Creating PDF Documents

It is common for applications to allow users to archive their data by printing it. However, it may also be useful to archive data digitally, and the best way to do this is to store them as PDF documents because the PDF (Adobe Portable Document Format) file format is designed to be portable among platforms and operating systems.

Document Structure

For instance, to convert text to an HTML page, you need to write a fairly simple piece of code that generates the basic HTML skeleton and then goes through a string list appending <br> for line breaks and <p> for paragraphs.

Creating a PDF document is much more difficult because a PDF document consists of a collection of objects that describe various pieces of the document: a page, a page's contents, a font, etc.

Essentially, a PDF document consists of the following four parts:

- The document header, which specifies the document version (PDF specification conformance)
- The document itself (the document body), which contains various objects that define the document and its contents
- The cross-reference table, which is a table of objects present in the document. The cross-reference table allows applications to access any object in the document without having to read the entire file. As a result, applications can easily manage both small and large documents, even ones that contain thousands of pages.
- The document trailer, which points to the location of the cross-reference table. The document trailer and the cross-reference table are very important parts of PDF documents because PDF documents are meant to be read from the end, through the document trailer and the cross- reference table.

To learn more about the structure of PDF documents, you should take a look at the source of a simple PDF document. Listing 27-3 shows the source of a very simple PDF document, and Figure 27-16 shows what the document looks like in a PDF reader application.

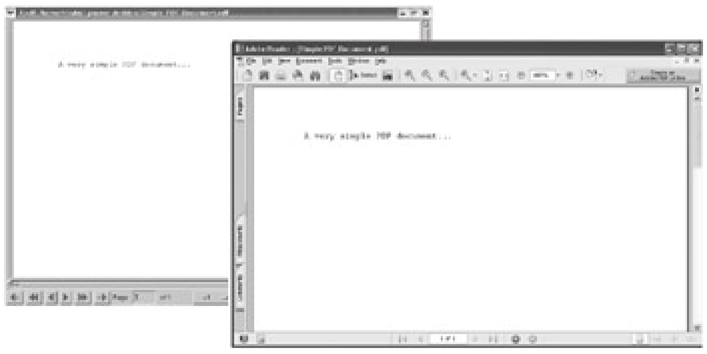


Figure 27-16: The above PDF document in Linux and Windows

The document header is the %PDF-1.2 line. Everything between the first line and the line that contains only the PDF reserved word `xref` is the document body. The cross-reference table begins with the reserved word `xref` and ends with the reserved word `trailer`. Not surprisingly, the document trailer begins with the reserved word `trailer` and ends with the end of file marker, %%EOF.

Listing 27-3: A simple PDF document

```
%PDF-1.2
1 0 obj
<<
/Author (Santa Claus)
/Producer (Inside Delphi Book - TPDFExport Component)
/Subject ()
/Title (Christmas Presents)
>>
endobj
2 0 obj
<< /Type /Font
/Subtype /Type1
/Name /F1
```

```
/BaseFont /Courier
/Encoding /WinAnsiEncoding
>>
endobj
3 0 obj
<< /ProcSet [ /PDF /Text]
/Font <<
/F1 2 0 R
>>
>>
endobj
4 0 obj
<< /Length 76
>>
stream
BT
/F1 12 Tf
72 720 Td
(A very simple PDF document...) Tj
0 -12 TD
ET
endstream
endobj
5 0 obj
<< /Type /Pages
/Kids [
6 0 R
]
/Count 1
>>
endobj
6 0 obj
<< /Type /Page
/Parent 5 0 R
/MediaBox [0 0 612 792]
/Contents 4 0 R
/Resources 3 0 R
>>
endobj
7 0 obj
<< /Type /Catalog
/Pages 5 0 R
>>
endobj
xref
0 8
0000000000 65535 f
0000000010 00000 n
0000000156 00000 n
0000000269 00000 n
0000000342 00000 n
0000000473 00000 n
0000000540 00000 n
0000000652 00000 n
trailer
<< /Size 9
/Root 7 0 R
/Info 1 0 R
>>
startxref
706
%%EOF
```

Creating the TPDFExport Component

The first thing that you need to do is derive the TPDFExport component from TComponent and add it to your package. Once you've created the component, we should override the constructor and destructor and introduce several basic properties, like PageHeight and PageWidth.

