Printing in Delphi 5

CHAPTER

10

IN THIS CHAPTER

Printing in Windows has been the bane of many a Windows programmer. However, don't be discouraged; Delphi simplifies most of what you need to know for printing. You can write simple printing routines to output text or bitmapped images with little effort. For more complex printing, a few concepts and techniques are all you really need to enable you to perform any type of custom printing. When you have that, printing isn't so difficult.

NOTE

You'll find a set of reporting components by QuSoft on the QReport page of the Component Palette. The documentation for this tool is located in the help file `QuickRpt.hlp`.

QuSoft's tools are suitable for applications that generate complex reports. However, they limit you from getting to the nuts and bolts of printing at the source-code level, where you have more control over what gets printed. This chapter doesn't cover QuickReports; instead, it covers creating your own reports in Delphi.

Delphi's `TPrinter` object, which encapsulates the Windows printing engine, does a great deal for you that you would otherwise have to handle yourself.

This chapter teaches you how to perform a whole range of printing operations by using `TPrinter`. You learn the simple tasks that Delphi has made much easier for generating printouts. You also learn the techniques for creating advanced printing routines that should start you on your way to becoming a printing guru.

The TPrinter Object

The `TPrinter` object encapsulates the Windows printing interface, making most of the printing management invisible to you. `TPrinter`'s methods and properties enable you to print onto its canvas as though you were drawing your output to a form's surface. The function `Printer()` returns a global `TPrinter` instance the first time it's called. `TPrinter`'s properties and methods are listed in Tables 10.1 and 10.2.

TABLE 10.1 `TPrinter` Properties

Property	*Purpose*
`Aborted`	Boolean variable that determines whether the user has aborted the print job.
`Canvas`	The printing surface for the current page.
`Fonts`	Contains a list of fonts supported by the printer.
`Handle`	A unique number representing the printer's device handle. See the sidebar "Handles" in Chapter 20, "Key Elements of the Visual Component Library."

Property	*Purpose*
`Orientation`	Determines horizontal (`poLandScape`) or vertical (`poPortrait`) printing.
`PageHeight`	Height, in pixels, of the printed page's surface.
`PageNumber`	Indicates the page being printed. This is incremented with each subsequent call to `TPrinter.NewPage()`.
`PageWidth`	Width, in pixels, of the printed page's surface.
`PrinterIndex`	Indicates the selected printer from the available printers on the user's system.
`Printers`	A list of the available printers on the system.
`Printing`	Determines whether a print job is printing.
`Title`	Text appearing in the Print Manager and on networked pages.

TABLE 10.2 `TPrinter` Methods

Method	*Purpose*
`Abort`	Terminates a print job.
`BeginDoc`	Begins a print job.
`EndDoc`	Ends a print job. (`EndDoc` ends a print job when printing is finished; `Abort` can terminate the job before printing is complete.)
`GetPrinter`	Retrieves the current printer.
`NewPage`	Forces the printer to start printing on a new page and increments the `PageCount` property.
`SetPrinter`	Specifies a printer as a current printer.

TPrinter.Canvas

`TPrinter.Canvas` is much like the canvas for your form; it represents the drawing surface on which text and graphics are drawn. The difference is that `TPrinter.Canvas` represents the drawing surface for your printed output as opposed to your screen. Most of the routines you use to draw text, to draw shapes, and to display images are used in the same manner for printed output. When printing, however, you must take into account some differences:

- Drawing to the screen is *dynamic*—you can erase what you've placed on the screen's output. Drawing to the printer isn't so flexible. What's drawn to the `TPrinter.Canvas` is printed to the printer.
- Drawing text or graphics to the screen is nearly instantaneous, whereas drawing to the printer is slow, even on some high-performance laser printers. You therefore must allow

users to abort a print job either by using an Abort dialog box or by some other method that enables them to terminate the print job.

- Because your users are running Windows, you can assume that their display supports graphics output. However, you can't assume the same for their printers. Different printers have different capabilities. Some printers may be high-resolution printers; other printers may be very low resolution and may not support graphics printing at all. You must take this into account in your printing routines.
- You'll never see an error message like this:

  ```
  Display ran out of screen space,
  please insert more screen space into your display.
  ```

  But you can bet that you'll see an error telling you that the printer ran out of paper. Windows NT/2000 and Windows 95/98 both provide error handling when this occurs. However, you should provide a way for the user to cancel the printout when this occurs.
- Text and graphics on your screen don't look the same on hard copy. Printers and displays have very different resolutions. That 300×300 bitmap might look spectacular on a 640×480 display, but it's a mere 1×1-inch square blob on your 300 dpi (dots per inch) laser printer. You're responsible for making adjustments to your drawing routines so that your users won't need a magnifying glass to read their printed output.

Simple Printing

In many cases, you want to send a stream of text to your printer without any regard for special formatting or placement of the text. Delphi facilitates simple printing, as the following sections illustrate.

Printing the Contents of a TMemo Component

Printing lines of text is actually quite simple using the `AssignPrn()` procedure. The `AssignPrn()` procedure enables you to assign a text file variable to the current printer. It's used with the `Rewrite()` and `CloseFile()` procedures. The following lines of code illustrate this syntax:

```
var
  f: TextFile;
begin
  AssignPrn(f);
  try
    Rewrite(f);
    writeln(f, 'Print the output');
  finally
```

```
    CloseFile(f);
  end;
end;
```

Printing a line of text to the printer is the same as printing a line of text to a file. You use this syntax:

```
writeln(f, 'This is my line of text');
```

In Chapter 16, "MDI Applications," you add menu options for printing the contents of the `TMdiEditForm` form. Listing 10.1 shows you how to print the contents from `TMdiEditForm`. You'll use this same technique for printing text from just about any source.

LISTING 10.1 Printing Code for `TMdiEditForm`

```
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;
```

Notice that the memo's font also was assigned to the `Printer`'s font, causing the output to print with the same font as `memMainMemo`.

CAUTION

Be aware that the printer will print with the font specified by `Printer.Font` only if the printer supports that font. Otherwise, the printer will use a font that approximates the characteristics of the specified font.

Printing a Bitmap

Printing a bitmap is simple as well. The `MdiApp` example in Chapter 16, "MDI Applications," shows how to print the contents of a bitmap in `TMdiBmpForm`. This event handler is shown in Listing 10.2.

LISTING 10.2 Printing Code for `TMdiBmpForm`

```
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;
```

Only three lines of code are needed to print the bitmap using the `TCanvas.StretchDraw()` method. This vast simplification of printing a bitmap is made possible by the fact that since Delphi 3, bitmaps are in DIB format by default, and DIBs are what the printer driver requires. If you happen to have a handle to a bitmap that isn't in DIB format, you can copy (`Assign`) it into a temporary `TBitmap`, force the temporary bitmap into DIB format by assigning `bmDIB` to the `TBitmap.HandleType` property, and then print from the new DIB.

NOTE

One of the keys to printing is to be able to print images as they appear onscreen at approximately the same size. A 3×3-inch image on a 640×480 pixel screen uses fewer pixels than it would on a 300 dpi printer, for example. Therefore, stretch the image to `TPrinter`'s canvas as was done in the example in the call to `StretchDIBits()`. Another technique is to draw the image using a different mapping mode, as described in Chapter 8, "Graphics Programming with GDI and Fonts." Keep in mind that some older printers may not support the stretching of images. You can obtain valuable information about the printer's capabilities by using the Win32 API function `GetDeviceCaps()`.

Printing Rich Text–Formatted Data

Printing the contents of a `TRichEdit` component is a matter of one method call. The following code shows how to do this (this is also the code for printing `TMdiRtfForm` in the `MdiApp` example in Chapter 16, "MDI Applications"):

```
procedure TMdiRtfForm.mmiPrintClick(Sender: TObject);
begin
  inherited;
  reMain.Print(Caption);
end;
```

Printing a Form

Conceptually, printing a form can be one of the more difficult tasks to perform. However, this task has been simplified greatly thanks to VCL's `Print()` method of `TForm`. The following one-line procedure prints your form's client areas as well as all components residing in the client area:

```
procedure TForm1.PrintMyForm(Sender: TObject);
begin
  Print;
end;
```

NOTE

Printing your form is a quick-and-dirty way to print graphical output. However, only what's visible onscreen will be printed, due to Windows' clipping. Also, the bitmap is created at screen pixel density and then stretched to printer resolution. Text on the form is not drawn at printer resolution; it's drawn at screen resolution and stretched, so overall the form will be noticeably jagged and blocky. You must use more elaborate techniques to print complex graphics; these techniques are discussed later in this chapter.

Advanced Printing

Often you need to print something very specific that isn't facilitated by the development tool you're using or a third-party reporting tool. In this case, you need to perform the low-level printing tasks yourself. The next several sections show you how to write such printing routines and present a methodology you can apply to all your printing tasks.

NOTE

Although this section covers printing, you should know that at the time of this writing, several third-party printing components are available that should handle most of your printing needs. You'll find demos of some of these tools on the CD with this book.

Printing a Columnar Report

Many applications, particularly those using databases, print some type of report. One common report style is the columnar report.

The next project prints a columnar report from one of the tables in Delphi's demo directories. Each page contains a header, column titles, and then the record list. Each subsequent page also has the header and column titles preceding the record list.

Figure 10.1 shows the main form for this project. The `TEdit`/`TUpDown` pairs enable the user to specify the column widths in tenths of inches. By using the `TUpDown` components, you can specify minimum and maximum values. The `TEdit1` control, `edtHeaderFont`, contains a header that can be printed using a font that differs from the one used for the rest of the report.

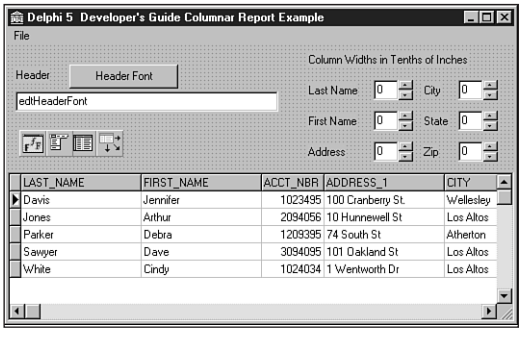

FIGURE 10.1
Columnar report main form.

