Printing In Delphi: Introducting TPrinter

by Xavier Pacheco

This article is the first of a series which will discuss printing in Delphi 2. This time I'll introduce you to the `TPrinter` class and illustrate some simple techniques for printing text, rich text formatted text and bitmap images. The examples presented will show you how to use some of `TPrinter`'s methods and properties and will lay the groundwork for the upcoming articles, where we'll discover how to use more complex printing techniques.

The TPrinter Object

Delphi 2's `TPrinter` object encapsulates the Windows printing interface and simplifies many of the printing management tasks that you would otherwise have to code yourself. Like a `TForm`, `TPrinter` has a `Canvas` property that represents its drawing surface. It is to the printer's `Canvas` that you draw text and graphics, just like you would draw to a form's `Canvas`. The difference is that you are drawing to printed output as opposed to screen output and therefore you must take into account certain factors.

Firstly, usually, you can easily erase something that you've drawn to the screen. When you draw to the printer's `Canvas`, on the other hand, it gets sent to the printer and you have no way of erasing what's been printed on paper.

Secondly, drawing to the printer is substantially slower then drawing to the screen. This is true even on some high-performance laser printers. You should provide a way for your users to cancel a lengthy print job.

Thirdly, you can assume that your users' displays support graphics output since they are already running Windows. You can't assume the same of their printers. Some printers may not support graphics output at all. Your printing routines may have to test for this case.

Lastly, screen and printer resolutions differ drastically. Therefore, what you see on screen won't look the same on the printer unless you handle the resolution difference. For example, a 300x300 pixel bitmap might look spectacular on a 640x480 pixel display, but on a 300dpi laser printer it's only a 1 inch by 1 inch square blob. You must make resolution adjustments to your drawing routines so that your users will not need a magnifying glass to read their printed output!

The global `TPrinter` object is accessed through the `Printer` function which is defined in the `Printers` unit. In Delphi 1, `Printer` was a global `TPrinter` variable. If you have Delphi 1 code that references the `Printer` global variable, it should compile without any errors.

Property	Purpose
Aborted	Returns `True` if the current print job has been terminated – the `Printer.Abort` procedure has been called.
Canvas	A `TCanvas` representing the printer's printable surface. The `TCanvas` drawing functions can be used to draw text and graphics to the printed page. The printer may not support some of the `Canvas`'s functions, such as `StretchDraw`, `Draw` and `CopyRect`.
Capabilities	Represents the capabilities for the currently selected printer. This property is a `set` of `TPrinterCapability` defined as `TPrinterCapability = (pcCopies, pcOrientation, pcCollation)`
Copies	Specifies the number of copies to the printer of the current print job.
Fonts	Specifies a list of fonts supported by the currently selected printer.
Handle	Represents the handle of the printer device.
Orientation	Determines if the printer prints horizontally or vertically. This value can be either `poPortrait` (vertical printing) or `prLandcape` (horizontal printing).
PageHeight	Height of the currently printing page, in pixels.
PageWidth	Width of the currently printing page, in pixels.
PageNumber	Page number of the currently printing page. This number is incremented with each call to `Printer.NewPage`.
PrinterIndex	The selected printer from the list of printers as kept in the `Printers` property. A value of -1 represents the default printer.
Printing	Specifies whether or not the printer is currently in a print job.
Printers	A list of installed printers.
Title	The text that appears on the Print Manager and on network header pages.

➤ *Table 1: TPrinter properties*

The one exception to this rule is if you have code that assigned to the Printer object. If this is the case, the Printers unit defines a function SetPrinter which takes a new TPrinter object as a parameter and returns the old TPrinter instance. It is then up to you to free the old TPrinter instance.

Table 1 lists the various TPrinter properties and explains their purpose. Table 2 lists the various TPrinter methods and explains their purpose.

The following sections illustrate how easy it is to output text, rich text and graphics images to your printer. These are probably the most common types of printed output.

Printing Text

Sending text to the printer is just as easy as sending text to a file. In fact, the process is practically the same. Before you can print, however, you must be sure to add the Printers unit to your unit's uses clause. Listing 1 illustrates how to print a line of text.

This code uses the AssignPrn procedure which is declared in the printers unit as:

```
procedure AssignPrn(
  var F: Text)
```

You will notice that AssignPrn is much like the Assign procedure for a file. AssignPrn, however, associates a text file variable with the printer device. Any subsequent Writeln or Write calls on that text file variable will go to the printer device. When you are done printing, you must call CloseFile on the text file variable.

Although this method of printing doesn't let you get too elaborate with your printed output, it is adequate and very simple to use when you just need to print text. As an example, Listing 2 prints the contents of a TMemo component. Additionally, the code sets the printer's font to that of the TMemo.

