MDI Applications

CHAPTER

16

IN THIS CHAPTER

The Multiple Document Interface, otherwise known as *MDI,* was introduced to Windows 2.0 in the Microsoft Excel spreadsheet program. MDI gave Excel users the ability to work on more than one spreadsheet at a time. Other uses of MDI included the Windows 3.1 Program Manager and File Manager programs. Borland Pascal for Windows is another MDI application.

During the development of Windows 95, many developers were under the impression that Microsoft was going to eliminate MDI capabilities. Much to their surprise, Microsoft kept MDI as part of Windows 95 and there has been no further word about Microsoft's intention to get rid of it.

CAUTION

Microsoft has acknowledged that the Windows MDI implementation is flawed. It advised developers against continuing to build apps in the MDI model. Since then, Microsoft has returned to building MS apps in the MDI model but does so without using the Windows MDI implementation. You can still use MDI, but be forewarned that the Windows MDI implementation is still flawed, and Microsoft has no plans to fix those problems. What we present in this chapter is a safe implementation of the MDI model.

Handling events simultaneously between multiple forms might seem difficult. In traditional Windows programming, you had to have knowledge of the Windows class `MDICLIENT`, MDI data structures, and the additional functions and messages specific to MDI. With Delphi 5, creating MDI applications is greatly simplified. When you finish this chapter, you'll have a solid foundation for building MDI applications, which you can easily expand to include more advanced techniques.

Creating the MDI Application

To create MDI applications, you need familiarity with the form styles `fsMDIForm` and `fsMDIChild` and a bit of MDI programming methodology. The following sections present some basic concepts regarding MDI and show how MDI works with special MDI child forms.

Understanding MDI Basics

To understand MDI applications, first you must understand how they're constructed. Figure 16.1 shows an MDI application similar to one you'll build in this chapter.

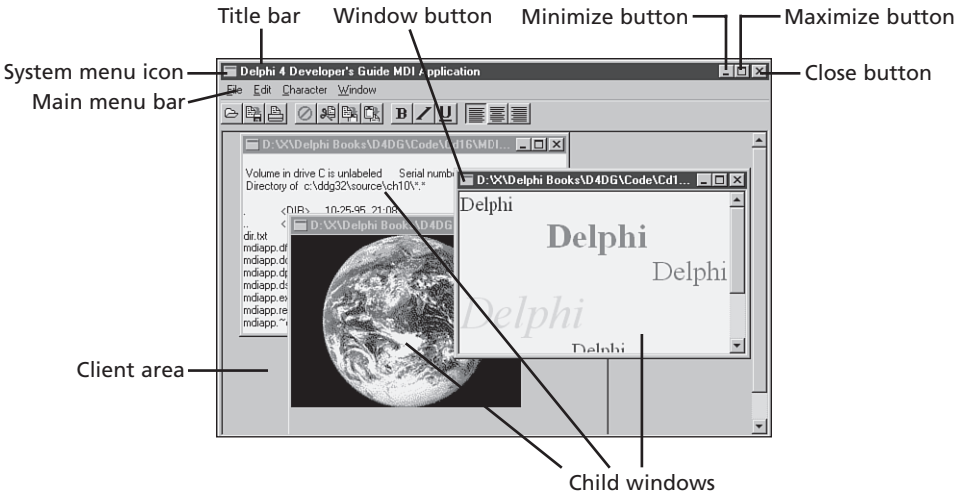

FIGURE 16.1
The structure of an MDI application.

Here are the windows involved with an MDI application:

- *Frame window.* The application's main window. It has a caption, menu bar, and system menu. Minimize, maximize, and close buttons appear in its upper-right corner. The blank space inside the frame window is known as its *client area* and is actually the client window.
- *Client window.* The manager for MDI applications. The client window handles all MDI-specific commands and manages the child windows that reside on its surface—including the drawing of MDI child windows. The client is created automatically by the *Visual Component Library* (VCL) when you create a frame window.
- *Child windows.* MDI child windows are your actual documents—text files, spreadsheets, bitmaps, and other document types. Child windows, like frame windows, have a caption, system menu, minimize, maximize, and close buttons, and possibly a menu. It's possible to place a help button on a child window. A child window's menu is combined with the frame window's menu. Child windows never move outside the client area.

Delphi 5 does not require you to be familiar with the special MDI window's messages. The client window is responsible for managing MDI functionality, such as cascading and tiling child windows. To cascade child windows using the traditional method, for example, use the Windows API function `SendMessage()` to send a `WM_MDICASCADE` message to the client window:

```
procedure TFrameForm.Cascade1Click(Sender: TObject);
begin
```

```
  SendMessage(ClientHandle, WM_MDICASCADE, 0, 0);
end;
```

In Delphi 5, just call the `Cascade()` method:

```
procedure TFrameForm.Cascade1Click(Sender: TObject);
begin
  cascade;
end;
```

The following sections show you a complete MDI application whose child MDI windows have the functionality of a text editor, a bitmap file viewer, and a rich text format editor. The purpose of this application is to show you how to build MDI applications whose child windows each display and edit different types of information. For example, the text editor allows you to edit any text-based file. The rich text editor allows you to edit rich text–formatted (`.rtf`) files. Finally, the bitmap viewer allows you to view any Windows bitmapped file.

We also show you how to perform some advanced MDI techniques using the Win32 API. These techniques mainly have to do with managing MDI child forms in an MDI-based application. First, we'll discuss the building of the child forms and their functionality. Then we'll talk about the main form.

The Child Form

As mentioned earlier, this MDI application contains three types of child forms: `TMdiEditForm`, `TMdiRTFForm`, and `TMdiBMPForm`. Each of these three types descends from `TMDIChildForm`, which serves as a base class. The following section describes the `TMDIChildForm` base class. The sections after that talk about the three child forms used in the MDI application.

The TMDIChildForm Base Class

The child forms used in the MDI application have some common functionality. They all have the same File menu and their `FormStyle` property is set to `fsMDIChild`. Additionally, they all make use of a `TToolBar` component. By deriving each child form from a base form class, you can avoid having to redefine these settings for each form. We defined a base form, `TMDIChildForm`, as shown in `MdiChildFrm.pas` (refer to Listing 16.1).

LISTING 16.1 `MdiChildFrm.pas`: A Unit Defining `TMDIChildForm`

```
unit MdiChildFrm;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Menus, ComCtrls, ToolWin, ImgList;
```

```
type

  TMDIChildForm = class(TForm)
  (* Component list removed, refer to online source. *)
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure mmiExitClick(Sender: TObject);
    procedure mmiCloseClick(Sender: TObject);
    procedure mmiOpenClick(Sender: TObject);
    procedure mmiNewClick(Sender: TObject);
    procedure FormActivate(Sender: TObject);
    procedure FormDeactivate(Sender: TObject);
end;

var
  MDIChildForm: TMDIChildForm;

implementation
uses MainFrm,  Printers;

{$R *.DFM}

procedure TMDIChildForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Action := caFree;
  { Reassign the toolbar parent }
  tlbMain.Parent := self;
  { If this is the last child form being displayed, then make the main form's
    toolbar visible }
  if (MainForm.MDIChildCount = 1) then
    MainForm.tlbMain.Visible := True
end;

procedure TMDIChildForm.mmiExitClick(Sender: TObject);
begin
  MainForm.Close;
end;

procedure TMDIChildForm.mmiCloseClick(Sender: TObject);
begin
  Close;
end;

procedure TMDIChildForm.mmiOpenClick(Sender: TObject);
begin
  MainForm.mmiOpenClick(nil);
end;
```

continues

LISTING 16.1 Continued

```
procedure TMDIChildForm.mmiNewClick(Sender: TObject);
begin
  MainForm.mmiNewClick(nil);
end;

procedure TMDIChildForm.FormActivate(Sender: TObject);
begin
  { When the form becomes active, hide the main form's toolbar and assign
    this child form's toolbar to the parent form.  Then display this
    child form's toolbar. }
  MainForm.tlbMain.Visible := False;
  tlbMain.Parent := MainForm;
  tlbMain.Visible := True;
end;

procedure TMDIChildForm.FormDeactivate(Sender: TObject);
begin
  { The child form becomes inactive when it is either destroyed or when another
    child form becomes active.  Hide this form's toolbar so that the next
form's
    toolbar will be visible. }
  tlbMain.Visible := False;
end;

end.
```

NOTE

Note that we have removed the component declarations for the `TMdiChildForm` base class from the printed text for space reasons.

`TMDIChildForm` contains event handlers for the menu items for its main menu as well as for some common tool buttons. Actually, the tool buttons are simply wired to the event handler of their corresponding menu item. Some of these event handlers call methods on the main form. For example, notice that the `mmiNewClick()` event handler calls the `MainForm.mmiNewClick()` event handler. `TMainForm.mmiNewClick()` contains functionality for creating a new MDI child form. You'll notice that there are other event handlers such as `mmiOpenClick()` and `mmiExitClick()` that call the respective event handlers on the main form. We'll cover `TMainForm`'s functionality later in the section "The Main Form."

Because each MDI child needs to have the same functionality, it makes sense to put this functionality into a base class from which the MDI child forms can descend. This way, the MDI

child forms do not have to define these same methods. They will inherit the main menu as well as the toolbar components that you see on the main form.

Notice in the `TMDIChildForm.FormClose()` event handler that you set the `Action` parameter to `caFree` to ensure that the `TMDIChildForm` instance is destroyed when closed. You do this because MDI child forms don't close automatically when you call their `Close()` method. You must specify, in the `OnClose` event handler, what you want done with the child form when its `Close()` method is called. The child form's `OnClose` event handler passes in a variable `Action`, of type `TCloseAction`, to which you must assign one of four possible values:

- `caNone`. Do nothing.
- `caHide`. Hide the form but don't destroy it.
- `caFree`. Free the form.
- `caMinimize`. Minimize the form (this is the default).