Listing 10.3 shows the source code for the project. The `mmiPrintClick()` event handler basically performs the following steps:

1. Initiates a print job.
2. Prints a header.
3. Prints column names.

4. Prints a page.
5. Continues steps 2, 3, and 4 until printing finishes.
6. Ends the print job.

LISTING 10.3 Columnar Report Demo

```
unit MainFrm;
interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Grids, DBGrids, DB, DBTables, Menus, StdCtrls, Spin,
  Gauges, ExtCtrls, ComCtrls;

type
  TMainForm = class(TForm)
   { components not included in listing,
     please refer to CD source }
  procedure mmiPrintClick(Sender: TObject);
    procedure btnHeaderFontClick(Sender: TObject);
  private
    PixelsInInchx: integer;
    LineHeight: Integer;
   { Keeps track of vertical space in pixels, printed on a page }
    AmountPrinted: integer;
   { Number of pixels in 1/10 of an inch. This is used for line spacing }
    TenthsOfInchPixelsY: integer;
    procedure PrintLine(Items: TStringList);
    procedure PrintHeader;
    procedure PrintColumnNames;
  end;

var
  MainForm: TMainForm;

implementation
uses printers, AbortFrm;

{$R *.DFM}

procedure TMainForm.PrintLine(Items: TStringList);
var
  OutRect: TRect;
  Inches: double;
  i: integer;
```

continues

LISTING 10.3 Continued

```
begin
  // First position the print rect on the print canvas
  OutRect.Left := 0;
  OutRect.Top := AmountPrinted;
  OutRect.Bottom := OutRect.Top + LineHeight;
  With Printer.Canvas do
    for i := 0 to Items.Count - 1 do
    begin
      Inches := longint(Items.Objects[i]) * 0.1;
      // Determine Right edge
      OutRect.Right := OutRect.Left + round(PixelsInInchx*Inches);
      if not Printer.Aborted then
       // Print the line
        TextRect(OutRect, OutRect.Left, OutRect.Top, Items[i]);
      // Adjust right edge
      OutRect.Left := OutRect.Right;
    end;
 { As each line prints, AmountPrinted must increase to reflect how
   much of a page has been printed on based on the line height. }
  AmountPrinted := AmountPrinted + TenthsOfInchPixelsY*2;
end;

procedure TMainForm.PrintHeader;
var
  SaveFont: TFont;
begin
 { Save the current printer's font, then set a new print font based
  on the selection for Edit1 }
  SaveFont := TFont.Create;
  try
    Savefont.Assign(Printer.Canvas.Font);
    Printer.Canvas.Font.Assign(edtHeaderFont.Font);
    // First print out the Header
    with Printer do
    begin
      if not Printer.Aborted then
        Canvas.TextOut((PageWidth div 2)-(Canvas.TextWidth(edtHeaderFont.Text)
                        div 2),0, edtHeaderFont.Text);
     // Increment AmountPrinted by the LineHeight
      AmountPrinted := AmountPrinted + LineHeight+TenthsOfInchPixelsY;
    end;
    // Restore the old font to the Printer's Canvas property
    Printer.Canvas.Font.Assign(SaveFont);
  finally
    SaveFont.Free;
```

```
  end;
end;

procedure TMainForm.PrintColumnNames;
var
  ColNames: TStringList;
begin
  { Create a TStringList to hold the column names and the
    positions where the width of each column is based on values
    in the TEdit controls. }
  ColNames := TStringList.Create;
  try
    // Print the column headers using a bold/underline style
    Printer.Canvas.Font.Style := [fsBold, fsUnderline];

    with ColNames do
    begin
      // Store the column headers and widths in the TStringList object
      AddObject('LAST NAME',  pointer(StrToInt(edtLastName.Text)));
      AddObject('FIRST NAME', pointer(StrToInt(edtFirstName.Text)));
      AddObject('ADDRESS',    pointer(StrToInt(edtAddress.Text)));
      AddObject('CITY',       pointer(StrToInt(edtCity.Text)));
      AddObject('STATE',      pointer(StrToInt(edtState.Text)));
      AddObject('ZIP',        pointer(StrToInt(edtZip.Text)));
    end;

    PrintLine(ColNames);
    Printer.Canvas.Font.Style := [];
  finally
    ColNames.Free;   // Free the column name TStringList instance
  end;
end;

procedure TMainForm.mmiPrintClick(Sender: TObject);
var
  Items: TStringList;
begin
  { Create a TStringList instance to hold the fields and the widths
    of the columns in which they'll be drawn based on the entries in
    the edit controls }
  Items := TStringList.Create;
  try
    // Determine pixels per inch horizontally
    PixelsInInchx := GetDeviceCaps(Printer.Handle, LOGPIXELSX);
    TenthsOfInchPixelsY := GetDeviceCaps(Printer.Handle,
                           LOGPIXELSY) div 10;
```

continues

LISTING 10.3 Continued

```
AmountPrinted := 0;
MainForm.Enabled := False; // Disable the parent form
try
  Printer.BeginDoc;
  AbortForm.Show;
  Application.ProcessMessages;
  { Calculate the line height based on text height using the
    currently rendered font }
  LineHeight := Printer.Canvas.TextHeight('X')+TenthsOfInchPixelsY;
  if edtHeaderFont.Text <> '' then
    PrintHeader;
  PrintColumnNames;
  tblClients.First;
  { Store each field value in the TStringList as well as its
    column width }
  while (not tblClients.Eof) or Printer.Aborted do
  begin

    Application.ProcessMessages;
    with Items do
    begin
      AddObject(tblClients.FieldByName('LAST_NAME').AsString,
                  pointer(StrToInt(edtLastName.Text)));
      AddObject(tblClients.FieldByName('FIRST_NAME').AsString,
                  pointer(StrToInt(edtFirstName.Text)));
      AddObject(tblClients.FieldByName('ADDRESS_1').AsString,
                  pointer(StrToInt(edtAddress.Text)));
      AddObject(tblClients.FieldByName('CITY').AsString,
                  pointer(StrToInt(edtCity.Text)));
      AddObject(tblClients.FieldByName('STATE').AsString,
                  pointer(StrToInt(edtState.Text)));
      AddObject(tblClients.FieldByName('ZIP').AsString,
                  pointer(StrToInt(edtZip.Text)));
    end;
    PrintLine(Items);
    { Force print job to begin a new page if printed output has
      exceeded page height }
    if AmountPrinted + LineHeight > Printer.PageHeight then
    begin
      AmountPrinted := 0;
      if not Printer.Aborted then
        Printer.NewPage;
      PrintHeader;
      PrintColumnNames;
```

```
        end;
        Items.Clear;
        tblClients.Next;
      end;
      AbortForm.Hide;
      if not Printer.Aborted then
        Printer.EndDoc;
    finally
      MainForm.Enabled := True;
    end;
  finally
    Items.Free;
  end;
end;

procedure TMainForm.btnHeaderFontClick(Sender: TObject);
begin
 { Assign the font selected with FontDialog1 to Edit1. }
  FontDialog.Font.Assign(edtHeaderFont.Font);
  if FontDialog.Execute then
    edtHeaderFont.Font.Assign(FontDialog.Font);
end;

end.
```

`mmiPrintClick()` first creates a `TStringList` instance to hold the strings for a line to be printed. Then the number of pixels per inch along the vertical axis is determined in `PixelsPerInchX`, which is used to calculate column widths. `TenthsOfInchPixelsY` is used to space each line by 0.1 inch. `AmountPrinted` holds the total amount of pixels along the printed surface's vertical axis for each line printed. This is required to determine whether to start a new page when `AmountPrinted` exceeds `Printer.PageHeight`.

If a header exists in `edtHeaderFont.Text`, it's printed in `PrintHeader()`. `PrintColumnNames()` prints the names of the columns for each field to be printed. (These two procedures are discussed later in this section.) Finally, the table's records are printed.

The following loop increments through `tblClients` records and prints selected fields within each of the records:

```
while (not tblClients.Eof) or Printer.Aborted do begin
```

Within the loop, the field values are added to the `TStringList` using the `AddObject()` method. Here, you store both the string and the column width. The column width is added to the `Items.Objects` array property. `Items` is then passed to the `PrintLine()` procedure, which prints the strings in a columnar format.

In much of the previous code, you saw references to `Printer.Aborted`. This is a test to determine whether the user has aborted the print job, which is covered in the next section.

TIP

The `TStrings` and `TStringList`'s `Objects` array properties are a convenient place to store integer values. Using `AddObject()` or `InsertObject()`, you can hold any number up to `MaxLongInt`. Because `AddObject()` expects a `TObject` reference as its second parameter, you must typecast that parameter as a pointer, as shown in the following code:

```
MyList.AddObject('SomeString', pointer(SomeInteger));
```

To retrieve the value, use a `Longint` typecast:

```
MyInteger := Longint(MyList.Objects[Index]);
```

The event handler then determines whether printing a new line will exceed the page height:

```
if AmountPrinted + LineHeight > Printer.PageHeight then
```

If this evaluates to `True`, `AmountPrinted` is set back to `0`, `Printer.NewPage` is invoked to print a new page, and the header and column names are printed again. `Printer.EndDoc` is called to end the print job after the `tblClients` records have printed.

The `PrintHeader()` procedure prints the header centered at the top of the report using `edtHeaderFont.Text` and `edtHeaderFont.Font`. `AmountPrinted` is then incremented and `Printer`'s font is restored to its original style.

As the name implies, `PrintColumnNames()` prints the column names of the report. In this method, names are added to a `TStringList` object, `ColNames`, which then is passed to `PrintLine()`. Notice that the column names are printed in a bold, underlined font. Setting `Printer.Canvas.Font` accordingly does this.

The `PrintLine()` procedure takes a `TStringList` argument called `Items` and prints each string in `Items` on a single line in a columnar manner. The variable `OutRect` holds values for a binding rectangle at a location on `Printer`'s canvas to which the text is drawn. `OutRect` is passed to `TextRect()`, along with the text to draw. By multiplying `Items.Object[i]` by 0.1, `OutRect.Right`'s value is obtained because `Items.Objects[i]` is in tenths of inches. Inside the `for` loop, `OutRect` is recalculated along the same X-axis to position it to the next column and draw the next text value. Finally, `AmountPrinted` is incremented by `LineHeight + TenthsOfInchPixelsY`.

Although this report is fully functional, you might consider extending it to include a footer, page numbers, and even margin settings.

Aborting the Printing Process

Earlier in this chapter, you learned that your users need a way to terminate printing after they've initiated it. The `TPrinter.Abort()` procedure and the `Aborted` property help you do this. The code in Listing 10.3 contains such logic. To add abort logic to your printing routines, your code must meet these three conditions:

- You must establish an event that, when activated, calls `Printer.Abort`, thus aborting the printing process.
- You must check for `TPrinter.Aborted = True` before calling any of `TPrinter`'s print functions, such as `TextOut()`, `NewPage()`, and `EndDoc()`.
- You must end your printing logic by checking the value of `TPrinter.Aborted` for `True`.

A simple Abort dialog box can satisfy the first condition. You used such a dialog box in the preceding example. This dialog box should contain a button that will invoke the abort process.

This button's event handler should simply call `TPrinter.Abort`, which terminates the print job and cancels any printing requests made to `TPrinter`.

In the unit `MainForm.pas`, examine the code to show `AbortForm` shortly after calling `TPrinter.Begindoc()`:

```
Printer.BeginDoc;
AbortForm.Show;
Application.ProcessMessages;
```

Because `AbortForm` is shown as a modeless dialog box, the call to `Application.ProcessMessages` ensures that it's drawn properly before any processing of the printing logic continues.

To satisfy the second condition, the test for `Printer.Aborted = True` is performed before calling any `TPrinter` methods. The `Aborted` property is set to `True` when the `Abort()` method is called from `AbortForm`. As an example, before you call `Printer.TextRect`, check for `Aborted = True`:

```
if not Printer.Aborted then
  TextRect(OutRect, OutRect.Left, OutRect.Top, Items[i]);
```

Also, you shouldn't call `EndDoc()` or any of `TPrinter.Canvas`'s drawing routines after calling `Abort()`, because the printer has been effectively closed.

To satisfy the third condition in this example, `while not Table.Eof` also checks whether the value of `Printer.Aborted` is `True`, which causes execution to jump out of the loop where the print logic is executed.

Printing Envelopes

The preceding example showed you a method for printing a columnar report. Although this technique was somewhat more complicated than sending a series of `writeln()` calls to the printer, it's still, for the most part, a line-by-line print. Printing envelopes introduces a few factors that complicate things a bit further and are common to most types of printing you'll do in Windows. First, the objects (items) you must print probably need to be positioned at some specific location on the printed surface. Second, the items' *metrics*, or units of measurement, can be completely different from those of the printer canvas. Taking these two factors into account, printing becomes much more than just printing a line and keeping track of how much print space you've used.

This envelope-printing example shows you a step-by-step process you can use to print just about anything. Keep in mind that everything drawn on the printer's canvas is drawn within some bounding rectangle on the canvas or to specific points on the printer canvas.

Printing in the Abstract

Think of the printing task in a more abstract sense for a moment. In all cases, two things are certain: You have a surface on which to print, and you have one or more elements to plot onto that surface. Take a look at Figure 10.2.

