The RichBar Example

This bonus chapter is provided with *Mastering Delphi 6*. It is an introduction to the basic features of the RichBar example, discussed in Chapter 7 of the book, and of the follow-up MdEdit example, discussed in Chapter 8.

This document explains how you create a simple editor based on the RichEdit control, using Delphi 6. The program has a toolbar and implements several features, including a complete scheme for opening and saving the text files, discussed in this document. In fact, we want to be able to ask the user to save any modified file before opening a new one, to avoid losing any changes. Sounds like a professional application, doesn't it?

File Operations

The most complex part of this program is implementing the commands of the File pull-down menu—New, Open, Save, and Save As. In each case, we need to track whether the current file has changed, saving the file only if it has. We should prompt the user to save the file each time the program creates a new file, loads an existing one, or terminates.

To accomplish this, I've added a field, a property, and three public methods to the class describing the form of the application:

```
private
  FModified: Boolean;
  FileName: string;
  procedure SetModified(const Value: Boolean);
  property Modified: Boolean read FModified write SetModified;
public
  function SaveChanges: Boolean;
  function Save: Boolean;
  function SaveAs: Boolean;
```

The `FileName` string and the `Modified` property are set when the form is created, by executing the code used to define a new file. These vales are later changed when a new file is loaded or the user renames a file with the Save As command. Here is the startup code:

```
procedure TFormRichNote.FormCreate(Sender: TObject);
begin
  Application.Title := Caption;
  NewExecute (Self);
end;
```

The value of the flag changes as soon as you type new characters in the RichEdit control (in its `OnChange` event handler):

```
procedure TFormRichNote.RichEdit1Change(Sender: TObject);
begin
```

```
    // enables save operations
    Modified := True;
  end;
```

When a new file is created, the program checks whether the text has been modified. If so, it calls the SaveChanges function, which asks the user whether to save the changes, discard them, or skip the current operation:

```
procedure TFormRichNote.New1Click(Sender: TObject);
begin
  if not Modified or SaveChanges then
  begin
    RichEdit1.Text := '';
    Modified := False;
    FileName := '';
    Caption := 'RichNote - [Untitled]';
  end;
end;
```

If the creation of a new file is confirmed, some simple operations take place, including using '*Untitled*' instead of the file name in the form's caption.

Short-Circuit Evaluation

The expression `if not Modified or SaveChanges then` requires some explanation. By default, Pascal performs what is called "short-circuit evaluation" of complex conditional expressions. The idea is simple: if the expression `not Modified` is true, we are sure that the whole expression is going to be true, and we don't need to evaluate the second expression. In this particular case, the second expression is a function call, and the function is called only if `Modified` is True. This behavior of `or` and `and` expressions can be changed by setting a Delphi compiler option called Complete Boolean Eval. You can find it on the Compiler page of the Project Options dialog box.

The message box displayed by the SaveChanges function has three options. If the user selects the Cancel button, the function returns False. If the user selects No, nothing happens (the file is not saved) and the function returns True, to indicate that although we haven't actually saved the file, the requested operation (such as creating a new file) can be accomplished. If the user selects Yes, the file is saved and the function returns True.

In the code of this function, notice in particular the call to the MessageDlg function used as the value of a case statement:

```
function TFormRichNote.SaveChanges: Boolean;
begin
```

```
    case MessageDlg ('The document ' + filename + ' has changed.' + #13#13 +
      'Do you want to save the changes?', mtConfirmation, mbYesNoCancel, 0) of
    idYes:
      // call Save and return its result
      Result := Save;
    idNo:
      // don't save and continue
      Result := True;
    else // idCancel:
      // don't save and abort operation
      Result := False;
    end;
  end;
```

NOTE In the `MessageDlg` call above, I've added explicit newline characters (`#13`) to improve the readability of the output. As an alternative to using a numeric character constant, you can call `Chr(13)`.