`TCloseAction` is an enumerated type.

When a form becomes active, its `OnActivate` event handler is called. You must perform some specific logic whenever a child form becomes active. Therefore, in the `TMdiChildForm.FormActivate()` event handler, you'll see that we make the main form's toolbar invisible while setting the child form's toolbar to visible. We also assign the main form as the parent to the child form's toolbar so that the toolbar appears on the main form and not on the child form. This is one way you might give the main form a different toolbar when a different type of MDI child form is active. The `OnDeactivate` event handler simply makes the child form's toolbar invisible. Finally, the `OnClose` event reassigns the child form as the parent to the toolbar, and if the current child form is the only child form, it makes the main form's toolbar visible. The effect is that the main form has a single toolbar with buttons that change depending on the type of active child form.

The Text Editor Form

The text editor form enables the user to load and edit any text file. This form, `TMdiEditForm`, is inherited from `TMDIChildForm`. `TMdiEditForm` contains a client-aligned `TMemo` component.

`TMdiEditForm` also contains `TPrintDialogs`, `TSaveDialog` and `TFontDialog` components.

`TMdiEditForm` is not an autocreated form and is removed from the list of autocreated forms in the Project Options dialog box.

NOTE

None of the forms, except for `TMainForm`, in the MDI project are automatically created and therefore have been removed from the list of autocreated forms. These forms are created dynamically in the project's source code.

TMdiEditForm's source code is given in Listing 16.2.

LISTING 16.2 MdiEditFrm.pas: A Unit Defining TMdiEditForm

```
unit MdiEditFrm;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Menus, ExtCtrls, Buttons, ComCtrls,
  ToolWin, MdiChildFrm, ImgList;

type

  TMdiEditForm = class(TMDIChildForm)
    memMainMemo: TMemo;
    SaveDialog: TSaveDialog;
    FontDialog: TFontDialog;
    mmiEdit: TMenuItem;
    mmiSelectAll: TMenuItem;
    N7: TMenuItem;
    mmiDelete: TMenuItem;
    mmiPaste: TMenuItem;
    mmiCopy: TMenuItem;
    mmiCut: TMenuItem;
    mmiCharacter: TMenuItem;
    mmiFont: TMenuItem;
    N8: TMenuItem;
    mmiWordWrap: TMenuItem;
    N9: TMenuItem;
    mmiCenter: TMenuItem;
    mmiRight: TMenuItem;
    mmiLeft: TMenuItem;
    mmiUndo: TMenuItem;
    N4: TMenuItem;
    mmiBold: TMenuItem;
    mmiItalic: TMenuItem;
    mmiUnderline: TMenuItem;
    PrintDialog: TPrintDialog;

    { File Event Handlers }
    procedure mmiSaveClick(Sender: TObject);
    procedure mmiSaveAsClick(Sender: TObject);
```

```
    { Edit Event Handlers }
    procedure mmiCutClick(Sender: TObject);
    procedure mmiCopyClick(Sender: TObject);
    procedure mmiPasteClick(Sender: TObject);
    procedure mmiDeleteClick(Sender: TObject);
    procedure mmiUndoClick(Sender: TObject);
    procedure mmiSelectAllClick(Sender: TObject);

    { Character Event Handlers }
    procedure CharAlignClick(Sender: TObject);
    procedure mmiBoldClick(Sender: TObject);
    procedure mmiItalicClick(Sender: TObject);
    procedure mmiUnderlineClick(Sender: TObject);
    procedure mmiWordWrapClick(Sender: TObject);
    procedure mmiFontClick(Sender: TObject);

    { Form Event Handlers }
    procedure FormCloseQuery(Sender: TObject; var CanClose: Boolean);
    procedure mmiPrintClick(Sender: TObject);
  public
    { User Defined Methods }
    procedure OpenFile(FileName: String);
    procedure SetButtons;
  end;

var
  MdiEditForm: TMdiEditForm;

implementation

uses Printers;

{$R *.DFM}
{ File Event Handlers }

procedure TMdiEditForm.mmiSaveClick(Sender: TObject);
begin
  inherited;
  { If there isn't a caption, then there isn't already a filename.
    Therefore, call mmiSaveAsClick since it gets a filename. }
  if Caption = '' then
    mmiSaveAsClick(nil)
  else begin
    { Save to the file specified by the form's Caption. }
    memMainMemo.Lines.SaveToFile(Caption);
    memMainMemo.Modified := false; // Set to false since the text is saved.
```

continues

LISTING 16.2 Continued

```
  end;
end;

procedure TMdiEditForm.mmiSaveAsClick(Sender: TObject);
begin
  inherited;
  SaveDialog.FileName := Caption;
  if SaveDialog.Execute then
  begin
    { Set caption to filename specified by SaveDialog1 since this
      may have changed. }
    Caption := SaveDialog.FileName;
    mmiSaveClick(nil); // Save the file.
  end;
end;

{ Edit Event Handlers }

procedure TMdiEditForm.mmiCutClick(Sender: TObject);
begin
  inherited;
  memMainMemo.CutToClipBoard;
end;

procedure TMdiEditForm.mmiCopyClick(Sender: TObject);
begin
  inherited;
  memMainMemo.CopyToClipBoard;
end;

procedure TMdiEditForm.mmiPasteClick(Sender: TObject);
begin
  inherited;
  memMainMemo.PasteFromClipBoard;
end;

procedure TMdiEditForm.mmiDeleteClick(Sender: TObject);
begin
  inherited;
  memMainMemo.ClearSelection;
end;

procedure TMdiEditForm.mmiUndoClick(Sender: TObject);
begin
  inherited;
```

```
  memMainMemo.Perform(EM_UNDO, 0, 0);
end;

procedure TMdiEditForm.mmiSelectAllClick(Sender: TObject);
begin
  inherited;
  memMainMemo.SelectAll;
end;

{ Character Event Handlers }
procedure TMdiEditForm.CharAlignClick(Sender: TObject);
begin
  inherited;
  mmiLeft.Checked := false;
  mmiRight.Checked := false;
  mmiCenter.Checked := false;

{ TAlignment is defined by VCL as:

  TAlignment = (taLeftJustify, taRightJustify, taCenter);

  Therefore each of the menu items contains the appropriate Tag property
  whose value represents one of the TAlignment values: 0, 1, 2 }

  { If the menu invoked this event handler, set it to checked and
    set the alignment for the memo }
  if Sender is TMenuItem then
  begin
    TMenuItem(Sender).Checked := true;
    memMainMemo.Alignment := TAlignment(TMenuItem(Sender).Tag);
  end
  { If a TToolButton from the main form invoked this event handler,
    set the memo's alignment and then check the appropriate TMenuItem. }
  else if Sender is TToolButton then
  begin
    memMainMemo.Alignment := TAlignment(TToolButton(Sender).Tag);
    case memMainMemo.Alignment of
      taLeftJustify:   mmiLeft.Checked := True;
      taRightJustify:  mmiRight.Checked := True;
      taCenter:        mmiCenter.Checked := True;
    end;
  end;
  SetButtons;
end;

procedure TMdiEditForm.mmiBoldClick(Sender: TObject);
```

continues

LISTING 16.2 Continued

```
begin
  inherited;
  if not mmiBold.Checked then
    memMainMemo.Font.Style := memMainMemo.Font.Style + [fsBold]
  else
    memMainMemo.Font.Style := memMainMemo.Font.Style - [fsBold];
  SetButtons;
end;

procedure TMdiEditForm.mmiItalicClick(Sender: TObject);
begin
  inherited;
  if not mmiItalic.Checked then
    memMainMemo.Font.Style := memMainMemo.Font.Style + [fsItalic]
  else
    memMainMemo.Font.Style := memMainMemo.Font.Style - [fsItalic];
  SetButtons;
end;

procedure TMdiEditForm.mmiUnderlineClick(Sender: TObject);
begin
  inherited;
  if not mmiUnderline.Checked then
    memMainMemo.Font.Style := memMainMemo.Font.Style + [fsUnderline]
  else
    memMainMemo.Font.Style := memMainMemo.Font.Style - [fsUnderline];
  SetButtons;
end;

procedure TMdiEditForm.mmiWordWrapClick(Sender: TObject);
begin
  inherited;
  with memMainMemo do
  begin
    WordWrap := not WordWrap;
    { Remove scrollbars if Memo1 is wordwrapped since they're not
      required. Otherwise, make sure scrollbars are present. }
    if WordWrap then
      ScrollBars := ssVertical
   else
      ScrollBars := ssBoth;
    mmiWordWrap.Checked := WordWrap;
  end;
end;

procedure TMdiEditForm.mmiFontClick(Sender: TObject);
```

```
begin
  inherited;
  FontDialog.Font := memMainMemo.Font;
  if FontDialog.Execute then
    memMainMemo.Font := FontDialog.Font;
end;

{ Form Event Handlers }
procedure TMdiEditForm.FormCloseQuery(Sender: TObject;
     var CanClose: Boolean);
{ This procedure ensures that the user has saved the contents of the
  memo if it was modified since the last time the file was saved. }
const
  CloseMsg = '''%s'' has been modified, Save?';
var
  MsgVal: integer;
  FileName: string;
begin
  inherited;
  FileName := Caption;
  if memMainMemo.Modified then
  begin
    MsgVal := MessageDlg(Format(CloseMsg, [FileName]), mtConfirmation,
mbYesNoCancel, 0);
    case MsgVal of
      mrYes:    mmiSaveClick(Self);
      mrCancel: CanClose := false;
    end;
  end;
end;

procedure TMdiEditForm.OpenFile(FileName: string);
begin
  memMainMemo.Lines.LoadFromFile(FileName);
  Caption := FileName;
end;

procedure TMdiEditForm.SetButtons;
{ This procedure ensures that menu items and buttons on the main form
  accurately reflect various settings for the memo. }
begin
  mmiBold.Checked := fsBold in memMainMemo.Font.Style;
  mmiItalic.Checked := fsItalic in memMainMemo.Font.Style;
  mmiUnderLine.Checked := fsUnderline in memMainMemo.Font.Style;

  tbBold.Down      := mmiBold.Checked;
```