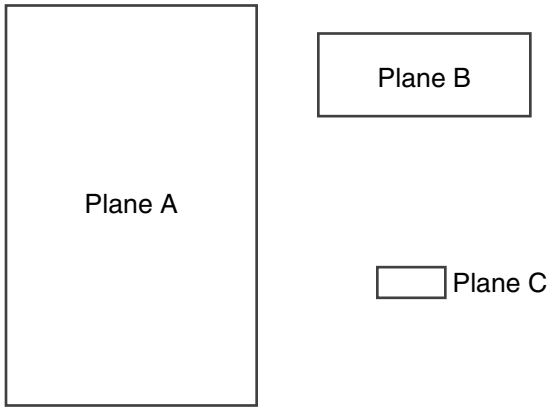

FIGURE 10.2
Three planes.

In Figure 10.2, Plane A is your destination surface. Planes B and C are the elements you want to superimpose (print) onto Plane A. Assume a coordinate system for each plane where the unit of measurement increases as you travel east along the X-axis and south along the Y-axis—that is, unless you live in Australia. Figure 10.3 depicts this coordinate system. The result of combining the planes is shown in Figure 10.4.

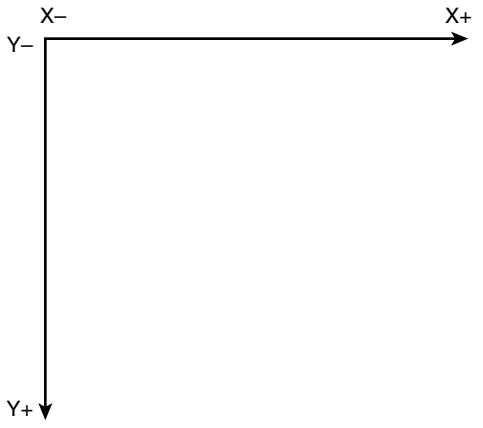

FIGURE 10.3
The Plane A, B, and C coordinate system.

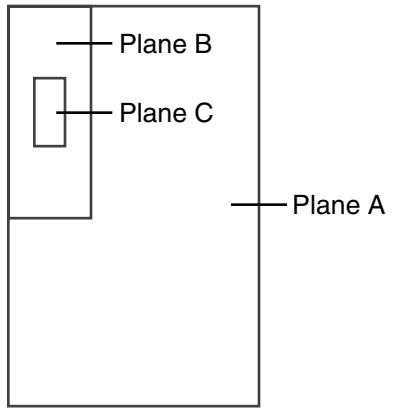

FIGURE 10.4
Planes B and C superimposed on Plane A.

Notice that Planes B and C were rotated by 90 degrees to achieve the final result. So far, this doesn't appear to be too bad. Given that your planes are measured using the same unit of measurement, you can easily draw out these rectangles to achieve the final result with some simple geometry. But what if they're not the same unit of measurement?

Suppose that Plane A represents a surface for which the measurements are given in pixels. Its dimensions are 2,550×3,300 pixels. Plane B is measured in inches: 6½×3¾ inches. Suppose

that you don't know the dimensions for Plane C; you do know, however, that it's measured in pixels, and you'll know its measurements later. These measurements are illustrated in Figure 10.5.

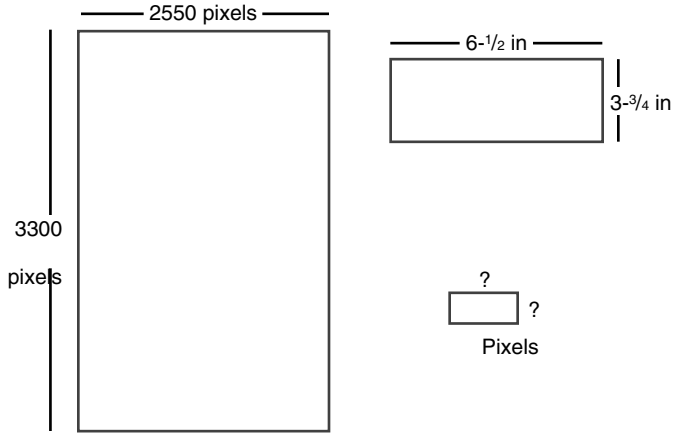

FIGURE 10.5
Plane measurements.

This abstraction illustrates the problem associated with printing. In fact, it illustrates the very task of printing an envelope. Plane A represents a printer's page size on a 300 dpi printer (at 300 dpi, 8½×11 inches equals 2,550×3,300 pixels). Plane B represents the envelope's size in inches, and Plane C represents the bounding rectangle for the text making up the address. Keep in mind, however, that this abstraction isn't tied to just envelopes. Planes B and C might represent `TImage` components measured in millimeters.

By looking at this task in its abstraction, you've achieved the first three steps to printing in Windows: Identify each element to print, identify the unit of measurement for the destination surface, and identify the units of measurement for each individual element to be plotted onto the destination surface.

Now consider another twist—literally. When you're printing an envelope in a vertical fashion, the text must rotate vertically.

A Step-by-Step Process for Printing

The following list summarizes the process you should follow when laying out your printed output in code:

1. Identify each element to be printed to the destination surface.
2. Identify the unit of measurement for the destination surface or printer canvas.

3. Identify the units of measurement for each individual element to be plotted onto the destination surface.
4. Decide on the common unit of measurement with which to perform all drawing routines. Almost always, this will be the printer canvas's units—pixels.
5. Write the translation routines to convert the other units of measurement to that of the common unit of measurement.
6. Write the routines to calculate the size for each element to print in the common unit of measurement. In Object Pascal, this can be represented by a `TPoint` structure. Keep in mind dependencies on other values. For example, the address's bounding rectangle is dependent on the envelope's position. Therefore, the envelope's data must be calculated first.
7. Write the routines to calculate the position of each element as it will appear on the printer canvas, based on the printer canvas's coordinate system and the sizes obtained from step 6. In Object Pascal, this can be represented by a `TRect` structure. Again, keep dependencies in mind.
8. Write your printing function, using the data gathered from the previous steps, to position items on the printed surface.

NOTE

Steps 5 and 6 can be achieved by using a technique of performing all drawing in a specific mapping mode. Mapping modes are discussed in Chapter 8, "Graphics Programming with GDI and Fonts."

Getting Down to Business

Given the step-by-step process, your task of printing an envelope should be much clearer. You'll see this in the envelope-printing project. The first step is to identify the elements to print or represent. The elements for the envelope example are the envelope, itself, and the address.

In this example, you learn how to print two standard envelope sizes: a size 10 and a size 6¾.

The following record holds the envelope sizes:

```
type

  TEnvelope = record
    Kind: string;   // Stores the envelope type's name
    Width: double;  // Holds the width of the envelope
    Height: double; // Holds the height of the envelope
  end;
```

```
const
  // This constant array stores envelope types
  EnvArray: array[1..2] of TEnvelope =
  ((Kind:'Size 10';Width:9.5;Height:4.125),     // 9-1/2 x 4-1/8
   (Kind:'Size 6-3/4';Width:6.5;Height:3.625)); // 6-1/2 x 3-3/4
```

Steps 2 and 3 are covered: You know that the destination surface is the `TPrinter.Canvas`, which is represented in pixels. The envelopes are represented in inches, and the address is represented in pixels. Step 4 requires you to select a common unit of measurement. For this project, you use pixels as the common unit of measurement.

For step 5, the only units you need to convert are from inches to pixels. The `GetDeviceCaps()` Win32 API function can return the amount of pixels per one inch along the horizontal and vertical axis for `Printer.Canvas`:

```
PixPerInX := GetDeviceCaps(Printer.Handle, LOGPIXELSX);
PixPerInY := GetDeviceCaps(Printer.Handle, LOGPIXELSY);
```

To convert the envelope's size to pixels, you just multiply the number of inches by `PixPerInX` or `PixPerInY` to get the horizontal or vertical measurement in pixels:

```
EnvelopeWidthInPixels := trunc(EnvelopeWidthValue * PixPerInX);
EnvelopeHeightInPixels := trunc(EnvelopeHeightValue * PixPerInY);
```

Because the envelope width or height can be a fractional value, it's necessary to use the `Trunc()` function to return the integer portion of the floating-point type rounded toward zero.

The sample project demonstrates how you would implement steps 6 and 7. The main form for this project is shown in Figure 10.6; Listing 10.4 shows the source code for the envelope-printing project.

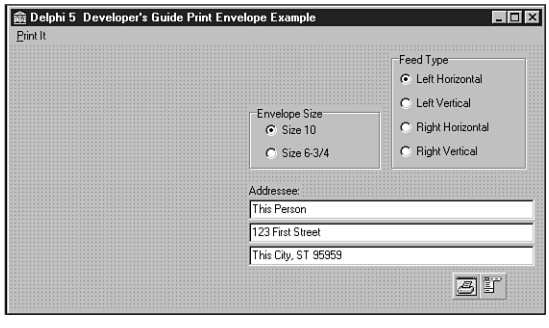

FIGURE 10.6
The main form for the envelope demo.

LISTING 10.4 Envelope Printing Demo

```
unit MainFrm;

interface

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

type

  TEnvelope = record
    Kind: string;   // Stores the envelope type's name
    Width: double;  // Holds the width of the envelope
    Height: double; // Holds the height of the envelope
  end;

const
  // This constant array stores envelope types
  EnvArray: array[1..2] of TEnvelope =
  ((Kind:'Size 10';Width:9.5;Height:4.125),     // 9-1/2 x 4-1/8
   (Kind:'Size 6-3/4';Width:6.5;Height:3.625)); // 6-1/2 x 3-3/4

type

  // This enumerated type represents printing positions.
  TFeedType = (epLHorz, epLVert, epRHorz, epRVert);

  TPrintPrevPanel = class(TPanel)
  public
    property Canvas; // Publicize the Canvas property
  end;

  TMainForm = class(TForm)
    gbEnvelopeSize: TGroupBox;
    rbSize10: TRadioButton;
    rbSize6: TRadioButton;
    mmMain: TMainMenu;
    mmiPrintIt: TMenuItem;
    lblAdressee: TLabel;
    edtName: TEdit;
    edtStreet: TEdit;
    edtCityState: TEdit;
    rgFeedType: TRadioGroup;
    PrintDialog: TPrintDialog;
```

continues

LISTING 10.4 Continued

```
    procedure FormCreate(Sender: TObject);
    procedure rgFeedTypeClick(Sender: TObject);
    procedure mmiPrintItClick(Sender: TObject);
  private
    PrintPrev: TPrintPrevPanel;  // Print  preview panel
    EnvSize: TPoint;     // Stores the envelope's size
    EnvPos: TRect;       // Stores the envelope's position
    ToAddrPos: TRect;    // Stores the address's position
    FeedType: TFeedType; // Stores the feed type from TEnvPosition
    function GetEnvelopeSize: TPoint;
    function GetEnvelopePos: TRect;
    function GetToAddrSize: TPoint;
    function GetToAddrPos: TRect;
    procedure DrawIt;
    procedure RotatePrintFont;
    procedure SetCopies(Copies: Integer);
  end;

var
  MainForm: TMainForm;

implementation
{$R *.DFM}

function TMainForm.GetEnvelopeSize: TPoint;
// Gets the envelope's size represented by a TPoint
var
  EnvW, EnvH: integer;
  PixPerInX,
  PixPerInY: integer;
begin
  // Pixels per inch along the horizontal axis
  PixPerInX := GetDeviceCaps(Printer.Handle, LOGPIXELSX);
  // Pixels per inch along the vertical axis
  PixPerInY := GetDeviceCaps(Printer.Handle, LOGPIXELSY);

  // Envelope size differs depending on the user's selection
  if RBSize10.Checked then
  begin
    EnvW := trunc(EnvArray[1].Width * PixPerInX);
    EnvH := trunc(EnvArray[1].Height * PixPerInY);
  end
  else begin
    EnvW := trunc(EnvArray[2].Width * PixPerInX);
```

```
    EnvH := trunc(EnvArray[2].Height * PixPerInY);
  end;

  // return Result as a TPoint record
  Result := Point(EnvW, EnvH)
end;

function TMainForm.GetEnvelopePos: TRect;
{ Returns the envelope's position relative to its feed type. This
  function requires that the variable EnvSize be initialized }
begin
  // Determine feed type based on user's selection.
  FeedType := TFeedType(rgFeedType.ItemIndex);

  { Return a TRect structure indicating the envelope's
    position as it is ejected from the printer. }
  case FeedType of
   epLHorz:
     Result := Rect(0, 0, EnvSize.X, EnvSize.Y);
   epLVert:
     Result := Rect(0, 0, EnvSize.Y, EnvSize.X);
   epRHorz:
     Result := Rect(Printer.PageWidth - EnvSize.X, 0,
➥Printer.PageWidth, EnvSize.Y);
   epRVert:
     Result := Rect(Printer.PageWidth - EnvSize.Y, 0,
➥Printer.PageWidth, EnvSize.X);
  end; // Case
end;

function MaxLn(V1, V2: Integer): Integer;
// Returns the larger of the two. If equal, returns the first
begin
  Result := V1;    // Default result to V1 }
  if V1 < V2 then
    Result := V2
end;

function TMainForm.GetToAddrSize: TPoint;
var
  TempPoint: TPoint;
begin
  // Calculate the size of the longest line using the MaxLn() function
  TempPoint.x := Printer.Canvas.TextWidth(edtName.Text);
  TempPoint.x := MaxLn(TempPoint.x, Printer.Canvas.TextWidth(edtStreet.Text));
  TempPoint.x := MaxLn(TempPoint.x,
```