The code in Listing 2 uses a simple for..do loop to print the contents of the TMemo. One of the nice features of this technique is that a new page will be started if a line to be printed extends beyond the page height.

Method	Definition/Purpose
Abort	procedure Abort Terminates the current printer job. This method is used to halt the print job before normal termination, which is done using the EndDoc method. After calling Abort, the printer is ready for the next print job to begin.
BeginDoc	procedure Begindoc Starts a print job. To successfully end a print job, EndDoc should be called.
EndDoc	procedure EndDoc Ends the current printer job. Once this method is called the printer will begin to print. EndDoc is used when the print job has ended successfully, otherwise Abort should be called to terminate an unsuccessful print job.
NewPage	procedure NewPage Starts printing on a new page. NewPage also increments the PageNumber property and sets the Pen property back to position (0, 0) on the printer's Canvas.
GetPrinter	procedure GetPrinter(ADevice, ADriver, APort: PChar; var ADeviceMode: THandle) Returns the device name, driver name and port information as null-terminated strings. Also returns a handle to the printer's TDeviceMode structure, which is documented as TDEVMODE in Delphi's online help.
SetPrinter	procedure SetPrinter(ADevice, ADriver, APort: PChar; ADeviceMode: THandle) Sets the printer specified by the ADevice, ADriver and APort variables as the current printer. ADeviceMode is a handle to a TDeviceMode structure that allows you to specify various printer settings. See TDEVMODE in Delphi's online help for information on this structure.

➤ *Table 2: TPrinter methods*

```
procedure TForm1.Print1Click(Sender: TObject);
var
  PrintText: TextFile;    // Declare a text file variable
begin
  AssignPrn(PrintText);   // Assign the text file variable to the printer
  Rewrite(PrintText);     // Open the printer for output
  try
    Writeln(PrintText, 'Delphi is GREAT!'); // Write text to the printer
  finally
    CloseFile(PrintText);  // Close the variable associated with the printer
  end;
end;
```

➤ *Listing 1*

Printing Rich Text Format

Printing rich text formatted text is ridiculously simple. It amounts to one line of code when using the TRichEdit component:

```
RichEdit1.Print(
  'Rich Edit Text');
  { Extremely difficult
    line of code <g> }
```

The TRichEdit.Print method is declared as:

```
procedure Print(
  const Caption: string);
```

The Caption is the title of the printed document.

Printing Bitmaps

Printing bitmap images isn't too difficult. One thing you'll have to remember is that the resolutions of

```
procedure TForm1.Print1Click(Sender: TObject);
var  PrintText: TextFile;      // Declare a text file variable
     i: integer;
begin
  if Memo1.Lines.Count  0 then begin
    Printer.Canvas.Font := Memo1.Font;  // Match fonts.
    AssignPrn(PrintText);   // Assign the text file variable to the printer
    Rewrite(PrintText);     // Open the printer for output
    try
      { Write Memo1's contents to the printer }
      for i := 0 to Memo1.Lines.Count - 1 do
        Writeln(PrintText, Memo1.Lines[i]);
    finally
      Closefile(PrintText);  // Close variable associated with printer
    end;
  end;
end;
```

➤ *Above: Listing 2*

➤ *Below: Listing 3*

```
procedure TForm1.Print1Click(Sender: TObject);
var
  InfoSize: Integer;    { used to determine size of memory to allocate for }
                        { a TBitmapInfo structure }
  ImageSize: Integer;   { Used to determine size of memory to allocate for }
                        { bitmap bits }
  Info: PBitmapInfo;    { Pointer to a TBitmapInfo structure which         }
                        { contains information on the dimensions and color }
                        { of a  Windows device independent bitmap          }
  Image: Pointer;     { Pointer to the DIB bits which is an array of bytes }
  ImWidth, ImHeight: Integer;  { Used for calculating size of image on the }
                               { destination canvas                        }
begin
  with Image1.Picture.Bitmap do begin
    { Call GetDIBSizes which returns the amount of memory needed to
      allocate for both the DIB info header and the DIB bitmap bits }
    GetDIBSizes(Handle, InfoSize, ImageSize);
    { Allocate memory for the info header based on the size obtained from
      GetDIBSizes }
    Info := MemAlloc(InfoSize);
    try
      { Allocate memory for the Image based on the size from GetDIBSizes }
      Image := MemAlloc(ImageSize);
      try
        { Retrieve the color palette information, the info header and the
          bitmap bits with the GetDIB procedure }
        GetDIB(Handle, Palette, Info^, Image^);
        with Info^.bmiHeader do begin
          Printer.BeginDoc;           // Start a print job
          try
            { Calculate the size of the output rectangle to which the
              image will be drawn. This will be based on one-half of the
              printer's page width }
            ImWidth := Printer.PageWidth div 2;
            ImHeight := trunc((ImWidth / biWidth) * biHeight);
            { Draw the information from the source bitmap to the
              destination device context, which is the printer's canvas }
            StretchDIBits(Printer.Canvas.Handle, 0, 0, ImWidth,
              ImHeight, 0, 0, biWidth, biHeight, Image, Info^,
              DIB_RGB_COLORS, SRCCOPY);
          finally
            Printer.EndDoc;           // End the print job
          end;
        end;
      finally
        FreeMem(Image, ImageSize);    // Free the allocated memory
      end;
    finally
      FreeMem(Info, InfoSize);        // Free the allocated memory
    end;
  end;
end;
```