continues

LISTING 16.2 Continued

```
  tbItalic.Down    := mmiItalic.Checked;
  tbUnderline.Down := mmiUnderLine.Checked;
  tbLAlign.Down    := mmiLeft.Checked;
  tbRAlign.Down    := mmiRight.Checked;
  tbCAlign.Down    := mmiCenter.Checked;
end;

procedure TMdiEditForm.mmiPrintClick(Sender: TObject);
var
  i: integer;
  PText: TextFile;
begin
  inherited;
  if PrintDialog.Execute then
  begin
    AssignPrn(PText);
    Rewrite(PText);
    try
      Printer.Canvas.Font := memMainMemo.Font;
      for i := 0 to memMainMemo.Lines.Count -1 do
        writeln(PText, memMainMemo.Lines[i]);
    finally
      CloseFile(PText);
    end;
  end;
  end;

end.
```

Most of the methods for `TMdiEditForm` are event handlers for the various menus in `TMdiEditForm`'s main menu, the same menu inherited from `TMdiChildForm`. Also notice that additional menu items have been added to the main menu that apply specifically to `TMdiEditForm`.

You'll notice that there are no event handlers for the File, New, File, Open, File, Close, and File, Exit menus because they're already linked to `TMdiChildForm`'s event handlers.

The event handlers for the Edit menu items are all single-line methods that interact with the `TMemo` component. For example, you'll notice that the event handlers `mmiCutClick()`, `mmiCopyClick()`, and `mmiPasteClick()` interact with the Windows Clipboard in order to perform cut, copy, and paste operations. The other edit event handlers perform various editing functions on the memo component that have to do with deleting, clearing, and selecting text.

The Character menu applies various formatting attributes to the memo.

Notice that we stored a unique value in the `Tag` property of the `TToolButton` components for setting text alignment. This `Tag` value represents a value in the `TAlignment` enumerated type. This value is extracted from the `Tag` value of the `TToolButton` component that invoked the event handler to set the appropriate alignment for the memo component.

All menu items and tool buttons that set text alignment are wired to the `CharAlignClick()` event handler. This is why you have to check and respond appropriately in the event handler depending on whether a `TMenuItem` or `TToolButton` component invoked the event.

`CharAlignClick()` calls the `SetButtons()` method, which sets various menu items and components accordingly based on the memo's attributes.

The `mmiWordWrapClick()` event handler simply toggles the memo's wordwrap attribute and then the `Checked` property for the menu item. This method also specifies whether the memo component contains scrollbars based on its word-wrapping capability.

The `mmiFontClick()` event handler invokes a `TFontDialog` component and applies the selected font to the memo. Notice that before launching the `FontDialog` component, the `Font` property is set to reflect the memo's font so that the correct font is displayed in the dialog box.

The `mmiSaveAsClick()` event handler invokes a `TSaveDialog` component to get a filename from the user to which the memo's contents will be saved. When the file is saved, the `TMdiEditForm.Caption` property is set to reflect the new filename.

The `mmiSaveClick()` event handler calls the `mmiSaveAsClick()` event handler if a filename doesn't exist. This is the case if the user creates a new file instead of opening an existing one. Otherwise, the memo's contents are saved to the existing file specified by the `MdiEditForm.Caption` property. Notice that this event handler also sets `memMainMemo.Modified` to `False`. `Modified` is automatically set to `True` whenever the user changes the contents of a `TMemo` component. However, it's not set to `False` automatically whenever its contents are saved.

The `FormCloseQuery()` method is the event handler for the `OnCloseQuery` event. This event handler evaluates the `memMainMemo.Modified` property when the user attempts to close the form. If the memo has been modified, the user is notified and asked whether he or she wants to save the contents of the memo.

The public method `TMdiEditForm.OpenFile()` loads the file specified by the `FileName` parameter and places the file's contents into the `memMainMemo.Lines` property and then sets the form's `Caption` to reflect this filename.

That completes the functionality for `TMdiEditForm`. The other forms are somewhat similar in functionality.

The Rich Text Editor Form

The rich text editor enables the user to load and edit rich text–formatted files. This form, `TMdiRtfForm`, is derived from `TMDIChild`. It contains a client-aligned `TRichEdit` component.

`TMdiRtfForm` and `TMdiEditForm` are practically identical except that `TMdiRtfForm` contains a `TRichEdit` component as its editor; `TMdiEditForm` uses a `TMemo` component. `TMdiRtfForm` differs from the text editor in that the text attributes applied to the `TRichEdit` component affect paragraphs or selected text in the `TRichEdit` component; they affect the entire text with the `TMemo` component.

The source code for `TMdiRtfForm` is shown in Listing 16.3.

LISTING 16.3 `MdiRtfFrm.pas`: A Unit Defining `TMdiRtfForm`

```
unit MdiRtfFrm;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, MdiChildFrm, StdCtrls, ComCtrls,
  ExtCtrls, Buttons, Menus, ToolWin, ImgList;

type

  TMdiRtfForm = class(TMDIChildForm)
    reMain: TRichEdit;
    FontDialog: TFontDialog;
    SaveDialog: TSaveDialog;
    mmiEdit: TMenuItem;
    mmiSelectAll: TMenuItem;
    N7: TMenuItem;
    mmiPaste: TMenuItem;
    mmiCopy: TMenuItem;
    mmiCut: TMenuItem;
    mmiCharacter: TMenuItem;
    mmiFont: TMenuItem;
    N8: TMenuItem;
    mmiWordWrap: TMenuItem;
    N9: TMenuItem;
    mmiCenter: TMenuItem;
    mmiRight: TMenuItem;
    mmiLeft: TMenuItem;
    mmiUndo: TMenuItem;
    mmiDelete: TMenuItem;
```

```
    N4: TMenuItem;
    mmiBold: TMenuItem;
    mmiItalic: TMenuItem;
    mmiUnderline: TMenuItem;

    { File Event Handlers }
    procedure mmiSaveClick(Sender: TObject);
    procedure mmiSaveAsClick(Sender: TObject);

    { Edit Event Handlers }
    procedure mmiCutClick(Sender: TObject);
    procedure mmiCopyClick(Sender: TObject);
    procedure mmiPasteClick(Sender: TObject);
    procedure mmiDeleteClick(Sender: TObject);
    procedure mmiUndoClick(Sender: TObject);
    procedure mmiSelectAllClick(Sender: TObject);

    { Character Event Handlers }
    procedure CharAlignClick(Sender: TObject);
    procedure mmiBoldClick(Sender: TObject);
    procedure mmiItalicClick(Sender: TObject);
    procedure mmiUnderlineClick(Sender: TObject);
    procedure mmiWordWrapClick(Sender: TObject);
    procedure mmiFontClick(Sender: TObject);

    { Form Event Handlers }
    procedure FormShow(Sender: TObject);
    procedure FormCloseQuery(Sender: TObject; var CanClose: Boolean);
    procedure reMainSelectionChange(Sender: TObject);
    procedure mmiPrintClick(Sender: TObject);
  public
    { User-Defined Functions. }
    procedure OpenFile(FileName: String);
    function GetCurrentText: TTextAttributes;
    procedure SetButtons;
  end;

var
  MdiRtfForm: TMdiRtfForm;

implementation
{$R *.DFM}
{ File Event Handlers }

procedure TMdiRtfForm.mmiSaveClick(Sender: TObject);
begin
```

continues

LISTING 16.2 Continued

```
  inherited;
  reMain.Lines.SaveToFile(Caption);
end;

procedure TMdiRtfForm.mmiSaveAsClick(Sender: TObject);
begin
  inherited;
  SaveDialog.FileName := Caption;
  if SaveDialog.Execute then
  begin
    Caption := SaveDialog.FileName;
    mmiSaveClick(Sender);
  end;
end;

{ Edit Event Handlers }

procedure TMdiRtfForm.mmiCutClick(Sender: TObject);
begin
  inherited;
  reMain.CutToClipBoard;
end;

procedure TMdiRtfForm.mmiCopyClick(Sender: TObject);
begin
  inherited;
  reMain.CopyToClipBoard;
end;

procedure TMdiRtfForm.mmiPasteClick(Sender: TObject);
begin
  inherited;
  reMain.PasteFromClipBoard;
end;

procedure TMdiRtfForm.mmiDeleteClick(Sender: TObject);
begin
  inherited;
  reMain.ClearSelection;
end;

procedure TMdiRtfForm.mmiUndoClick(Sender: TObject);
begin
  inherited;
  reMain.Perform(EM_UNDO, 0, 0);
```

```
end;

procedure TMdiRtfForm.mmiSelectAllClick(Sender: TObject);
begin
  inherited;
  reMain.SelectAll;
end;

{ Character Event Handlers }

procedure TMdiRtfForm.CharAlignClick(Sender: TObject);
begin
  inherited;
  mmiLeft.Checked := false;
  mmiRight.Checked := false;
  mmiCenter.Checked := false;

  { If a TMenuItem invoked this event handler, set its checked
    property to true and set the attribute to RichEdit1's current
    paragraph. }
  if Sender is TMenuItem then
  begin
    TMenuItem(Sender).Checked := true;
    with reMain.Paragraph do
      if mmiLeft.Checked then
        Alignment := taLeftJustify
      else if mmiRight.Checked then
        Alignment := taRightJustify
      else if mmiCenter.Checked then
        Alignment := taCenter;
  end
  { If one of the main form's tool buttons invoked this event handler
    set the attribute to reMain's current paragraph and set the
    alignment menu items accordingly. }
  else if Sender is TSpeedButton then
  begin
    reMain.Paragraph.Alignment :=
       TAlignment(TSpeedButton(Sender).Tag);
    case reMain.Paragraph.Alignment of
      taLeftJustify: mmiLeft.Checked := True;
      taRightJustify: mmiRight.Checked := True;
      taCenter: mmiCenter.Checked := True;
    end;
  end;
  SetButtons;
end;
```