continues

LISTING 10.4 Continued

```
     Printer.Canvas.TextWidth(edtCityState.Text))+10;

  // Calculate the height of all the address lines
  TempPoint.y := Printer.Canvas.TextHeight(edtName.Text)+
    ➥Printer.Canvas.TextHeight(edtStreet.Text)+
    ➥Printer.Canvas.TextHeight(edtCityState.Text)+10;
  Result := TempPoint;
end;

function TMainForm.GetToAddrPos: TRect;
// This function requires that EnvSize, and EnvPos be initialized
Var
  TempSize: TPoint;
  LT, RB: TPoint;
begin
  // Determine the size of the Address bounding rectangle
  TempSize := GetToAddrSize;
  { Calculate two points, one representing the Left Top (LT) position
    and one representing the Right Bottom (RB) position of the
    address's bounding rectangle. This depends on the FeedType }
  case FeedType of
    epLHorz:
      begin
        LT := Point((EnvSize.x div 2) - (TempSize.x div 2),
          ((EnvSize.y div 2) - (TempSize.y div 2)));
        RB := Point(LT.x + TempSize.x, LT.y + TempSize.Y);
      end;
    epLVert:
      begin
        LT := Point((EnvSize.y div 2) - (TempSize.y div 2),
          ((EnvSize.x div 2) - (TempSize.x div 2)));
        RB := Point(LT.x + TempSize.y, LT.y + TempSize.x);
      end;
    epRHorz:
    begin
      LT := Point((EnvSize.x div 2) - (TempSize.x div 2) + EnvPos.Left,
        ((EnvSize.y div 2) - (TempSize.y div 2)));
      RB := Point(LT.x + TempSize.x, LT.y + TempSize.Y);
    end;
    epRVert:
    begin
      LT := Point((EnvSize.y div 2) - (TempSize.y div 2) + EnvPos.Left,
        ((EnvSize.x div 2) - (TempSize.x div 2)));
      RB := Point(LT.x + TempSize.y, LT.y + TempSize.x);
    end;
```

```
  end; // End Case

  Result := Rect(LT.x, LT.y, RB.x, RB.y);
end;

procedure TMainForm.DrawIt;
// This procedure assumes that EnvPos and EnvSize have been initialized
begin
  PrintPrev.Invalidate;   // Erase contents of Panel
  PrintPrev.Update;
  // Set the mapping mode for the panel to MM_ISOTROPIC
  SetMapMode(PrintPrev.Canvas.Handle, MM_ISOTROPIC);
  // Set the TPanel's extent to match that of the printer boundaries.
  SetWindowExtEx(PrintPrev.Canvas.Handle,
      Printer.PageWidth, Printer.PageHeight, nil);
  // Set the viewport extent to that of the PrintPrev TPanel size.
  SetViewPortExtEx(PrintPrev.Canvas.Handle,
      PrintPrev.Width, PrintPrev.Height, nil);
  // Set the origin to the position at 0, 0
  SetViewportOrgEx(PrintPrev.Canvas.Handle, 0, 0, nil);
  PrintPrev.Brush.Style := bsSolid;

  with EnvPos do
   // Draw a rectangle to represent the envelope
    PrintPrev.Canvas.Rectangle(Left, Top, Right, Bottom);

  with ToAddrPos, PrintPrev.Canvas do
    case FeedType of
    epLHorz, epRHorz:
      begin
        Rectangle(Left, Top, Right, Top+2);
        Rectangle(Left, Top+(Bottom-Top) div 2, Right,
➥Top+(Bottom-Top) div 2+2);
        Rectangle(Left, Bottom, Right, Bottom+2);
      end;
    epLVert, epRVert:
      begin
        Rectangle(Left, Top, Left+2, Bottom);
        Rectangle(Left + (Right-Left)div 2, Top,
➥Left + (Right-Left)div 2+2, Bottom);
        Rectangle(Right, Top, Right+2, Bottom);
      end;
    end; // case
end;

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

continues

LISTING 10.4 Continued

```
var
  Ratio: double;
begin
  // Calculate a ratio of PageWidth to PageHeight
  Ratio := Printer.PageHeight / Printer.PageWidth;

  // Create a new TPanel instance
  with TPanel.Create(self) do
  begin
    SetBounds(15, 15, 203, trunc(203*Ratio));
    Color := clBlack;
    BevelInner := bvNone;
    BevelOuter := bvNone;
    Parent := self;
  end;

  // Create a Print preview panel
  PrintPrev := TPrintPrevPanel.Create(self);

  with PrintPrev do
  begin
    SetBounds(10, 10, 200, trunc(200*Ratio));
    Color := clWhite;
    BevelInner := bvNone;
    BevelOuter := bvNone;
    BorderStyle := bsSingle;
    Parent := self;
  end;

end;

procedure TMainForm.rgFeedTypeClick(Sender: TObject);
begin
  EnvSize := GetEnvelopeSize;
  EnvPos := GetEnvelopePos;
  ToAddrPos := GetToAddrPos;
  DrawIt;
end;

procedure TMainForm.SetCopies(Copies: Integer);
var
  ADevice, ADriver, APort: String;
  ADeviceMode: THandle;
  DevMode: PDeviceMode;
```

```
begin
 SetLength(ADevice, 255);
 SetLength(ADriver, 255);
 SetLength(APort, 255);

 { If ADeviceMode is zero, a printer driver is not loaded. Therefore,
   setting PrinterIndex forces the driver to load. }
 if ADeviceMode = 0 then
 begin
   Printer.PrinterIndex := Printer.PrinterIndex;
   Printer.GetPrinter(PChar(ADevice), PChar(ADriver),
➥PChar(APort), ADeviceMode);
 end;

 if ADeviceMode <> 0 then
 begin
   DevMode := GlobalLock(ADeviceMode);
   try
     DevMode^.dmFields := DevMode^.dmFields or DM_Copies;
     DevMode^.dmCopies := Copies;
   finally
     GlobalUnlock(ADeviceMode);
   end;
 end
 else
   raise Exception.Create('Could not set printer copies');
end;

procedure TMainForm.mmiPrintItClick(Sender: TObject);
var
  TempHeight: integer;
  SaveFont: TFont;
begin
  if PrintDialog.Execute then
  begin
    // Set the number of copies to print
    SetCopies(PrintDialog.Copies);
    Printer.BeginDoc;
    try
      // Calculate a temporary line height
      TempHeight := Printer.Canvas.TextHeight(edtName.Text);
      with ToAddrPos do
      begin
        { When printing vertically, rotate the font such that it paints
          at a 90 degree angle. }
        if (FeedType = eplVert) or (FeedType = epRVert) then
```

continues

LISTING 10.4 Continued

```
        begin
          SaveFont := TFont.Create;
          try
            // Save the original font
            SaveFont.Assign(Printer.Canvas.Font);
            RotatePrintFont;
            // Write out the address lines to the printer's Canvas
            Printer.Canvas.TextOut(Left, Bottom, edtName.Text);
            Printer.Canvas.TextOut(Left+TempHeight+2, Bottom,
➥edtStreet.Text);
            Printer.Canvas.TextOut(Left+TempHeight*2+2, Bottom,
➥edtCityState.Text);
            // Restore the original font
            Printer.Canvas.Font.Assign(SaveFont);
          finally
            SaveFont.Free;
          end;
        end
        else begin
          { If the envelope is not printed vertically, then
            just draw the address lines normally. }
          Printer.Canvas.TextOut(Left, Top, edtName.Text);
          Printer.Canvas.TextOut(Left, Top+TempHeight+2, edtStreet.Text);
          Printer.Canvas.TextOut(Left, Top+TempHeight*2+2,
➥edtCityState.Text);
        end;
      end;
    finally
      Printer.EndDoc;
    end;
  end;
end;

procedure TMainForm.RotatePrintFont;
var
  LogFont: TLogFont;
begin
  with Printer.Canvas do
  begin
    with LogFont do
    begin
      lfHeight := Font.Height; // Set to Printer.Canvas.font.height
      lfWidth := 0;            // let font mapper choose width
```

```
      lfEscapement := 900;        // tenths of degrees so 900 = 90 degrees
      lfOrientation := lfEscapement; // Always set to value of lfEscapement
      lfWeight := FW_NORMAL;   // default
      lfItalic := 0;           // no italics
      lfUnderline := 0;        // no underline
      lfStrikeOut := 0;        // no strikeout
      lfCharSet := ANSI_CHARSET;  //default
      StrPCopy(lfFaceName, Font.Name); // Printer.Canvas's font's name
      lfQuality := PROOF_QUALITY;
      lfOutPrecision := OUT_TT_ONLY_PRECIS;   // force TrueType fonts
      lfClipPrecision := CLIP_DEFAULT_PRECIS; // default
      lfPitchAndFamily := Variable_Pitch;     // default
    end;
  end;
  Printer.Canvas.Font.Handle := CreateFontIndirect(LogFont);
end;

end.
```

When the user clicks one of the radio buttons in `gbEnvelopeSize` or `gbFeedType`, the `FeedTypeClick()` event handler is called. This event handler calls the routines to calculate the envelope's size and position based on the radio button choices.

The address rectangle's size and position also are calculated in these event handlers. This rectangle's width is based on the longest text width of the text in each of the three `TEdit` components. The rectangle's height consists of the combined height of the three `TEdit` components.

All calculations are based on `Printer.Canvas`'s pixels. `mmiPrintItClick()` contains logic to print the envelope based on the choices selected. Additional logic to handle font rotation when the envelope is positioned vertically is also provided. Additionally, a pseudo–print preview is created in the `FormCreate()` event handler. This print preview is updated as the user selects the radio buttons.

The `TFeedType` enumerated type represents each position of the envelope as it may feed out of the printer:

```
TFeedType = (epLHorz, epLVert, epRHorz, epRVert);
```

`TMainForm` contains variables to hold the envelope's size and position, the address's `TRect` size and position, and the current `TFeedType`.

`TMainForm` declares the methods `GetEnvelopeSize()`, `GetEnvelopePos()`, `GetToAddrSize()`, and `GetToAddrPos()` to determine the various measurements for elements to be printed, as specified in steps 6 and 7 of this chapter's model.

In GetEnvelopeSize(), the GetDeviceCaps() function is used to convert the envelope size in inches to pixels, based on the selection from gbEnvelopeSize. GetEnvelopPos() determines the position of the envelope on TPrinter.Canvas, based on Printer.Canvas's coordinate system.

GetToAddrSize() calculates the size of the address's bounding rectangle, based on the measurements of text contained in the three TEdit components. Here, Printer.Canvas's TextHeight() and TextWidth() methods are used to determine these sizes. The function MaxLn() is a helper function used to determine the longest text line of the three TEdit components, which is used as the rectangle's width. You can also use the Max() function from the Math.pas unit to determine the longest text line.

GetToAddrPos() calls GetToAddrSize() and uses the returned value to calculate the address's bounding rectangle's position on Printer.Canvas. Note that the envelope's size and placement are needed for this function to position the address rectangle properly.

The mmiPrintItClick() event handler performs the actual printing logic. First, it initializes printing with the BeginDoc() method. Then it calculates a temporary line height used for text positioning. It determines the TFeedType, and if it's one of the vertical types, saves the printer's font and calls the method RotatePrintFont(), which rotates the font 90 degrees. When it returns form RotatePrintFont(), it restores Printer.Canvas's original font. If the TFeedType is one of the horizontal types, it performs the TextOut() calls to print the address. Finally, mmiPrintItClick() ends printing with the EndDoc() method.

RotatePrintFont() creates a TLogFont structure and initializes its various values obtained from Printer.Canvas and other default values. Notice the assignment to its lfEscapement member. Remember from Chapter 8, "Graphics Programming with GDI and Fonts," that lfEscapement specifies an angle in tenths of degrees at which the font is to be drawn. Here, you specify to print the font at a 90-degree angle by assigning 900 to lfEscapement. One thing to note here is that only TrueType fonts can be rotated.

A Simple Print Preview

Often, a good way to help your users not make a mistake by choosing the wrong selection is to enable them to view what the printed output would look like before actually printing. The project in this section contains a print preview panel. You did this by constructing a descendant class of TPanel and publicizing its Canvas property:

```
TPrintPrevPanel = class(TPanel)
public
  property Canvas; // Publicize this property
end;
```

The FormCreate() event handler performs the logic to instantiate a TPrintPrevPanel. The following line determines the ratio of the printer's width to its height:

```
Ratio := Printer.PageHeight / Printer.PageWidth;
```

This ratio is used to calculate the width and height for the `TPrintPrevPanel` instance.

Before the `TPrintPrevPanel` is created, however, a regular `TPanel` with a black color is created to serve as a shadow to the `TPrintPrevPanel` instance, `PrintPrev`. Its boundaries are adjusted so that they're slightly to the right of and below the `PrintPrev`'s boundaries. The effect is that it gives `PrintPrev` a three-dimensional look with a shadow behind it. `PrintPrev` is used primarily to show how the envelope would be printed. The routine `DrawIt()` performs this logic.

`TEnvPrintForm.DrawIt()` calls `PrintPrev.Invalidate` to erase its previous contents. Then it calls `PrintPrev.Update()` to ensure that the paint message is processed before executing the remaining code. It then sets `PrintPrev`'s mapping mode to `MM_ISOTROPIC` to allow it to accept arbitrary extents along the X- and Y-axes. `SetWindowExt()` sets `PrintPrev`'s windows' extents to those of `Printer.Canvas`, and `SetViewPortExt()` sets `PrintPrev`'s viewport extents to its own height and width (see Chapter 8, "Graphics Programming with GDI and Fonts," for a discussion on mapping modes).

This enables `DrawIt()` to use the same metric values used for the `Printer.Canvas`, the envelope, the address rectangle, and the `PrintPrev` panel. This routine also uses rectangles to represent text lines. The effect is shown in Figure 10.7.

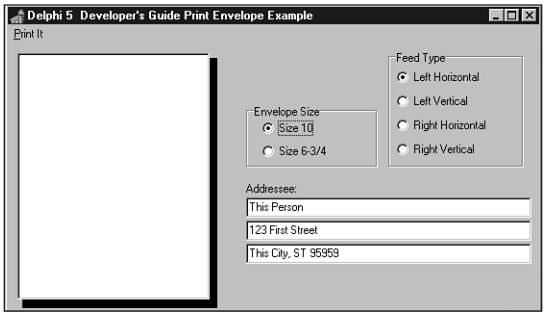

FIGURE 10.7
An envelope-printing form with a print preview feature.

NOTE

An alternative and better print preview can be created with metafiles. Create the metafile using the printer handle as the reference device, then draw into the metafile canvas just as you would the printer canvas, and then draw the metafile on the screen. No scaling or viewport extent tweaking is required.

Miscellaneous Printing Tasks

Occasionally, you'll need to perform a printing task that isn't available through the `TPrinter` object, such specifying the print quality of your print job. To perform these tasks, you must resort to the Win32 API method. However, this isn't too difficult. First, you must understand the `TDeviceMode` structure. The next section discusses this. The following sections show you how to use this structure to perform these various printing tasks.

The TDeviceMode Structure

The `TDeviceMode` structure contains information about a printer driver's initialization and environment data. Programmers use this structure to retrieve information about or set various attributes of the current printer. This structure is defined in the Windows.pas file.

You'll find definitions for each of the fields in Delphi's online help. The following sections cover some of the more common fields of this structure, but it would be a good idea to take a look at the online help and read what some of the other fields are used for. In some cases, you might need to refer to these fields, and some of them are used differently in Windows NT/2000 than in Windows 95/98.

To obtain a pointer to the current printer's `TDeviceMode` structure, you can first use `TPrinter.GetPrinter()` to obtain a handle to the memory block that the structure occupies. Then use the `GlobalLock()` function to retrieve a pointer to this structure. Listing 10.5 illustrates how to get the pointer to the `TDeviceMode` structure.

LISTING 10.5 Obtaining a Pointer to a `TDeviceMode` Structure

```
var
  ADevice, ADriver, APort: array [0..255] of Char;
  DeviceHandle: THandle;
  DevMode: PDeviceMode; // A Pointer to a TDeviceMode structure
begin
  { First obtain a handle to the TPrinter's DeviceMode structure }
  Printer.GetPrinter(ADevice, ADriver, APort, DeviceHandle);
  { If DeviceHandle is still 0, then the driver was not loaded. Set
    the printer index to force the printer driver to load making the
    handle available }
  if DeviceHandle = 0 then
  begin
    Printer.PrinterIndex := Printer.PrinterIndex;
    Printer.GetPrinter(ADevice, ADriver, APort, DeviceHandle);
  end;
  { If DeviceHandle is still 0, then an error has occurred. Otherwise,
    use GlobalLock() to get a pointer to the TDeviceMode structure }
```

```
  if DeviceHandle = 0 then
    Raise Exception.Create('Could Not Initialize TDeviceMode structure')
  else
    DevMode := GlobalLock(DeviceHandle);
  { Code to use the DevMode structure goes here }
  { !!!! }
  if not DeviceHandle = 0 then
    GlobalUnlock(DeviceHandle);
end;
```

The comments in the preceding listing explain the steps required to obtain the pointer to the `TDeviceMode` structure. After you've obtained this pointer, you can perform various printer routines, as illustrated in the following sections. First, however, notice this comment in the preceding listing:

```
{ Code to use the DevMode structure goes here }
  { !!!! }
```

It's here that you place the code examples to follow.

Before you can initialize any of the members of the `TDeviceMode` structure, however, you must specify which member you're initializing by setting the appropriate bit in the `dmFields` bit flags. Table 10.3 lists the various bit flags of `dmFields` and also specifies to which `TDeviceMode` member they pertain.

TABLE 10.3 `TDeviceMode.dmFields` Bit Flags

dmField Value	*Corresponding Field*
DM_ORIENTATION	dmOrientation
DM_PAPERSIZE	dmPaperSize
DM_PAPERLENGTH	dmPaperLength
DM_PAPERWIDTH	dmPaperWidth
DM_SCALE	dmScale
DM_COPIES	dmCopies
DM_DEFAULTSOURCE	dmDefaultSource
DM_PRINTQUALITY	dmPrintQuality
DM_COLOR	dmColor
DM_DUPLEX	dmDuplex
DM_YRESOLUTION	dmYResolution
DM_TTOPTION	dmTTOption
DM_COLLATE	dmCollate

continues

TABLE 10.3 Continued

dmField Value	*Corresponding Field*
DM_FORMNAME	dmFormName
DM_LOGPIXELS	dmLogPixels
DM_BITSPERPEL	dmBitsPerPel
DM_PELSWIDTH	dmPelsWidth
DM_PELSHEIGHT	dmPelsHeight
DM_DISPLAYFLAGS	dmDisplayFlags
DM_DISPLAYFREQUENCY	dmDisplayFrequency
DM_ICMMETHOD	dmICMMethod (Windows 95 only)
DM_ICMINTENT	dmICMIntent (Windows 95 only)
DM_MEDIATYPE	dmMediaType (Windows 95 only)
DM_DITHERTYPE	dmDitherType (Windows 95 only)

In the examples that follow, you'll see how to set the appropriate bit flag as well as the corresponding TDeviceMode member.

Specifying Copies to Print

You can tell a print job how many copies to print by specifying the number of copies in the dmCopies field of the TDeviceMode structure. The following code illustrates how to do this:

```
with DevMode^ do
begin
   dmFields := dmFields or DM_COPIES;
   dmCopies := Copies;
end;
```

First, you must set the appropriate bit flag of the dmFields field to indicate which member of the TDeviceMode structure has been initialized. The preceding code is what you would insert into the code in Listing 10.6 where specified. Then, whenever you start your print job, the number of copies specified should be sent to the printer. It's worth mentioning that although this examples illustrates how to set the copies to print using the TDeviceMode structure, the TPrinter.Copies property does the same.

Specifying Printer Orientation

Specifying printer orientation is similar to specifying copies except that you initialize a different TDeviceMode structure:

```
with DevMode^ do
begin
   dmFields := dmFields or DM_ORIENTATION;
   dmOrientation := DMORIENT_LANDSCAPE;
end;
```

The two options for `dmOrientation` are `DMORIENT_LANDSCAPE` and `DMORIENT_PORTRAIT`. You might also look at the `TPrinter.Orientation` property.

Specifying Paper Size

To specify a paper size, you initialize `TDeviceMode`'s `dmPaperSize` member:

```
with DevMode^ do
begin
   dmFields := dmFields or DM_PAPERSIZE;
   dmPaperSize := DMPAPER_LETTER;  // Letter, 8-1/2 by 11 inches
end;
```

Several predefined values exist for the `dmPaperSize` member, which you can look up in the online help under `TDeviceMode`. The `dmPaperSize` member can be set to zero if the paper size is specified by the `dmPaperWidth` and `dmPaperHeight` members.

Specifying Paper Length

You can specify the paper length in tenths of a millimeter for the printed output by setting the `dmPaperLength` field. This overrides any settings applied to the `dmPaperSize` field. The following code illustrates setting the paper length:

```
with DevMode^ do
begin
   dmFields := dmFields or DM_PAPERLENGTH;
   dmPaperLength := SomeLength;
end;
```

Specifying Paper Width

Paper width is also specified in tenths of a millimeter. To set the paper width, you must initialize the `dmPaperWidth` field of the `TDeviceMode` structure. The following code illustrates this setting:

```
with DevMode^ do
begin
   dmFields := dmFields or DM_PAPERWIDTH;
   dmPaperWidth := SomeWidth;
end;
```

This also overrides the settings for the `dmPaperSize` field.

Specifying Print Scale

The *print scale* is the factor by which the printed output is scaled. Therefore, the resulting page size is scaled from the physical page size by a factor of `TDeviceMode.dmScale` divided by 100. Therefore, to shrink the printed output (graphics and text) by half their original size, you would assign the value of 50 to the `dmScale` field. The following code illustrates how to set the print scale:

```
with DevMode^ do
begin
   dmFields := dmFields or DM_SCALE;
   dmScale := 50;
end;
```

Specifying Print Color

For printers that support color printing, you can specify whether the printer is to render color or monochrome printing by initializing the `dmColor` field, as shown here:

```
with DevMode^ do
begin
   dmFields := dmFields or DM_COLOR;
   dmColor := DMCOLOR_COLOR;
end;
```

Another value that can be assigned to the `dmColor` field is `DMCOLOR_MONOCHROME`.

Specifying Print Quality

Print quality is the resolution at which the printer prints its output. Four predefined values exist for setting the print quality, as shown in the following list:

- `DMRES_HIGH`. High-resolution printing
- `DMRES_MEDIUM`. Medium-resolution printing
- `DMRES_LOW`. Low-resolution printing
- `DMRES_DRAFT`. Draft-resolution printing

To change the quality of print, you initialize the `dmPrintQuality` field of the `TDeviceMode` structure:

```
with DevMode^ do
begin
   dmFields := dmFields or DM_PRINTQUALITY;
   dmPrintQuality := DMRES_DRAFT;
end;
```

Specifying Duplex Printing

Some printers are capable of duplex printing—printing on both sides of the paper. You can tell the printer to perform double-sided printing by initializing the `dmDuplex` field of the `TDeviceMode` structure to one of these values:

- `DMDUP_SIMPLEX`
- `DMDUP_HORIZONTAL`
- `DMDUP_VERTICAL`

Here's an example:

```
with DevMode^ do
begin
   dmFields := dmFields or DM_DUPLEX;
   dmDuplex := DMDUP_HORIZONTAL;
end;
```

Changing the Default Printer

Although it's possible to change the default printer by launching the printer folder, you might want to change the default printer at runtime. This is possible as illustrated in the sample project shown in Listing 10.6.

LISTING 10.6 Changing the Default Printer

```
unit MainFrm;

interface

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

type
  TMainForm = class(TForm)
    cbPrinters: TComboBox;
    lblPrinter: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure cbPrintersChange(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

continues

LISTING 10.6 Continued

```
var
  MainForm: TMainForm;

implementation
uses IniFiles, Printers;

{$R *.DFM}

procedure TMainForm.FormCreate(Sender: TObject);
begin
  { Copy the printer names to the combobox and set the combobox to
    show the currently selected default printer }
  cbPrinters.Items.Assign(Printer.Printers);
  cbPrinters.Text := Printer.Printers[Printer.PrinterIndex];
  // Update the label to reflect the default printer
  lblPrinter.Caption := Printer.Printers[Printer.PrinterIndex];
end;

procedure TMainForm.cbPrintersChange(Sender: TObject);
var
  IniFile: TIniFile;
  TempStr1, TempStr2: String;
begin
  with Printer do
  begin
    // Set the new printer based on the ComboBox's selected printer
    PrinterIndex := cbPrinters.ItemIndex;
    // Store the printer name into a temporary string
    TempStr1 := Printers[PrinterIndex];
    // Delete the unnecessary portion of the printer name
    System.Delete(TempStr1, Pos(' on ', TempStr1), Length(TempStr1));
    // Create a TIniFile class
    IniFile := TIniFile.Create('WIN.INI');
    try
      // Retrieve the device name of the selected printer
      TempStr2 := IniFile.ReadString('Devices', TempStr1, '');
      // Change the default printer to that chosen by the user
      IniFile.WriteString('windows', 'device', TempStr1 + ',' + TempStr2);
    finally
      IniFile.Free;
    end;
  end;
  // Update the label to reflect the new printer selection
  lblPrinter.Caption := Printer.Printers[Printer.PrinterIndex];
end;

end.
```

The preceding project consists of a main form with a `TComboBox` and a `TLabel` component. Upon form creation, the `TComboBox` component is initialized with the string list of printer names obtained from the `Printer.Printers` property. The `TLabel` component is then updated to reflect the currently selected printer. The `cbPrintersChange()` event handler is where we placed the code to modify the system-wide default printer. What this entails is changing the `[device]` entry in the `[windows]` section of the `WIN.INI` file, located in the `Windows` directory. The comments in the preceding code go on to explain the process of making these modifications.

Obtaining Printer Information

This section illustrates how you can retrieve information about a printer device such as physical characteristics (number of bins, paper sizes supported, and so on) as well as the printer's text- and graphics-drawing capabilities.

You might want to get information about a particular printer for several reasons. For example, you might need to know whether the printer supports a particular capability. A typical example is to determine whether the current printer supports banding. *Banding* is a process that can improve printing speed and disk space requirements for printers with memory limitations. To use banding, you must make API calls specific to this capability. On a printer that doesn't support this capability, these calls wouldn't function. Therefore, you can first determine whether the printer will support banding (and use it, if so); otherwise, you can avoid the banding API calls.

GetDeviceCaps() and DeviceCapabilities()

The Win32 API function `GetDeviceCaps()` allows you to obtain information about devices such as printers, plotters, screens, and so on. Generally, these are devices that have a device context. You use `GetDeviceCaps()` by supplying it a handle to a device context and an index that specifies the information you want to retrieve.

`DeviceCapabilities()` is specific to printers. In fact, the information obtained from `DeviceCapabilities()` is provided by the printer driver for a specified printer. Use `DeviceCapabilities()` by supplying it with strings identifying the printer device as well as an index specifying the data you want to retrieve. Sometimes two calls to `DeviceCapabilities()` are required to retrieve certain data. The first call is made to determine how much memory you must allocate for the data to be retrieved. The second call stores the data in the memory block you've allocated. This section illustrates how to do this.

One thing you should know is that most of the drawing capabilities that aren't supported by a particular printer will still work if you use them. For example, when `GetDeviceCaps()` or `DeviceCapabilities()` indicates that `BitBlt()`, `StretchBlt()`, or printing TrueType fonts isn't supported, you can still use any of these functions; GDI will simulate these functions for you. Note, however, that GDI cannot simulate `BitBlt()` on a device that doesn't support raster scanline pixels; `BitBlt()` will always fail on a pen plotter, for example.

Printer Information Sample Program

Figure 10.8 shows the main form for the sample program. This program contains eight pages, each of which lists different printer capabilities for the printer selected in the combo box.

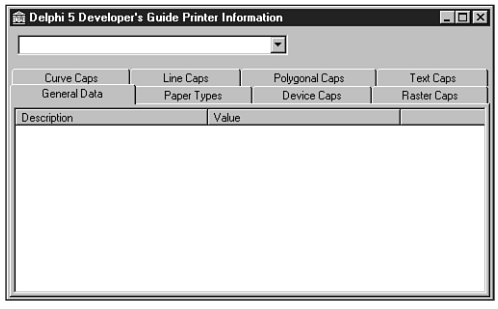

FIGURE 10.8
The main form for the printer information example.

Declaring the DeviceCapabilitiesA Function

If you attempt to use the function `DeviceCapabilities()` defined in `Windows.pas`, you won't be able to run your program because this function isn't defined in `GDI32.DLL` as `Windows.pas` indicates. Instead, this function in `GDI32.DLL` is `DeviceCapabilitiesEx()`. However, even if you define this function's prototype as follows, the function won't work as expected and returns erroneous results:

```
function DeviceCapabilitiesEx(pDevice, pPort: Pchar; fwCapability: Word;
         pOutput: Pchar; DevMode: PdeviceMode):
         Integer; stdcall; external 'Gdi32.dll';
```

It turns out that two functions—`DeviceCapabilitiesA()` for ANSI strings and `DeviceCapabilitiesW()` for wide strings—are defined in `WINSPOOL.DRV`, which is the Win32 print spooler interface. This function is the correct one to use as indicated in the Microsoft Developer's Network CD (MSDN). The correct definition for the function prototype that's used in the sample program in Listing 10.8 (shown in the following section) is as follows:

```
function DeviceCapabilitiesA(pDevice, pPort: Pchar; fwCapability: Word;
         pOutput: Pchar; DevMode: PdeviceMode):
         Integer; stdcall; external 'winspool.drv';
```

Note that the preceding declaration can be found in `WINSPOOL.PAS` in Delphi 5.

Sample Program Functionality

Listing 10.8 (shown at the end of this section) contains the source for the Printer Information sample program. The main form's `OnCreate` event handler simply populates the combo box

with the list of available printers on the system. The `OnChange` event handler for the combo box is the central point of the application where the methods to retrieve the printer information are called.

The first page on the form General Data contains general information about the printer device. You'll see that the printer's device name, driver, and port location are obtained by calling the `TPrinter.GetPrinter()` method. This method also retrieves a handle to a `TDeviceMode` structure for the currently selected printer. This information is then added to the General Data page. To retrieve the printer driver version, you use the `DeviceCapabilitiesA()` function and pass the `DC_DRIVER` index. The rest of the `PrinterComboChange` event handler calls the various routines to populate the list boxes on the various pages of the main form.

The `GetBinNames()` method illustrates how to use the `DeviceCapabilitiesA()` function to retrieve the bin names for the selected printer. This method first gets the number of bin names available by calling `DeviceCapabilitiesA()`, passing the `DC_BINNAMES` index, and passing `nil` as the `pOutput` and `DevMode` parameters. The result of this function call specifies how much memory must be allocated to hold the bin names. According to the documentation on `DeviceCapabilitiesA()`, each bin name is defined as an array of 24 characters. We defined a `TBinName` data type like this:

```
TBinName = array[0..23] of char;
```

We also defined an array of `TBinName`:

```
TBinNames = array[0..0] of TBinName;
```

This type is used to typecast a pointer as an array of `TBinName data types`. To access an element at some index into the array, you must disable range checking, because this array is defined to have a range of `0..0`, as illustrated in the `GetBinNames()` method. The bin names are added to the appropriate list box.

This same technique of determining the amount of memory required and allocating this memory dynamically is also used in the methods `GetDevCapsPaperNames()` and `GetResolutions()`.

The methods `GetDuplexSupport()`, `GetCopies()`, and `GetEMFStatus()` all use the `DeviceCapabilitiesA()` function to return a value of the requested information. For example, the following code determines whether the selected printer supports duplex printing by returning a value of `1` if duplex printing is supported or `0` if not:

```
DeviceCapabilitiesA(Device, Port, DC_DUPLEX, nil, nil);
```

Also, the following statement returns the maximum number of copies the device can print:

```
DeviceCapabilitiesA(Device, Port, DC_COPIES, nil, nil);
```

The remaining methods use the `GetDeviceCaps()` function to determine the various capabilities of the selected device. In some cases, `GetDeviceCaps()` returns the specific value requested. For example, the following statement returns the width, in millimeters, of the printer device:

```
GetDeviceCaps(Printer.Handle, HORZSIZE);
```

In other cases, `GetDeviceCaps()` returns an integer value whose bits are masked to determine a particular capability. For example, the `GetRasterCaps()` method first retrieves the integer value that contains the bitmasked fields:

```
RCaps := GetDeviceCaps(Printer.Handle, RASTERCAPS);
```

Then, to determine whether the printer supports banding, you must mask out the `RC_BANDING` field by performing an `AND` operation whose result should equal the value of `RC_BANDING`:

```
(RCaps and RC_BANDING) = RC_BANDING
```

This evaluation is passed to one of the helper functions, `BoolToYesNoStr()`, which returns the string `Yes` or `No`, based on the result of the evaluation. Other fields are masked in the same manner. This same technique is used in other areas where bitmasked fields are returned from `GetDeviceCaps()` as well as from the `DeviceCapabilitiesA()` function, such as in the `GetTrueTypeInfo()` method.

You'll find both functions, `DeviceCapabilties()` and `GetDeviceCaps()`, well documented in the online Win32 API help.