the screen and printer devices differ. Therefore, you'll need to stretch the image that you are drawing so that it will fit on the printed page. The code in Listing 3 illustrates how you can draw an image using the `StretchDIBits` Win32 API function, which will work with both device-independent and device-dependent bitmaps. The listing illustrates how you would printing a bitmap so that it will fit to the printer's page size. The explanation of this code is embedded in the comments.

This function uses the `BeginDoc` procedure to start a print job. The `EndDoc` procedure is placed in a `finally` block to ensure that it is called once the print job is complete.

The `StretchDIBits` function copies the colors from a rectangular portion of a device independent bitmap to a rectangle in a destination device context, which is actually the printer's canvas. `StretchDIBits` takes several parameters. Listing 4 shows the definition of `StretchDIBits` and explains the purpose of its parameters. The comments in Listing 3 explain the usage of this function.

If you are drawing a device-dependent bitmap you could use the `Printer.Canvas.StretchDraw` method instead. However, you should be aware that using this method against a device-independent bitmap will cause the printout to lose the color information. Listing 5 shows a simple procedure which uses the method `Printer.Canvas.StretchDraw` to draw a device-dependent bitmap.

A word of caution on the code in Listing 5. One of the common problems that Borland's Developer Support department gets calls on is that this code, or rather, the `StretchDraw` method fails on some printers. This is not the fault of the code or of the `StretchDraw` itself. This is more due to deficiencies with whatever printer driver you happen to be using. It is therefore advisable to use the code presented in Listing 3.

To further this caution, you should also know that even Listing 3 may not work as expected because some printer drivers will not fill the palette entries for the bitmap correctly. There is always the possibility to create your own palette entries, but that's another topic altogether. The bottom line is that your printing code very much relies on the various printer drivers which most likely aren't bug free. If you suspect a bug in your printing code, it might be a good idea to try it on several systems

before spending hours (or days even) debugging your work.

Conclusion

This concludes this introduction to Delphi 2 printing. Next month, we'll look at performing complex printing where you'll examine how to print items to scale. The examples we'll examine next month will illustrate how to write print procedures and functions that use more common units of measurement like inches and centimeters instead of pixels. This will allow you to achieve a more accurate representation with your printed output.

Xavier Pacheco is a Field Consulting Engineer with Borland International and co-author of *Delphi 2.0 Developer's Guide* which should be out around June. You can send him your home brewing recipes by email to xpacheco@wpo.borland.com or on CompuServe at 76711,666

```
function StretchDIBits(
  DC: HDC;      // Device context handle
  DestX,        // Upper left x-coordinate of the destination rect
  DestY,        // Upper left y-coordinate of the destination rect
  DestWidth,    // Width of the destination rectangle
  DestHegiht,   // Height of the destination rectangle
  SrcX,         // Upper left x-coordinate of the source rect
  SrcY,         // Upper left y-coordinate of the source rect
  SrcWidth,     // Width of the source rectangle
  SrcHeight: Integer;   // Height of the source rectangle
  Bits: Pointer;        // Address of the bitmap bits
  var BitsInfo: TBitmapInfo;    // Bitmap info structure
  Usage: UINT;          // Contains either DIB_PAL_COLORS or DIB_RGB_COLORS
  Rop: DWORD            // Copy operation of source and destination
): Integer;
```

➤ *Listing 4*

```
procedure TForm1.Print21Click(Sender: TObject);
var
  ImWidth, ImHeight: Integer;
begin
  Printer.BeginDoc;  // Start a print job
  try
   { Calculate the image size for the destination canvas }
   ImWidth := Printer.PageWidth div 2;
   ImHeight := trunc((ImWidth / Image1.Width) * Image1.Height);
   { Print the bitmap }
    Printer.Canvas.StretchDraw(Rect(0, 0, ImWidth, ImHeight),
       Image1.Picture.Graphic);
  finally
    Printer.EndDoc; // End the print job
  end;
end;
```

➤ *Listing 5*