continues

LISTING 16.2 Continued

```
procedure TMdiRtfForm.mmiBoldClick(Sender: TObject);
begin
  inherited;
  if not mmiBold.Checked then
    GetCurrentText.Style := GetCurrentText.Style + [fsBold]
  else
    GetCurrentText.Style := GetCurrentText.Style - [fsBold];
end;

procedure TMdiRtfForm.mmiItalicClick(Sender: TObject);
begin
  inherited;
  if not mmiItalic.Checked then
    GetCurrentText.Style := GetCurrentText.Style + [fsItalic]
  else
    GetCurrentText.Style := GetCurrentText.Style - [fsItalic];
end;

procedure TMdiRtfForm.mmiUnderlineClick(Sender: TObject);
begin
  inherited;
  if not mmiUnderline.Checked then
    GetCurrentText.Style := GetCurrentText.Style + [fsUnderline]
  else
    GetCurrentText.Style := GetCurrentText.Style - [fsUnderline];
end;

procedure TMdiRtfForm.mmiWordWrapClick(Sender: TObject);
begin
  inherited;
  with reMain do
  begin
    { Remove scrollbars if Memo1 is wordwrapped since they're not
      required. Otherwise, make sure scrollbars are present. }
    WordWrap := not WordWrap;    if WordWrap then
      ScrollBars := ssVertical
    else
      ScrollBars := ssNone;
    mmiWordWrap.Checked := WordWrap;
  end;
end;

procedure TMdiRtfForm.mmiFontClick(Sender: TObject);
begin
  inherited;
  FontDialog.Font.Assign(reMain.SelAttributes);
```

```
  if FontDialog.Execute then
    GetCurrentText.Assign(FontDialog.Font);
  reMain.SetFocus;
end;

{ Form Event Handlers }

procedure TMdiRtfForm.FormShow(Sender: TObject);
begin
  inherited;
  reMainSelectionChange(nil);
end;


procedure TMdiRtfForm.FormCloseQuery(Sender: TObject;
   var CanClose: Boolean);
{ This procedure ensures that the user has saved the contents of
  reMain if it was modified since the last time the file was saved. }
const
  CloseMsg = '''%s'' has been modified, Save?';
var
  MsgVal: integer;
  FileName: string;
begin
  inherited;
  FileName := Caption;
  if reMain.Modified then
  begin
    MsgVal := MessageDlg(Format(CloseMsg, [FileName]), mtConfirmation,
mbYesNoCancel, 0);
    case MsgVal of
      mrYes: mmiSaveClick(Self);
      mrCancel: CanClose := false;
    end;
  end;
end;

procedure TMdiRtfForm.reMainSelectionChange(Sender: TObject);
begin
  inherited;
  SetButtons;
end;

procedure TMdiRtfForm.OpenFile(FileName: String);
begin
  reMain.Lines.LoadFromFile(FileName);
```

continues

LISTING 16.2 Continued

```
  Caption := FileName;
end;

function TMdiRtfForm.GetCurrentText: TTextAttributes;
{ This procedure returns the text attributes of the current paragraph
  or based on the selected text of reMain.}
begin
  if reMain.SelLength > 0 then
    Result := reMain.SelAttributes
  else
    Result := reMain.DefAttributes;
end;

procedure TMdiRtfForm.SetButtons;
{ Ensures that the controls on the form reflect the
  current attributes of the paragraph by looking at the paragraph
  attributes themselves and setting the controls accordingly. }
begin

  with reMain.Paragraph do
  begin
    mmiLeft.Checked := Alignment = taLeftJustify;
    mmiRight.Checked := Alignment = taRightJustify;
    mmiCenter.Checked := Alignment = taCenter;
  end;

  with reMain.SelAttributes do
  begin
    mmiBold.Checked := fsBold in Style;
    mmiItalic.Checked := fsItalic in Style;
    mmiUnderline.Checked := fsUnderline in Style;
  end;
  mmiWordWrap.Checked := reMain.WordWrap;

  tbBold.Down      := mmiBold.Checked;
  tbItalic.Down    := mmiItalic.Checked;
  tbUnderline.Down := mmiUnderline.Checked;
  tbLAlign.Down    := mmiLeft.Checked;
  tbRAlign.Down    := mmiRight.Checked;
  tbCAlign.Down    := mmiCenter.Checked;
end;

procedure TMdiRtfForm.mmiPrintClick(Sender: TObject);
begin
```

```
  inherited;
  reMain.Print(Caption);
end;

end.
```

Like `TMdiEditForm`, most of `TMdiRtfForm`'s methods are event handlers for the various menu items and tool buttons. These event handlers are similar to `TMdiEditForm`'s event handlers.

`TMdiRtfForm`'s File menu items invoke the File menu items of the `TMdiChildForm` base class. Recall that `TMdiChildForm` is the ancestor to `TMdiRtfForm`. The event handlers, `mmiSaveClick()` and `mmiSaveAsClick()`, both call `reMain.Lines.SaveToFile()` to save `reMain`'s contents.

The event handlers for `TMdiRtfForm`'s Edit menu items are single-line methods similar to the `TMdiEditForm`'s Edit menu event handlers except that these event handlers call methods applicable to `reMain`. The method names are the same as the memo methods that perform the same operations.

`TMdiRtfForm`'s Character menu items modify the alignment of *paragraphs* or selected text within the `TRichEdit` component (as opposed to the text within the entire component, as is the behavior with a `TMemo` component). Whether these attributes are applied to a paragraph or to selected text depends on the return value of the `GetCurrentText()` function. `GetCurrentText()` determines whether any text is selected by looking at the value of `TRichEdit.SelLength`. A zero value indicates that no text is selected. The `TRichEdit.SelAttributes` property refers to any selected text in the `TRichEdit` component. `TRichEdit.DefAttributes` refers to the current paragraph of the `TRichEdit` component.

The `mmiFontClick()` event handler allows the user to specify font attributes for a paragraph. Note that a paragraph can also refer to selected text.

Word wrapping is handled the same with the `TRichEdit` component as with the `TMemo` component in the text editor.

The `TRichEdit.OnSelectionChange` event handler is available to allow the programmer to provide some functionality whenever the selection of the component has changed. When the user moves the caret within the `TRichEdit` component, the component's `SelStart` property value changes. Because this action causes the `OnSelectionChange` event handler to be called, code was added to change the status of the various `TMenuItem` and `TSpeedButton` components on the main form to reflect the attributes of the text as the user scrolls through text in the `TRichEdit` component. This is necessary because text attributes in the `TRichEdit` component can differ; this is not the case with a `TMemo` component because attributes applied to a `TMemo` component apply to the entire component.

In functionality, the rich text editor form and the text editor form are, for the most part, very similar. The main difference is that the rich text editor allows users to change the attributes for separate paragraphs or selected text; the text editor is incapable of doing this.

The Bitmap Viewer—The Third MDI Child Form

The bitmap viewer enables the user to load and view Windows bitmap files. Like the other two MDI child forms, the bitmap viewer form, `TMdiBmpForm`, is derived from the `TMDIChildForm` base class. It contains a client-aligned `TImage` component.

`TMdiBmpForm` contains only its inherited `TMainMenu` component. Listing 16.4 shows the source code that defines `TMdiBmpForm`.

LISTING 16.4 `MdiBmpFrm.pas`: A Unit Defining `TMdiBmpForm`

```
unit MdiBmpFrm;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls, Forms, Dialogs,
  MdiChildFrm, ExtCtrls, Menus, Buttons, ComCtrls, ToolWin, ImgList;

type
  TMdiBMPForm = class(TMDIChildForm)
    mmiEdit: TMenuItem;
    mmiCopy: TMenuItem;
    mmiPaste: TMenuItem;
    imgMain: TImage;
    procedure mmiCopyClick(Sender: TObject);
    procedure mmiPasteClick(Sender: TObject);
    procedure mmiPrintClick(Sender: TObject);
  public
    procedure OpenFile(FileName: string);
  end;

var
  MdiBMPForm: TMdiBMPForm;

implementation

uses ClipBrd, Printers;

{$R *.DFM}

procedure TMdiBMPForm.OpenFile(FileName: String);
```

```
begin
  imgMain.Picture.LoadFromFile(FileName);
  Caption := FileName;
end;

procedure TMdiBMPForm.mmiCopyClick(Sender: TObject);
begin
  inherited;
  ClipBoard.Assign(imgMain.Picture);
end;

procedure TMdiBMPForm.mmiPasteClick(Sender: TObject);
{ This method copies the contents from the clipboard into imgMain }
begin
  inherited;
  // Copy clipboard content to imgMain
  imgMain.Picture.Assign(ClipBoard);
  ClientWidth := imgMain.Picture.Width;
  { Adjust clientwidth to adjust the scollbars }
  VertScrollBar.Range := imgMain.Picture.Height;
  HorzScrollBar.Range := imgMain.Picture.Width;
end;

procedure TMdiBMPForm.mmiPrintClick(Sender: TObject);
begin
  inherited;

  with ImgMain.Picture.Bitmap do
  begin
      Printer.BeginDoc;
      Printer.Canvas.StretchDraw(Canvas.ClipRect, imgMain.Picture.Bitmap);
      Printer.EndDoc;
  end; { with }
end;

end.
```

There is not as much code for `TMdiBmpForm` as there was for the two previous forms. The File menu items invoke the `TMDIChildForm`'s event handlers just as the `TMdiEditForm` and `TMdiRtfForm` File menu items do. The Edit menu items copy and paste the bitmap to and from the Windows Clipboard, respectively. Before calling the `TImage.Picture.Assign()` method to assign the Clipboard data to the `TImage` component. The `TImage` component recognizes both the `CF_BITMAP` and `CF_PICTURE` formats as bitmaps.