LISTING 10.7 Printer Information Sample Program

```
unit MainFrm;

interface

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

type
  TMainForm = class(TForm)
    pgcPrinterInfo: TPageControl;
    tbsPaperTypes: TTabSheet;
    tbsGeneralData: TTabSheet;
    lbPaperTypes: TListBox;
    tbsDeviceCaps: TTabSheet;
    tbsRasterCaps: TTabSheet;
    tbsCurveCaps: TTabSheet;
    tbsLineCaps: TTabSheet;
    tbsPolygonalCaps: TTabSheet;
```

```
    tbsTextCaps: TTabSheet;
    lvGeneralData: TListView;
    lvCurveCaps: TListView;
    Splitter1: TSplitter;
    lvDeviceCaps: TListView;
    lvRasterCaps: TListView;
    pnlTop: TPanel;
    cbPrinters: TComboBox;
    lvLineCaps: TListView;
    lvPolyCaps: TListView;
    lvTextCaps: TListView;
    procedure FormCreate(Sender: TObject);
    procedure cbPrintersChange(Sender: TObject);
  private
    Device, Driver, Port: array[0..255] of char;
    ADevMode: THandle;
   public
    procedure GetBinNames;
    procedure GetDuplexSupport;
    procedure GetCopies;
    procedure GetEMFStatus;
    procedure GetResolutions;
    procedure GetTrueTypeInfo;
    procedure GetDevCapsPaperNames;
    procedure GetDevCaps;
    procedure GetRasterCaps;
    procedure GetCurveCaps;
    procedure GetLineCaps;
    procedure GetPolyCaps;
    procedure GetTextCaps;
  end;

var
  MainForm: TMainForm;

implementation
uses
  Printers, WinSpool;

const
  NoYesArray: array[Boolean] of String = ('No', 'Yes');
type

  // Types for holding bin names
  TBinName = array[0..23] of char;
  // Where used set $R- to prevent error
```

continues

LISTING 10.7 Continued

```
  TBinNames = array[0..0] of TBinName;

  // Types for holding paper names
  TPName = array[0..63] of char;

  // Where used set $R- to prevent error
  TPNames = array[0..0] of TPName;

  // Types for holding resolutions
  TResolution = array[0..1] of integer;
  // Where used set $R- to prevent error
  TResolutions = array[0..0] of TResolution;

  // Type for holding array of pages sizes (word types)
  TPageSizeArray = Array[0..0] of word;

var
 Rslt: Integer;

{$R *.DFM}
(*
function BoolToYesNoStr(aVal: Boolean): String;
// Returns the string "YES" or "NO" based on the boolean value
begin
  if aVal then
    Result := 'Yes'
  else
    Result := 'No';
end;
*)
procedure AddListViewItem(const aCaption, aValue: String; aLV: TListView);
// This method is used to add a TListItem to the TListView, aLV
var
  NewItem: TListItem;
begin
  NewItem := aLV.Items.Add;
  NewItem.Caption := aCaption;
  NewItem.SubItems.Add(aValue);
end;

procedure TMainForm.GetBinNames;
var
  BinNames: Pointer;
  i: integer;
```

```
begin
{$R-} // Range checking must be turned off here.
  // First determine how many bin names are available.
  Rslt := DeviceCapabilitiesA(Device, Port, DC_BINNAMES, nil, nil);
  if Rslt > 0 then
  begin
    { Each bin name is 24 bytes long. Therefore, allocate Rslt*24 bytes to hold
      the bin names. }
    GetMem(BinNames, Rslt*24);
    try
      // Now retrieve the bin names in the allocated block of memory.
      if DeviceCapabilitiesA(Device, Port, DC_BINNAMES, BinNames, nil) = -1
then
        raise Exception.Create('DevCap Error');
      //{ Add the information to the appropriate list box.
      AddListViewItem('BIN NAMES', EmptyStr, lvGeneralData);
      for i := 0 to Rslt - 1 do
      begin
        AddListViewItem(Format('   Bin Name %d', [i]),
          StrPas(TBinNames(BinNames^)[i]), lvGeneralData);
      end;
    finally
      FreeMem(BinNames, Rslt*24);
    end;
  end;
{$R+} // Turn range checking back on.
end;

procedure TMainForm.GetDuplexSupport;
begin
  { This function uses DeviceCapabilitiesA to determine whether or not the
    printer device supports duplex printing. }
  Rslt := DeviceCapabilitiesA(Device, Port, DC_DUPLEX, nil, nil);
  AddListViewItem('Duplex Printing', NoYesArray[Rslt = 1], lvGeneralData);
end;

procedure TMainForm.GetCopies;
begin
  { This function determines how many copies the device can be set to print.
    If the result is not greater than 1 then the print logic must be
    executed multiple times }
  Rslt := DeviceCapabilitiesA(Device, Port, DC_COPIES, nil, nil);
  AddListViewItem('Copies that printer can print',
     InttoStr(Rslt), lvGeneralData);

end;
```

continues

LISTING 10.7 Continued

```
procedure TMainForm.GetEMFStatus;
begin
  // This function determines if the device supports the enhanced metafiles.
  Rslt := DeviceCapabilitiesA(Device, Port, DC_EMF_COMPLIANT, nil, nil);
  AddListViewItem('EMF Compliant', NoYesArray[Rslt=1], lvGeneralData);
end;

procedure TMainForm.GetResolutions;
var
  Resolutions: Pointer;
  i: integer;
begin
{$R-}  // Range checking must be turned off.
  // Determine how many resolutions are available.
  Rslt := DeviceCapabilitiesA(Device, Port, DC_ENUMRESOLUTIONS, nil, nil);
  if Rslt > 0 then begin
    { Allocate the memory to hold the different resolutions which are
      represented by integer pairs, ie: 300, 300 }
    GetMem(Resolutions, (SizeOf(Integer)*2)*Rslt);
    try
      // Retrieve the different resolutions.
      if DeviceCapabilitiesA(Device, Port, DC_ENUMRESOLUTIONS,
      Resolutions, nil) = -1 then
        Raise Exception.Create('DevCaps Error');
      // Add the resolution information to the appropriate list box.
      AddListViewItem('RESOLUTION CONFIGURATIONS', EmptyStr, lvGeneralData);

      for i := 0 to Rslt - 1 do
      begin
        AddListViewItem('   Resolution Configuration',
          IntToStr(TResolutions(Resolutions^)[i][0])+
          ' '+IntToStr(TResolutions(Resolutions^)[i][1]), lvGeneralData);
      end;
    finally
      FreeMem(Resolutions, SizeOf(Integer)*Rslt*2);
    end;
  end;
{$R+}  // Turn range checking back on.
end;

procedure TMainForm.GetTrueTypeInfo;
begin
  // Get the TrueType font capabilities of the device represented as bitmasks
  Rslt := DeviceCapabilitiesA(Device, Port, DC_TRUETYPE, nil, nil);
  if Rslt <> 0 then
```

```
    { Now mask out the individual TrueType capabilities and indicate the
      result in the appropriate list box. }
    AddListViewItem('TRUE TYPE FONTS', EmptyStr, lvGeneralData);
    with lvGeneralData.Items do
    begin
      AddListViewItem('   Prints TrueType fonts as graphics',
        NoYesArray[(Rslt and DCTT_BITMAP) = DCTT_BITMAP], lvGeneralData);

      AddListViewItem('   Downloads TrueType fonts',
        NoYesArray[(Rslt and DCTT_DOWNLOAD) = DCTT_DOWNLOAD],
➥lvGeneralData);

      AddListViewItem('   Downloads outline TrueType fonts',
        NoYesArray[(Rslt and DCTT_DOWNLOAD_OUTLINE) =
➥DCTT_DOWNLOAD_OUTLINE],
        lvGeneralData);

      AddListViewItem('   Substitutes device for TrueType fonts',
        NoYesArray[(Rslt and DCTT_SUBDEV) = DCTT_SUBDEV], lvGeneralData);
    end;
end;

procedure TMainForm.GetDevCapsPaperNames;
{ This method gets the paper types available on a selected printer from the
  DeviceCapabilitiesA function. }
var
  PaperNames: Pointer;
  i: integer;
begin
{$R-}  // Range checking off.
  lbPaperTypes.Items.Clear;
  // First get the number of paper names available.
  Rslt := DeviceCapabilitiesA(Device, Port, DC_PAPERNAMES, nil, nil);
  if Rslt > 0 then begin
    { Now allocate the array of paper names. Each paper name is 64 bytes.
      Therefore, allocate Rslt*64 of memory. }
    GetMem(PaperNames, Rslt*64);
    try
      // Retrieve the list of names into the allocated memory block.
      if DeviceCapabilitiesA(Device, Port, DC_PAPERNAMES,
      PaperNames, nil) = - 1 then
        raise Exception.Create('DevCap Error');
      // Add the paper names to the appropriate list box.
      for i := 0 to Rslt - 1 do
       lbPaperTypes.Items.Add(StrPas(TPNames(PaperNames^)[i]));
    finally
```

continues

LISTING 10.7 Continued

```
      FreeMem(PaperNames, Rslt*64);
    end;
  end;
{$R+} // Range checking back on.
end;

procedure TMainForm.GetDevCaps;
{ This method retrieves various capabilities of the selected printer device by
  using the GetDeviceCaps function. Refer to the Online API help for the
  meaning of each of these items. }
begin
  with lvDeviceCaps.Items do
  begin
    Clear;
    AddListViewItem('Width in millimeters',
      IntToStr(GetDeviceCaps(Printer.Handle, HORZSIZE)), lvDeviceCaps);
    AddListViewItem('Height in millimeter',
      IntToStr(GetDeviceCaps(Printer.Handle, VERTSIZE)), lvDeviceCaps);
    AddListViewItem('Width in pixels',
      IntToStr(GetDeviceCaps(Printer.Handle, HORZRES)), lvDeviceCaps);
    AddListViewItem('Height in pixels',
      IntToStr(GetDeviceCaps(Printer.Handle, VERTRES)), lvDeviceCaps);
    AddListViewItem('Pixels per horizontal inch',
      IntToStr(GetDeviceCaps(Printer.Handle, LOGPIXELSX)), lvDeviceCaps);
    AddListViewItem('Pixels per vertical inch',
      IntToStr(GetDeviceCaps(Printer.Handle, LOGPIXELSY)), lvDeviceCaps);
    AddListViewItem('Color bits per pixel',
      IntToStr(GetDeviceCaps(Printer.Handle, BITSPIXEL)), lvDeviceCaps);
    AddListViewItem('Number of color planes',
      IntToStr(GetDeviceCaps(Printer.Handle, PLANES)), lvDeviceCaps);
    AddListViewItem('Number of brushes',
      IntToStr(GetDeviceCaps(Printer.Handle, NUMBRUSHES)), lvDeviceCaps);
    AddListViewItem('Number of pens',
      IntToStr(GetDeviceCaps(Printer.Handle, NUMPENS)), lvDeviceCaps);
    AddListViewItem('Number of fonts',
      IntToStr(GetDeviceCaps(Printer.Handle, NUMFONTS)), lvDeviceCaps);
    Rslt := GetDeviceCaps(Printer.Handle, NUMCOLORS);
    if Rslt = -1 then
      AddListViewItem('Number of entries in color table', ' > 8', lvDeviceCaps)
    else AddListViewItem('Number of entries in color table',
        IntToStr(Rslt), lvDeviceCaps);
    AddListViewItem('Relative pixel drawing width',
      IntToStr(GetDeviceCaps(Printer.Handle, ASPECTX)), lvDeviceCaps);
    AddListViewItem('Relative pixel drawing height',
```

```
      IntToStr(GetDeviceCaps(Printer.Handle, ASPECTY)), lvDeviceCaps);
    AddListViewItem('Diagonal pixel drawing width',
      IntToStr(GetDeviceCaps(Printer.Handle, ASPECTXY)), lvDeviceCaps);
    if GetDeviceCaps(Printer.Handle, CLIPCAPS) = 1 then
      AddListViewItem('Clip to rectangle', 'Yes', lvDeviceCaps)
    else AddListViewItem('Clip to rectangle', 'No', lvDeviceCaps);
  end;
end;

procedure TMainForm.GetRasterCaps;
{ This method gets the various raster capabilities of the selected printer
  device by using the GetDeviceCaps function with the RASTERCAPS index. Refer
  to the online help for information on each capability. }
var
  RCaps: Integer;
begin
  with lvRasterCaps.Items do
  begin
    Clear;
    RCaps := GetDeviceCaps(Printer.Handle, RASTERCAPS);
    AddListViewItem('Banding',
      NoYesArray[(RCaps and RC_BANDING) = RC_BANDING], lvRasterCaps);
    AddListViewItem('BitBlt Capable',
      NoYesArray[(RCaps and RC_BITBLT) = RC_BITBLT], lvRasterCaps);
    AddListViewItem('Supports bitmaps > 64K',
      NoYesArray[(RCaps and RC_BITMAP64) = RC_BITMAP64], lvRasterCaps);
    AddListViewItem('DIB support',
      NoYesArray[(RCaps and RC_DI_BITMAP) = RC_DI_BITMAP], lvRasterCaps);
    AddListViewItem('Floodfill support',
      NoYesArray[(RCaps and RC_FLOODFILL) = RC_FLOODFILL], lvRasterCaps);
    AddListViewItem('Windows 2.0 support',
      NoYesArray[(RCaps and RC_GDI20_OUTPUT) = RC_GDI20_OUTPUT],
➥lvRasterCaps);
    AddListViewItem('Palette based device',
      NoYesArray[(RCaps and RC_PALETTE) = RC_PALETTE], lvRasterCaps);
    AddListViewItem('Scaling support',
      NoYesArray[(RCaps and RC_SCALING) = RC_SCALING], lvRasterCaps);
    AddListViewItem('StretchBlt support',
      NoYesArray[(RCaps and RC_STRETCHBLT) = RC_STRETCHBLT],
➥lvRasterCaps);
    AddListViewItem('StretchDIBits support',
      NoYesArray[(RCaps and RC_STRETCHDIB) = RC_STRETCHDIB],
➥lvRasterCaps);
  end;
end;
```

