your own! If anyone is able to get chapter and verse on this from Microsoft, do please let us know so we can pass on the good/bad news.

As with last month's column, the disk includes the testbed source and executable, a program which is designed only to highlight various aspects of what I've covered here. It is not intended as a fully-fledged word processor and many parts of the program (such as open/save file dialogs) are simply not implemented. As noted last time round, this code was written using the Raize components version 1.6 to implement the toolbar and the status bar. Thus, if you don't have Raize components, you won't be able to rebuild the code. If you insist on rebuilding the program, and you don't want to pay for the Raize components in order to do so, then you can get the shareware version of Raize Components from www.raize.com. Finally, please note that in order to reduce disk space requirements, this executable was built using packages. It expects to find the Delphi 3 VCL runtime code, but for your convenience I have incorporated the Raize code into the executable itself. Thus, you can try the demo without even installing the Raize components.

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 contact Dave at Dave@HexManiac.com