The Windows Clipboard

The Clipboard provides the easiest way for two applications to share information. It's nothing more than a global memory block that Windows maintains for any application to access through a specific set of Windows functions.

The Clipboard supports several standard formats, such as text, OEM text, bitmaps, and metafiles; it also supports other specialized formats. Additionally, you can extend the Clipboard to support application-specific formats.

Delphi 5 encapsulates the Windows Clipboard with the global variable `Clipboard` of type `TClipBoard`, making it much easier for you to use. The `TClipBoard` class is covered in detail in Chapter 17, "Sharing Information with the Clipboard."

The Main Form

The main form is the form with which the user initially works to create or switch between MDI child forms. This form is appropriately named `MainForm`. `MainForm` serves as the parent to the text editor, bitmap viewer, and RTF editor MDI child forms.

`TMainForm` is not a descendant of `TMDIChildForm` as are the other forms discussed so far in this chapter. `TMainForm` has the `FormStyle` of `fsMDIForm` (the other three forms inherited the style `fsMDIChild` from `TMDIChild`). `TMainForm` contains a `TMainMenu` component and a `TOpenDialog` component. `TMainForm` also contains a toolbar that contains only one button. `TMainForm`'s source code is shown in Listing 16.5.

LISTING 16.5 `MdiMainForm.pas`: A Unit Defining `TMainForm`

```
unit MainFrm;

interface

uses
  WinTypes, WinProcs, Classes, Graphics, Forms, Controls, Menus,
  StdCtrls, Messages, Dialogs, SysUtils, ComCtrls,
  ToolWin, ExtCtrls, Buttons, ImgList;

type
  TMainForm = class(TForm)
    mmMain: TMainMenu;
    OpenDialog: TOpenDialog;
    mmiFile: TMenuItem;
    mmiExit: TMenuItem;
    N3: TMenuItem;
    mmiOpen: TMenuItem;
```

```
    mmiNew: TMenuItem;
    mmiWindow: TMenuItem;
    mmiArrangeIcons: TMenuItem;
    mmiCascade: TMenuItem;
    mmiTile: TMenuItem;
    mmiCloseAll: TMenuItem;
    tlbMain: TToolBar;
    ilMain: TImageList;
    tbFileOpen: TToolButton;

    { File Event Handlers }
    procedure mmiNewClick(Sender: TObject);
    procedure mmiOpenClick(Sender: TObject);
    procedure mmiExitClick(Sender: TObject);

    { Window Event Handlers }
    procedure mmiTileClick(Sender: TObject);
    procedure mmiArrangeIconsClick(Sender: TObject);
    procedure mmiCascadeClick(Sender: TObject);
    procedure mmiCloseAllClick(Sender: TObject);
  public
    { User defined methods }
    procedure OpenTextFile(EditForm: TForm; Filename: string);
    procedure OpenBMPFile(FileName: String);
    procedure OpenRTFFile(RTFForm: TForm; FileName: string);
  end;

var
  MainForm: TMainForm;

implementation
uses MDIBmpFrm, MdiEditFrm, MdiRtfFrm, FTypForm;

const
  { Define constants to represent file name extensions }
  BMPExt      = '.BMP';  // Bitmapped file
  TextExt     = '.TXT';  // Text file
  RTFExt      = '.RTF';  // Rich Text Format file

{$R *.DFM}

procedure TMainForm.mmiNewClick(Sender: TObject);
begin
  { Determine the file type the user wishes to open by calling the
    GetFileType function. Call the appropriate method based on the
    retrieved file type. }
```

continues

LISTING 16.5 Continued

```
  case GetFileType of
    mrTXT: OpenTextFile(nil, ''); // Open a text file.
    mrRTF: OpenRTFFile(nil, '');  // Open an RTF file.
    mrBMP:
     begin
        { Set the default filter for OpenDialog1 for BMP files. }
        OpenDialog.FilterIndex := 2;
        mmiOpenClick(nil);
     end;
  end;
end;

procedure TMainForm.mmiOpenClick(Sender: TObject);
var
  Ext: string[4];
begin
  { Call the appropriate method based on the file type of the file
    selected from OpenDialog1 }
  if OpenDialog.Execute then
  begin
    { Get the file's extension and compare it to determine the
      file type the user is opening. Call the appropriate method and
      pass in the file name. }
    Ext := ExtractFileExt(OpenDialog.FileName);
    if CompareStr(UpperCase(Ext), TextExt) = 0 then
      OpenTextFile(ActiveMDIChild, OpenDialog.FileName)
    else if CompareStr(UpperCase(Ext), BMPExt) = 0 then
      OpenBMPFile(OpenDialog.FileName)
    else if CompareStr(UpperCase(Ext), RTFExt) = 0 then
      OpenRTFFile(ActiveMDIChild, OpenDialog.FileName);
  end;
end;

procedure TMainForm.mmiExitClick(Sender: TObject);
begin
  Close;
end;

{ Window Event Handlers }

procedure TMainForm.mmiTileClick(Sender: TObject);
begin
  Tile;
end;

procedure TMainForm.mmiArrangeIconsClick(Sender: TObject);
```

```
begin
   ArrangeIcons;
end;

procedure TMainForm.mmiCascadeClick(Sender: TObject);
begin
  Cascade;
end;


procedure TMainForm.mmiCloseAllClick(Sender: TObject);
var
  i: integer;
begin
  { Close all forms in reverse order as they appear in the
    MDIChildren property. }
  for i := MdiChildCount - 1 downto 0 do
    MDIChildren[i].Close;
end;

{ User Defined Methods }
procedure TMainForm.OpenTextFile(EditForm: TForm; FileName: string);
begin
  { If EditForm is of a TEditForm type, then give the user the option
   of loading the file contents into this form. Otherwise, create a
   new TEditForm instance and load the file into that instance }
  if (EditForm <> nil) and (EditForm is TMdiEditForm) then
    if MessageDlg('Load file into current form?', mtConfirmation,
      [mbYes, mbNo], 0) = mrYes then
    begin
      TMdiEditForm(EditForm).OpenFile(FileName);
       Exit;
    end;
  { Create a new TEditForm and call its OpenFile() method }
  with TMdiEditForm.Create(self) do
    if FileName <> '' then
      OpenFile(FileName)
end;

procedure TMainForm.OpenRTFFile(RTFForm: TForm; FileName: string);
begin
  { If RTFForm is of a TRTFForm type, then give the user the option
   of loading the file contents into this form. Otherwise, create a
   new TRTFForm instance and load the file into that instance }
  if (RTFForm <> nil) and (RTFForm is TMdiRTFForm) then
    if MessageDlg('Load file into current form?', mtConfirmation,
```

continues

LISTING 16.5 Continued

```
      [mbYes, mbNo], 0) = mrYes then begin
        (RTFForm as TMdiRTFForm).OpenFile(FileName);
      Exit;
    end;
    { Create a new TRTFForm and call its OpenFile() method }
    with TMdiRTFForm.Create(self) do
      if FileName <> '' then
        OpenFile(FileName);
end;

procedure TMainForm.OpenBMPFile(FileName: String);
begin
  { Create a new TBMPForm instances and load a BMP file into it. }
  with TMdiBmpForm.Create(self) do
    OpenFile(FileName);
end;

end.
```

`TMainForm` uses another form, `FileTypeForm`, of the type `TFileTypeForm`. Listing 16.6 shows the source code for this form.

LISTING 16.6 The `FTYPFORM.PAS` Unit Defining `TFileTypeForm`

```
unit FTypForm;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls, Buttons;

const
  mrTXT = mrYesToAll+1;
  mrBMP = mrYesToAll+2;
  mrRTF = mrYesToAll+3;


type

  TFileTypeForm = class(TForm)
    rgFormType: TRadioGroup;
    btnOK: TButton;
    procedure btnOkClick(Sender: TObject);
  end;
```

```
var
  FileTypeForm: TFileTypeForm;


function GetFileType: Integer;

implementation

function GetFileType: Integer;
{ This function returns the file type selected by the user as
  represented by one of the above defined constants. }
begin
  FileTypeForm := TFileTypeForm.Create(Application);
  try
    Result := FileTypeForm.ShowModal;
  finally
    FileTypeForm.Free;
  end;
end;

{$R *.DFM}

procedure TFileTypeForm.btnOkClick(Sender: TObject);
begin
  { Return the correct modal result based on the selected file type }
  case rgFormType.ItemIndex of
   0: ModalResult := mrTXT;
   1: ModalResult := mrRTF;
   2: ModalResult := mrBMP;
  end;
end;

end.
```

`TFileTypeForm` is used to prompt the user for a file type to create. This form returns the `ModalResult` based on which `TRadioButton` the user selected to indicate the type of file. The `GetFileType()` function takes care of creating, showing, and freeing the `TFileTypeForm` instance. This function returns the `TFileTypeForm.ModalResult` property. This form is not automatically created and has been removed from the list of autocreated forms for the project.