continues

LISTING 10.7 Continued

```
procedure TMainForm.GetCurveCaps;
{ This method gets the various curve capabilities of the selected printer
  device by using the GetDeviceCaps function with the CURVECAPS index. Refer
  to the online help for information on each capability. }
var
  CCaps: Integer;
begin
  with lvCurveCaps.Items do
  begin
    Clear;
    CCaps := GetDeviceCaps(Printer.Handle, CURVECAPS);

    AddListViewItem('Curve support',
      NoYesArray[(CCaps and CC_NONE) = CC_NONE], lvCurveCaps);

    AddListViewItem('Circle support',
      NoYesArray[(CCaps and CC_CIRCLES) = CC_CIRCLES], lvCurveCaps);

    AddListViewItem('Pie support',
      NoYesArray[(CCaps and CC_PIE) = CC_PIE], lvCurveCaps);

    AddListViewItem('Chord arc support',
      NoYesArray[(CCaps and CC_CHORD) = CC_CHORD], lvCurveCaps);

    AddListViewItem('Ellipse support',
      NoYesArray[(CCaps and CC_ELLIPSES) = CC_ELLIPSES], lvCurveCaps);

    AddListViewItem('Wide border support',
      NoYesArray[(CCaps and CC_WIDE) = CC_WIDE], lvCurveCaps);

    AddListViewItem('Styled border support',
      NoYesArray[(CCaps and CC_STYLED) = CC_STYLED], lvCurveCaps);

    AddListViewItem('Round rectangle support',
      NoYesArray[(CCaps and CC_ROUNDRECT) = CC_ROUNDRECT], lvCurveCaps);

  end;
end;

procedure TMainForm.GetLineCaps;
{ This method gets the various line drawing capabilities of the selected
printer device by using the GetDeviceCaps function with the LINECAPS index.
Refer to the online help for information on each capability. }
var
  LCaps: Integer;
```

```
begin
  with lvLineCaps.Items do
  begin
    Clear;
    LCaps := GetDeviceCaps(Printer.Handle, LINECAPS);

    AddListViewItem('Line support',
      NoYesArray[(LCaps and LC_NONE) = LC_NONE], lvLineCaps);

    AddListViewItem('Polyline support',
      NoYesArray[(LCaps and LC_POLYLINE) = LC_POLYLINE], lvLineCaps);

    AddListViewItem('Marker support',
      NoYesArray[(LCaps and LC_MARKER) = LC_MARKER], lvLineCaps);

    AddListViewItem('Multiple marker support',
      NoYesArray[(LCaps and LC_POLYMARKER) = LC_POLYMARKER], lvLineCaps);

    AddListViewItem('Wide line support',
      NoYesArray[(LCaps and LC_WIDE) = LC_WIDE], lvLineCaps);

    AddListViewItem('Styled line support',
      NoYesArray[(LCaps and LC_STYLED) = LC_STYLED], lvLineCaps);

    AddListViewItem('Wide and styled line support',
      NoYesArray[(LCaps and LC_WIDESTYLED) = LC_WIDESTYLED], lvLineCaps);

    AddListViewItem('Interior support',
      NoYesArray[(LCaps and LC_INTERIORS) = LC_INTERIORS], lvLineCaps);
  end;
end;

procedure TMainForm.GetPolyCaps;
{ This method gets the various polygonal capabilities of the selected printer
  device by using the GetDeviceCaps function with the POLYGONALCAPS index.
Refer to the online help for information on each capability. }
var
  PCaps: Integer;
begin
  with lvPolyCaps.Items do
  begin
    Clear;
    PCaps := GetDeviceCaps(Printer.Handle, POLYGONALCAPS);

    AddListViewItem('Polygon support',
      NoYesArray[(PCaps and PC_NONE) = PC_NONE], lvPolyCaps);
```

continues

LISTING 10.7 Continued

```
    AddListViewItem('Alternate fill polygon support',
      NoYesArray[(PCaps and PC_POLYGON) = PC_POLYGON], lvPolyCaps);

    AddListViewItem('Rectangle support',
      NoYesArray[(PCaps and PC_RECTANGLE) = PC_RECTANGLE], lvPolyCaps);

    AddListViewItem('Winding-fill polygon support',
      NoYesArray[(PCaps and PC_WINDPOLYGON) = PC_WINDPOLYGON], lvPolyCaps);

    AddListViewItem('Single scanline support',
      NoYesArray[(PCaps and PC_SCANLINE) = PC_SCANLINE], lvPolyCaps);

    AddListViewItem('Wide border support',
      NoYesArray[(PCaps and PC_WIDE) = PC_WIDE], lvPolyCaps);

    AddListViewItem('Styled border support',
      NoYesArray[(PCaps and PC_STYLED) = PC_STYLED], lvPolyCaps);

    AddListViewItem('Wide and styled border support',
      NoYesArray[(PCaps and PC_WIDESTYLED) = PC_WIDESTYLED], lvPolyCaps);

    AddListViewItem('Interior support',
      NoYesArray[(PCaps and PC_INTERIORS) = PC_INTERIORS], lvPolyCaps);
  end;
end;

procedure TMainForm.GetTextCaps;
{ This method gets the various text drawing capabilities of the selected
  printer device by using the GetDeviceCaps function with the TEXTCAPS index.
  Refer to the online help for information on each capability. }
var
  TCaps: Integer;
begin
  with lvTextCaps.Items do
  begin
    Clear;
    TCaps := GetDeviceCaps(Printer.Handle, TEXTCAPS);

    AddListViewItem('Character output precision',
      NoYesArray[(TCaps and TC_OP_CHARACTER) = TC_OP_CHARACTER], lvTextCaps);

    AddListViewItem('Stroke output precision',
      NoYesArray[(TCaps and TC_OP_STROKE) = TC_OP_STROKE], lvTextCaps);

    AddListViewItem('Stroke clip precision',
```

```
      NoYesArray[(TCaps and TC_CP_STROKE) = TC_CP_STROKE], lvTextCaps);

    AddListViewItem('90 degree character rotation',
      NoYesArray[(TCaps and TC_CR_90) = TC_CR_90], lvTextCaps);

    AddListViewItem('Any degree character rotation',
      NoYesArray[(TCaps and TC_CR_ANY) = TC_CR_ANY], lvTextCaps);

    AddListViewItem('Independent scale in X and Y direction',
      NoYesArray[(TCaps and TC_SF_X_YINDEP) = TC_SF_X_YINDEP], lvTextCaps);

    AddListViewItem('Doubled character for scaling',
      NoYesArray[(TCaps and TC_SA_DOUBLE) = TC_SA_DOUBLE], lvTextCaps);

    AddListViewItem('Integer multiples only for character scaling',
      NoYesArray[(TCaps and TC_SA_INTEGER) = TC_SA_INTEGER], lvTextCaps);

    AddListViewItem('Any multiples for exact character scaling',
      NoYesArray[(TCaps and TC_SA_CONTIN) = TC_SA_CONTIN], lvTextCaps);

    AddListViewItem('Double weight characters',
      NoYesArray[(TCaps and TC_EA_DOUBLE) = TC_EA_DOUBLE], lvTextCaps);

    AddListViewItem('Italicized characters',
      NoYesArray[(TCaps and TC_IA_ABLE) = TC_IA_ABLE], lvTextCaps);

    AddListViewItem('Underlined characters',
      NoYesArray[(TCaps and TC_UA_ABLE) = TC_UA_ABLE], lvTextCaps);

    AddListViewItem('Strikeout characters',
      NoYesArray[(TCaps and TC_SO_ABLE) = TC_SO_ABLE], lvTextCaps);

    AddListViewItem('Raster fonts',
      NoYesArray[(TCaps and TC_RA_ABLE) = TC_RA_ABLE], lvTextCaps);

    AddListViewItem('Vector fonts',
      NoYesArray[(TCaps and TC_VA_ABLE) = TC_VA_ABLE], lvTextCaps);

    AddListViewItem('Scrolling using bit-block transfer',
      NoYesArray[(TCaps and TC_SCROLLBLT) = TC_SCROLLBLT], lvTextCaps);
  end;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
```

continues

LISTING 10.7 Continued

```
  // Store the printer names in the combo box.
  cbPrinters.Items.Assign(Printer.Printers);
  // Display the default printer in the combo box.
  cbPrinters.ItemIndex := Printer.PrinterIndex;
  // Invoke the combo's OnChange event
  cbPrintersChange(nil);
end;

procedure TMainForm.cbPrintersChange(Sender: TObject);
begin
  Screen.Cursor := crHourGlass;
  try
    // Populate combo with available printers
    Printer.PrinterIndex := cbPrinters.ItemIndex;
    with Printer do
      GetPrinter(Device, Driver, Port, ADevMode);
    // Fill the general page with printer information
    with lvGeneralData.Items do
    begin
      Clear;
      AddListViewItem('Port', Port, lvGeneralData);
      AddListViewItem('Device', Device, lvGeneralData);

      Rslt := DeviceCapabilitiesA(Device, Port, DC_DRIVER, nil, nil);
      AddListViewItem('Driver Version', IntToStr(Rslt), lvGeneralData);
    end;

    // The functions below make use of the GetDeviceCapabilitiesA function.
    GetBinNames;
    GetDuplexSupport;
    GetCopies;
    GetEMFStatus;
    GetResolutions;
    GetTrueTypeInfo;

    // The functions below make use of the GetDeviceCaps function.
    GetDevCapsPaperNames;
    GetDevCaps;    // Fill Device Caps page.
    GetRasterCaps; // Fill Raster Caps page.
    GetCurveCaps;  // Fill Curve Caps page.
    GetLineCaps;   // Fill Line Caps page.
    GetPolyCaps;   // Fill Polygonal Caps page.
    GetTextCaps;   // Fill Text Caps page.
  finally
```

```
    Screen.Cursor := crDefault;
  end;
end;

end.
```

Summary

This chapter teaches the techniques you need to know in order to program any type of custom printing, from simple printing to more advanced techniques. You also learned a methodology that you can apply to any printing task. Additionally, you learned about the `TDeviceMode` structure and how to perform common printing tasks. You'll use more of this knowledge in upcoming chapters, where you build even more powerful printing methods.