To actually save the file, another function is invoked: `Save`. This method saves the file if it already has a proper file name or asks the user to enter a name, calling the `SaveAs` functions. These are two more internal functions, not directly connected with menu items:

```
function TFormRichNote.Save: Boolean;
begin
  if Filename = '' then
    Result := SaveAs // ask for a file name
  else
  begin
    RichEdit1.Lines.SaveToFile (FileName);
    Modified := False;
    Result := True;
  end;
end;

function TFormRichNote.SaveAs: Boolean;
begin
  SaveDialog1.FileName := Filename;
  if SaveDialog1.Execute then
  begin
    Filename := SaveDialog1.FileName;
    Save;
    Caption := 'RichNote - ' + Filename;
    Result := True;
  end
  else
    Result := False;
end;
```

I use two functions to perform the Save and Save As operations for completeness, even if the RichBar program has only a Save button and not a Save As button. The MdEdit version in Chapter 8 offers this extra feature. Moreover, the Save button is enabled only if the file has not been modified, as indicated in the `SetModified` method:

```
procedure TFormRichNote.SetModified(const Value: Boolean);
begin
  FModified := Value;
  tbtnSave.Enabled := Modified;
end;
```

Opening a file is much simpler. Before loading a new file, the program checks whether the current file has changed, asking the user to save it with the `SaveChanges` function, as before. The `OpenExecute` method is based on the OpenDialog component, another default dialog box provided by Windows and supported by Delphi:

```
procedure TFormRichNote.OpenExecute(Sender: TObject);
begin
  if not Modified or SaveChanges then
    if OpenDialog1.Execute then
    begin
      Filename := OpenDialog1.FileName;
      RichEdit1.Lines.LoadFromFile (FileName);
      Modified := False;
      Caption := 'RichNote - ' + FileName;
    end;
end;
```

The only other detail related to file operations is that both the OpenDialog and SaveDialog components of the form have a particular value for their `Filter` and `DefaultExt` properties, as you can see in the following fragment from the textual description of the form:

```
object OpenDialog1: TOpenDialog
  DefaultExt = 'rtf'
  FileEditStyle = fsEdit
  Filter = 'Rich Text File (*.rtf)|*.rtf|Any file (*.*)|*.*'
  Options = [ofHideReadOnly, ofPathMustExist,ofFileMustExist]
end
```

The string used for the `Filter` property contains four pairs of substrings, separated by the | symbol. Each pair has a description of the type of file that will appear in the File Open or File Save dialog box, and the filter to be applied to the files in the directory, such as `*.RTF`. To set the filters in Delphi, you can simply invoke the editor of this property, which displays a list with two columns.

The file-related methods above are also called from the `FormCloseQuery` method (the handler of the `OnCloseQuery` event), which is called each time the user tries to close the form, terminating the program. We can make this happen in various ways: by double-clicking the system menu icon, by selecting the system menu's Close command, by pressing Alt+F4, or by calling the `Close` method in the code, as in the File ➢ Exit menu command.

In `FormCloseQuery`, you can decide whether to actually close the application by setting the `CanClose` parameter, which is passed by reference. Again, if the current file has been modified, we call the `SaveChanges` function and use its return value. Again we can use the short-circuit evaluation technique:

```
procedure TFormRichNote.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
begin
  CanClose := not Modified or SaveChanges;
end;
```

The last file-related command is the Print command. The RichEdit component includes print capabilities, and they are very simple to use. Here is the code, which actually produces a very nice printout:

```
procedure TFormRichNote.PrintExecute (Sender: TObject);
begin
  RichEdit1.Print (FileName);
end;
```

Conclusion

As mentioned at the beginning, the file support provided by this example is rather complex. This is something you'll probably need to handle in any file-related application. As this was too long for inclusion in the printed *Mastering Delphi 6*, I've decided to place it on the CD instead of skipping it altogether. You can now get back to the book (mainly Chapters 7 and 8) to see how the example can be extended in a number of different ways.