`TMainForm`'s toolbar contains only one button, which is used to open the initial child form. When a child form becomes active, its toolbar replaces the main form's toolbar. This logic is handled by the `OnActivate` event of the child form. `TMainForm`'s public methods `OpenTextFile()`, `OpenRTFFile()`, and `OpenBMPFile()` are called from the event handler `TMainForm.mmiOpenClick()`, which is invoked whenever the user selects the File, Open menu.

`OpenTextFile()` takes two parameters: a `TForm` instance and a filename. The `TForm` instance represents the currently active form for the application. The reason for passing this `TForm` instance to the `OpenTextFile()` method is so that the method can determine whether the `TForm` passed to it is of the `TMdiEditForm` class. If so, it's possible that the user is opening a text file in the existing `TMdiEditForm` instance rather than creating a new `TMdiEditForm` instance. If a `TMdiEditForm` instance is passed to this method, the user is prompted whether he or she wants the text file to be placed into this `TForm` parameter. If the user replies no or the `TMdiEditForm` parameter is `nil`, a new `TMdiEditForm` instance is created.

`OpenRTFFile()` operates the same as `OpenTextFile()` except that it checks for a `TRFTForm` class as the currently active form represented by the `TForm` parameter. The functionality is the same.

`OpenBMPFile()` always assumes that the user is opening a new file. This is because the `TMdiBmpForm` is only a viewer and not an editor. If the form allowed the user to edit a bitmapped image, the `OpenBMPFile()` method would function as do `OpenTextFile()` and `OpenRTFFile()`.

The `mmiNewClick()` event handler calls the `GetFileType()` function to retrieve a file type from the user. It then calls the appropriate `Open`*`XXX`*`File()` method based on the return value. If the file is a `.bmp` file, the `OpenDialog.Filter` property is set to the BMP filter by default and the `mmiOpenClick()` method is invoked because the user is not creating a new `.bmp` file but is opening an existing one.

The `mmiOpenClick()` event handler invokes `OpenDialog` and calls the appropriate `Open`*`XXX`*`File()` method. Notice that `OpenTextFile()` and `OpenRTFFile()` are passed the `TMainForm.ActiveMDIChild` property as the first parameter. `ActiveMDIChild` is the MDI child that currently has focus. Recall that both these methods determine whether the user wants to open a file into an existing MDI child form. If no forms are active, `ActiveMDIChild` is `nil`. If `ActiveMDIChild` is pointing to a `TMdiRTFForm` and `OpenTextFile()` is called, `OpenTextFile()` still functions correctly because of this statement:

```
if (RTFForm <> nil) and (RTFForm is TMdiRTFForm) then
```

This statement determines whether `ActiveMDIChild` points to a `TMdiRtfForm`. If not, a new form is created.

The event handler, `mmiExitClick()`, calls `TMainForm.Close()`; this method not only closes the main form, it also terminates the application. If there are any child forms open at the time this event handler is invoked, the child forms are also closed and destroyed.

The Window menu event handlers are single-line methods that affect how the MDI child forms are arranged on the main form's client area. Figures 16.2 and 16.3 show tiled and cascaded forms, respectively.

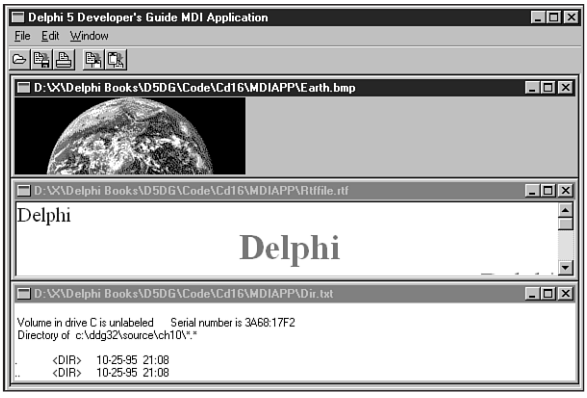

FIGURE 16.2
Tiled child forms.

FIGURE 16.3
Cascaded child forms.

The `mmiArrangeIconsClick()` method simply rearranges the icons in the main form's client area so that they're evenly spaced and do not overlap.

The `mmiCloseAllClick()` event handler closes all open MDI child forms. The loop that closes the child forms loops through all child forms in reverse order as they appear in the `MDIChildren` array property. The `MDIChildren` property is a zero-based array property of all MDI children active in an application. The `MDIChildCount` property is the number of children that are active.

This completes the discussion of the functionality of the MDI application. The following sections discuss some techniques and some of the components used with the various forms in the application.

Working with Menus

Using menus in MDI applications is no more difficult than using them in any other type of application. However, there are some differences in how menus work in MDI applications. The following sections show how an MDI application allows its child forms to share the same menu bar using a method called *menu merging*. You also learn how to make non-MDI applications share the same menu bar.

Merging Menus with MDI Applications

Take a look at the `TMainMenu` for the `TMdiEditForm`. By double-clicking the `TMainMenu` icon, you bring up the menu editor.

`TMdiEditForm`'s main menu contains three menu items along the menu bar. These items are File, Edit, and Character. Each of these menu items has a `GroupIndex` property that shows up in the Object Inspector as you click a menu item in the menu editor. Notice that the File menu item has a `GroupIndex` value of `0`. The Edit and Character menu items both have `GroupIndex` values of `1`.

Notice that `TMainForm`'s main menu has two menu items along its menu bar: File and Window. Like `TMdiEditForm`, `TMainForm`'s File menu item has a `GroupIndex` value of `0`. The Window menu item's `GroupIndex` property, on the other hand, has a value of `9`.

Also notice that the File menu for `TMainForm` and the File menu for `TMdiEditForm` have different submenu items. `TMdiEditForm`'s File menu has more submenu items than does `TMainForm`'s File menu.

The `GroupIndex` property is important because it allows menus of forms to be "merged." This means that when the main form launches a child form, the child form's main menu is merged with the main form's main menu. The `GroupIndex` property determines how the menus are ordered and which menus of the main form are replaced by menus of the child form. Note that menu merging applies only to menu items along the menu bar of a `TMainMenu` component and not to submenus.

Whenever a `GroupIndex` property for a child form's menu item has the same value as the `GroupIndex` property for a menu item on the main form, the child form's menu item replaces the main form's menu item. The remaining menus are arranged along the menu bar in the order specified by the `GroupIndex` properties of all combined menu items. When `MdiEditForm` is the active form in the project, the menu items that appear along the main form's menu bar are File, Edit, Character, and Window, in that order. Note that the File menu is `TMdiEditForm`'s File menu because both File menus have `GroupIndex` property values of `0`. Therefore, `TMdiChildForm`'s File menu replaces `TMainForm`'s File menu. The order of these menus directly reflects the order of the `GroupIndex` properties for each menu item along the menu bar: `0`, `1`, `1`, `9`.

NOTE

Although we don't use them here, there are certain numbering guidelines that you should follow so that your applications will better integrate with OLE Container's menu merging. These guidelines are explained in the "Borland Delphi Library Reference Guide."

This behavior is the same with the other forms in the MDI application. Whenever a form becomes active, the menu along the main menu bar changes to reflect the merging of menus for both the main form and child form. When you run the project, the menu bar changes depending on which child form is active.

Merging menus with MDI applications is automatic. As long as the values of the menu items' `GroupIndex` property is set in the order you specify, your menus merge correctly when you invoke MDI child forms.

For non-MDI applications, the process is just as easy but requires an extra step. We gave a quick example in Chapter 4, "Application Frameworks and Design Concepts," on merging menus in non-MDI applications when we discussed the `TNavStatForm`. However, in that application, we based this merging on child forms that were actually child windows to a control, other than the main form, and had to explicitly call the `Merge()` and `Unmerge()` functions. For merging menus with non–MDI-based applications in general, this process is not automatic, as it is with MDI applications. You must set the `AutoMerge` property to `True` for the `TMainMenu` on the form whose menus are to be merged with the main form. A sample project that shows menu merging for non-MDI forms can be found in the project `NonMDI.dpr` on the CD.

Adding a List of Open Documents to the Menu

To add a list of open documents to the Window menu, set the `WindowMenu` property of the main form to the menu item's instance that is to hold the list of open documents. For example, the `TMainForm.WindowMenu` property in the sample MDI application is set to `mmiWindow`, which refers to the Window menu along the menu bar. The selection you choose for this property must be a menu item that appears on the menu bar—it cannot be a submenu. The application displays a list of open documents in the Window menu.

Miscellaneous MDI Techniques

The following sections show various common techniques applicable to MDI applications.

Drawing a Bitmap in the MDI Client Window

When designing an MDI application, you might want to place a background image, such as a company logo, on the client area of an MDI application's main form. For regular (non-MDI) forms, this procedure is simple. You just place a `TImage` component on the form, set its `Align` property to `alClient`, and you're done (refer back to the bitmap viewer in the MDI sample application, earlier in this chapter). Placing an image on the main form of an MDI application, however, is a different story.

Recall that the client window of an MDI application is a separate window from the main form. The client window has many responsibilities of carrying out MDI-specific tasks, including the drawing of MDI child windows.

Think of it as though the main form is a transparent window over the client window. Whenever you place components such as `TButton`, `TEdit`, and `TImage` over the client area of the main form, these components are actually placed on the main form's transparent window. When the client window performs its drawing of child windows—or rather child forms—the forms are drawn *underneath* the components that appear on the main form, much like placing stickers on the glass of a picture frame (see Figure 16.4).

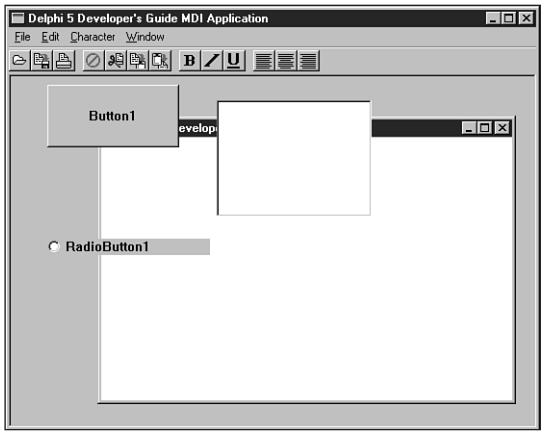

FIGURE 16.4
Client forms drawn underneath the main form's components.

So how do you go about drawing on the client window? Because Delphi 5 doesn't provide a VCL encapsulation of the client window, you must use the Win32 API. The method used is to subclass the client window and capture the message responsible for painting the client window's background—`WM_ERASEBKGND`. There, you take over the default behavior and perform your own custom drawing.

The following code is from the project `MdiBknd.dpr` on the CD. This project is an MDI application with a `TImage` component that contains a bitmap. From the menu, you can specify how to draw the image on the MDI client window—centered, tiled, or stretched, as shown respectively in Figures 16.5, 16.6, and 16.7.

FIGURE 16.5
The MDI client window with a centered image.

FIGURE 16.6
The MDI client window with a tiled image.

FIGURE 16.7
The MDI client window with a stretched image.

Listing 16.7 shows the unit code that performs the drawing.

LISTING 16.7 Drawing Images on the MDI Client Window

```
unit MainFrm;

interface

uses Windows, SysUtils, Classes, Graphics, Forms, Controls, Menus,
  StdCtrls, Dialogs, Buttons, Messages, ExtCtrls, JPeg;

type
  TMainForm = class(TForm)
    mmMain: TMainMenu;
    mmiFile: TMenuItem;
    mmiNew: TMenuItem;
    mmiClose: TMenuItem;
    N1: TMenuItem;
    mmiExit: TMenuItem;
    mmiImage: TMenuItem;
    mmiTile: TMenuItem;
    mmiCenter: TMenuItem;
    mmiStretch: TMenuItem;
    imgMain: TImage;
    procedure mmiNewClick(Sender: TObject);
    procedure mmiCloseClick(Sender: TObject);
    procedure mmiExitClick(Sender: TObject);
    procedure mmiTileClick(Sender: TObject);
```

```
  private
    FOldClientProc,
    FNewClientProc: TFarProc;
    FDrawDC: hDC;
    procedure CreateMDIChild(const Name: string);
    procedure ClientWndProc(var Message: TMessage);
    procedure DrawStretched;
    procedure DrawCentered;
    procedure DrawTiled;
  protected
    procedure CreateWnd; override;
  end;

var
  MainForm: TMainForm;

implementation

uses MdiChildFrm;

{$R *.DFM}

procedure TMainForm.CreateWnd;
begin
  inherited CreateWnd;
  // Turn the ClientWndProc method into a valid window procedure
  FNewClientProc := MakeObjectInstance(ClientWndProc);
  // Get a pointer to the original window procedure
  FOldClientProc := Pointer(GetWindowLong(ClientHandle, GWL_WNDPROC));
  // Set ClientWndProc as the new window procedure
  SetWindowLong(ClientHandle, GWL_WNDPROC, LongInt(FNewClientProc));
end;

procedure TMainForm.DrawCentered;
{ This procedure centers the image on the form's client area }
var
  CR: TRect;
begin
  GetWindowRect(ClientHandle, CR);
   with imgMain do
     BitBlt(FDrawDC, ((CR.Right - CR.Left) - Picture.Width) div 2,
            ((CR.Bottom - CR.Top) - Picture.Height) div 2,
            Picture.Graphic.Width, Picture.Graphic.Height,
            Picture.Bitmap.Canvas.Handle, 0, 0, SRCCOPY);
end;
```

continues

LISTING 16.7 Continued

```
procedure TMainForm.DrawStretched;
{ This procedure stretches the image on the form's client area }
var
  CR: TRect;
begin
  GetWindowRect(ClientHandle, CR);
  StretchBlt(FDrawDC, 0, 0, CR.Right, CR.Bottom,
             imgMain.Picture.Bitmap.Canvas.Handle, 0, 0,
             imgMain.Picture.Width, imgMain.Picture.Height, SRCCOPY);
end;

procedure TMainForm.DrawTiled;
{ This procedure tiles the image on the form's client area }
var
  Row, Col: Integer;
  CR, IR: TRect;
  NumRows, NumCols: Integer;
begin
  GetWindowRect(ClientHandle, CR);
  IR := imgMain.ClientRect;
  NumRows := CR.Bottom div IR.Bottom;
  NumCols := CR.Right div IR.Right;
  with imgMain do
    for Row := 0 to NumRows+1 do
      for Col := 0 to NumCols+1  do
        BitBlt(FDrawDC, Col * Picture.Width, Row * Picture.Height,
               Picture.Width, Picture.Height, Picture.Bitmap.Canvas.Handle,
               0, 0, SRCCOPY);
end;

procedure TMainForm.ClientWndProc(var Message: TMessage);
begin
  case Message.Msg of
    // Capture the WM_ERASEBKGND messages and perform the client area drawing
    WM_ERASEBKGND:
      begin
        CallWindowProc(FOldClientProc, ClientHandle, Message.Msg,
Message.wParam,
          Message.lParam);
        FDrawDC := TWMEraseBkGnd(Message).DC;
        if mmiStretch.Checked then
          DrawStretched
        else if mmiCenter.Checked then
          DrawCentered
        else DrawTiled;
```

```
        Message.Result := 1;
      end;
    { Capture the scrolling messages and ensure the client area
      is redrawn by calling InvalidateRect }
    WM_VSCROLL, WM_HSCROLL:
      begin
        Message.Result := CallWindowProc(FOldClientProc, ClientHandle,
Message.Msg,
          Message.wParam, Message.lParam);
        InvalidateRect(ClientHandle, nil, True);
      end;
    else
    // By Default, call the original window procedure
      Message.Result := CallWindowProc(FOldClientProc, ClientHandle,
Message.Msg,
        Message.wParam, Message.lParam);
  end; { case }
end;

procedure TMainForm.CreateMDIChild(const Name: string);
var
  MdiChild: TMDIChildForm;
begin
  MdiChild := TMDIChildForm.Create(Application);
  MdiChild.Caption := Name;
end;

procedure TMainForm.mmiNewClick(Sender: TObject);
begin
  CreateMDIChild('NONAME' + IntToStr(MDIChildCount + 1));
end;

procedure TMainForm.mmiCloseClick(Sender: TObject);
begin
  if ActiveMDIChild <> nil then
    ActiveMDIChild.Close;
end;

procedure TMainForm.mmiExitClick(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.mmiTileClick(Sender: TObject);
begin
  mmiTile.Checked := false;
```

continues

LISTING 16.7 Continued

```
  mmiCenter.Checked := False;
  mmiStretch.Checked := False;
  { Set the Checked property for the menu item which invoked }
  { this event handler to Checked                            }
  if Sender is TMenuItem then
    TMenuItem(Sender).Checked := not TMenuItem(Sender).Checked;
  { Redraw the client area of the form }
  InvalidateRect(ClientHandle, nil, True);
end;

end.
```

To paint the image to the client window of the MDI application, you must use a technique called *subclassing.* Subclassing is discussed in Chapter 5, "Understanding Messages." To subclass the client window, you must store the client window's original window procedure so that you can call it. You must also have a pointer to the new window procedure. The form variable `FOldClientProc` stores the original window procedure, and the variable `FNewClientProc` points to the new window procedure.

The procedure `ClientWndProc()` is the procedure to which `FNewClientProc` points. Actually, because `ClientWndProc()` is a method of `TMainForm`, you must use the `MakeObjectInstance()` function to return a pointer to a window procedure created from the method `MakeObjectInstance()`, as discussed in Chapter 13, "Hard-core Techniques."

The `TMainForm.CreateWnd()` method was overridden when the main form's client window was subclassed by using the `GetWindowLong()` and `SetWindowLong()` Win32 API functions. `ClientWndProc()` is the new window procedure.

`TMainForm` contains three private methods: `DrawCentered()`, `DrawTiled()`, and `DrawStretched()`. Each of these methods uses Win32 API functions to perform the GDI drawing routines to paint the bitmap. Win32 API functions are used because the client window's device context isn't encapsulated by `TCanvas`, so you can't normally use the built-in Delphi 5 methods. Actually, it's possible to assign the device context to a `TCanvas.Handle` property. You would have to instantiate a `TCanvas` instance in order to do this, but it is possible.

You must capture three messages to perform the background drawing: `WM_ERASEBKGND`, `WM_VSCROLL`, and `WM_HSCROLL`. The `WM_ERASEBKGND` message is sent to a window when it's to be erased. This is an opportune time to perform the specialized drawing of the image. In the procedure, you determine which drawing procedure to call based on which menu item is selected. The `WM_VSCROLL` and `WM_HSCROLL` messages are captured to ensure that the background image

is properly drawn when the user scrolls the main form. Finally, all other messages are sent to the original window procedure with this statement:

```
Message.Result := CallWindowProc(FOldClientProc, ClientHandle, Message.Msg,
        Message.wParam, Message.lParam);
```

This example not only demonstrates how you can visually enhance your applications; it also shows how you can perform API-level development with techniques not provided by the VCL.

Creating a Hidden MDI Child Form

Delphi 5 returns an error if you ever attempt to hide an MDI child form using a statement such as this one:

```
ChildForm.Hide;
```

The error indicates that hiding an MDI child form is not allowed. The reason for this is because the Delphi developers found that in the Windows implementation of MDI, hiding MDI child forms corrupts the z-order of the child windows. Unless you're extremely careful about when you use such a technique, trying to hide an MDI child form can wreak havoc with your application. Nevertheless, you might have the need to hide a child form. There are two ways in which you can hide MDI child forms. Just be aware of the anomaly and use these techniques with caution.

One way to hide an MDI child form is to prevent the client window from drawing the child form altogether. Do this by using the `LockWindowUpdate()` Win32 API function to disable drawing to the MDI client window. This technique is useful if you want to create an MDI child form but don't want to show that form to the user unless some process has completed successfully. For example, such a process might be a database query; if the process fails, you might want to free the form. Unless you use some method to hide the form, you'll see a flicker on the screen as the form is created before you have an opportunity to destroy it. The `LockWindowUpdate()` function disables drawing to a window's canvas. Only one window can be locked at any given time. Passing `0` to `LockWindowUpdate` reenables drawing to the window's canvas.

The other method of hiding an MDI child form is to actually hide the child form by using the Win32 API function `ShowWindow()`. You hide the form by specifying the `SW_HIDE` flag along with the function. You must then use the `SetWindowPos()` function to restore the child window. You can use this technique to hide the MDI child form if it's already created and displayed to the user.

Listing 16.8 illustrates the techniques just described and is the main form for the project `MdiHide.dpr` on the CD.

LISTING 16.8 A Unit Showing MDI Child Form–Hiding Techniques

```
unit MainFrm;

interface

uses Windows, SysUtils, Classes, Graphics, Forms, Controls, Menus,
  StdCtrls, Dialogs, Buttons, Messages, ExtCtrls, ComCtrls, MdiChildFrm;

type
  TMainForm = class(TForm)
    mmMain: TMainMenu;
    mmiFile: TMenuItem;
    mmiNew: TMenuItem;
    mmiClose: TMenuItem;
    mmiWindow: TMenuItem;
    N1: TMenuItem;
    mmiExit: TMenuItem;
    mmiHide: TMenuItem;
    mmiShow: TMenuItem;
    mmiHideForm: TMenuItem;
    procedure mmiNewClick(Sender: TObject);
    procedure mmiCloseClick(Sender: TObject);
    procedure mmiExitClick(Sender: TObject);
    procedure mmiHideClick(Sender: TObject);
    procedure mmiShowClick(Sender: TObject);
    procedure mmiHideFormClick(Sender: TObject);
  private
    procedure CreateMDIChild(const Name: string);
  public
    HideForm: TMDIChildForm;
    Hidden: Boolean;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.CreateMDIChild(const Name: string);
var
  MdiChild: TMDIChildForm;
begin
  MdiChild := TMDIChildForm.Create(Application);
  MdiChild.Caption := Name;
```

```
end;

procedure TMainForm.mmiNewClick(Sender: TObject);
begin
  CreateMDIChild('NONAME' + IntToStr(MDIChildCount + 1));
end;

procedure TMainForm.mmiCloseClick(Sender: TObject);
begin
  if ActiveMDIChild <> nil then
    ActiveMDIChild.Close;
end;

procedure TMainForm.mmiExitClick(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.mmiHideClick(Sender: TObject);
begin
  if Assigned(HideForm) then
    ShowWindow(HideForm.Handle, SW_HIDE);
  Hidden := True;
end;

procedure TMainForm.mmiShowClick(Sender: TObject);
begin
  if Assigned(HideForm) then
    SetWindowPos(HideForm.handle, HWND_TOP, 0, 0, 0, 0, SWP_NOSIZE
       or SWP_NOMOVE or SWP_SHOWWINDOW);
  Hidden := False;
end;

procedure TMainForm.mmiHideFormClick(Sender: TObject);
begin
  if not Assigned(HideForm) then
  begin
    if MessageDlg('Create Hidden?', mtConfirmation, [mbYes, mbNo], 0) = mrYes
then
    begin
       LockWindowUpdate(Handle);
      try
        HideForm := TMDIChildForm.Create(Application);
        HideForm.Caption := 'HideForm';
        ShowMessage('Form created and hidden. Press OK to show form');
      finally
```

continues

LISTING 16.8 Continued

```
        LockWindowUpdate(0);
      end;
    end
    else begin
      HideForm := TMDIChildForm.Create(Application);
      HideForm.Caption := 'HideForm';
    end;
  end
  else if not Hidden then
      HideForm.SetFocus;
end;

end.
```

The project is a simple MDI application. The event handler `mmiHideFormClick()` creates a child form that can either be created and hidden or hidden by the user after it's displayed.

When `mmiHideFormClick()` is invoked, it checks whether an instance of `THideForm` has been created. If so, it displays only the `THideForm` instance, provided that it has not been hidden by the user. If there is no instance of `THideForm` present, the user is prompted whether it should be created and hidden. If the user responds affirmatively, drawing to the client window is disabled before the form is created. If drawing to the client window is not disabled, the form is displayed as it's created. The user is then shown a message box indicating that the form is created. When the user closes the message box, drawing to the client window is reenabled and the child form is displayed by forcing the client window to repaint itself. You can replace the message box telling the user that the form is created with some lengthy process that requires the child form to be created but not displayed. If the user chooses not to create the form as hidden, it's created normally.

The second method used to hide the form after it has already been displayed calls the Win32 API function `ShowWindow()` and passes the child form's handle and the `SW_HIDE` flag. This effectively hides the form. To redisplay the form, call the Win32 API function `SetWindowPos()`, using the child form's handle and the flags specified in the listing. `SetWindowPos()` is used to change a window's size, position, or z-order. In this example, `SetWindowPos()` is used to redisplay the hidden window by setting its z-order; in this case, the z-order of the hidden form is set to be the top window by specifying the `HWND_TOP` flag.

Minimizing, Maximizing, and Restoring All MDI Child Windows

Often, you need to perform a task across all active MDI forms in the project. Changing the form's `WindowState` property is a typical example of a process to be performed on every

instance of an MDI child form. This task is quite simple and only requires that you walk through the forms using the main form's `MDIChildren` array property. The main form's `MDIChildren` property holds the number of active MDI child forms. Listing 16.9 shows the event handlers that minimize, maximize, and restore all MDI child windows in an application. This project can be found on the CD as the `Min_Max.dpr` project.

LISTING 16.9 Minimizing, Maximizing, and Restoring All MDI Child Forms

```
unit MainFrm;

interface

uses Windows, SysUtils, Classes, Graphics, Forms, Controls, Menus,
  StdCtrls, Dialogs, Buttons, Messages, ExtCtrls, ComCtrls;

type
  TMainForm = class(TForm)
    MainMenu1: TMainMenu;
    mmiFile: TMenuItem;
    mmiNew: TMenuItem;
    mmiClose: TMenuItem;
    mmiWindow: TMenuItem;
    N1: TMenuItem;
    mmiExit: TMenuItem;
    mmiMinimizeAll: TMenuItem;
    mmiMaximizeAll: TMenuItem;
    mmiRestoreAll: TMenuItem;
    procedure mmiNewClick(Sender: TObject);
    procedure mmiCloseClick(Sender: TObject);
    procedure mmiExitClick(Sender: TObject);
    procedure mmiMinimizeAllClick(Sender: TObject);
    procedure mmiMaximizeAllClick(Sender: TObject);
    procedure mmiRestoreAllClick(Sender: TObject);
  private
    { Private declarations }
    procedure CreateMDIChild(const Name: string);
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;

implementation
uses MdiChildFrm;
```

continues

LISTING 16.9 Continued

```
{$R *.DFM}

procedure TMainForm.CreateMDIChild(const Name: string);
var
  Child: TMDIChildForm;
begin
  Child := TMDIChildForm.Create(Application);
  Child.Caption := Name;
end;

procedure TMainForm.mmiNewClick(Sender: TObject);
begin
  CreateMDIChild('NONAME' + IntToStr(MDIChildCount + 1));
end;

procedure TMainForm.mmiCloseClick(Sender: TObject);
begin
  if ActiveMDIChild <> nil then
    ActiveMDIChild.Close;
end;

procedure TMainForm.mmiExitClick(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.mmiMinimizeAllClick(Sender: TObject);
var
  i: integer;
begin
  for i := MDIChildCount - 1 downto 0 do
    MDIChildren[i].WindowState := wsMinimized;
end;

procedure TMainForm.mmiMaximizeAllClick(Sender: TObject);
var
  i: integer;
begin
  for i := 0 to MDIChildCount - 1 do
    MDIChildren[i].WindowState := wsMaximized;
end;

procedure TMainForm.mmiRestoreAllClick(Sender: TObject);
var
```

```
  i: integer;
begin
  for i := 0 to MDIChildCount - 1 do
    MDIChildren[i].WindowState := wsNormal;
end;

end.
```

Summary

This chapter showed you how to build MDI applications in Delphi 5. You also learned some advanced techniques specific to MDI applications. With the foundation you received in this chapter, you should be well on your way to creating professional-looking MDI applications.

16

MDI APPLICATIONS

16

MDI APPLICATIONS