In this component, we are going to use the TMemoryStream class to store the PDF document, although the TStringList class can also be used, because, as you've already seen, a PDF document can be represented as text. While a TStringList-oriented implementation would be a bit easier to write, we would lose the ability to compress pieces of the PDF document using the zlib/flate compression.

To successfully create a PDF document, we need to create and use three objects: the TMemoryStream object for storing the PDF document and two string lists. One string list will be used for the component's Strings property, so that we can add text to TPDFExport component at design time (and run time). The second string list must be created because we have to build the cross-reference table while we're generating other pieces of the document. The cross-reference table must contain byte offsets of every object found in the document, and the best way to construct the table is to have it available at all times and add the objects' offsets to the table as soon as we create them.

Besides the three objects, let's define four basic properties: Compress, FontSize, PageWidth, and PageHeight. The Compress property allows the user to select whether to compress the contents of the document using zlib compression or to leave the contents uncompressed.

Even though PDF documents are able to use any number of fonts, the TPDFExport component only uses the Courier New font, simply to help you understand how to work with fonts and because we'll only be generating documents that contain plain text. The only thing the TPDFExport component allows you to do with the font is to change its size.

Every page in a PDF document can have its own size, but again, to keep things simple, TPDFExport uses only one size, specified in the PageHeight and PageWidth properties, for all pages in the document. Default values of 792 for PageHeight and 612 for PageWidth are expressed in units called points, where 72 points equals one inch, and define an 8.5 x 11-inch page (lettersize). Table 27-1 shows typical page sizes in centimeters, inches, and points.

Table 27-1: Letter and A4 page sizes

Paper Type/Size	**Centimeters**	**Inches (cm / 2.54)**	**Points (inches * 72**
Letter Width	21.59	8.5	612
Letter Height	27.94	11	792
A4 Width	21	8.27	596
A4 Height	29.7	11.7	842

Listing 27-4 shows the TPDFExport component's constructor and destructor and the abovementioned four properties.

Listing 27-4: TPDFExport component, the beginning

```
unit PDFExport;

interface

uses
  SysUtils, Classes;

type
  TPDFExport = class(TComponent)
  private
    { Private declarations }
    PDF: TMemoryStream;
    FOffsetList: TStringList;
    FStrings: TStrings;
    FPageWidth: Integer;
    FPageHeight: Integer;
    FFontSize: Integer;
    FCompress: Boolean;
  protected
    { Protected declarations }
  public
    { Public declarations }
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    { Published declarations }
    property Compress: Boolean
      read FCompress write FCompress default False;
    property FontSize: Integer
      read FFontSize write FFontSize default 12;
    property PageHeight: Integer
      read FPageHeight write FPageHeight default 792;
    property PageWidth: Integer
      read FPageWidth write FPageWidth default 612;
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('My Components', [TPDFExport]);
end;

constructor TPDFExport.Create(AOwner: TComponent);
begin
  inherited;
  PDF := TMemoryStream.Create;
  FOffsetList := TStringList.Create;
  FStrings := TStringList.Create;

  FPageWidth := 612;
  FPageHeight := 792;
  FFontSize := 12;
  FCompress := False;
end;

destructor TPDFExport.Destroy;
begin
  FStrings.Free;
  FOffsetList.Free;
  PDF.Free;
  inherited;
end;

end.
```

You've probably noticed that the way the inherited constructor and destructor are called in TPDFExport is different from the way we've been calling inherited constructors in previous components. When you only write inherited in a method, Delphi will call the ancestor method that has the same name and the same parameter list. In these two cases, the reserved word `inherited` will result in a call to TComponent.Create and TComponent.Destroy.

Utility Methods and the Document Header

The simplest part of the PDF document, and the easiest to create, is the document header:

```
%PDF-1.2
```

The TPDFExport component should conform to at least PDF 1.2, because this enables the user to compress data using zlib compression, which was introduced in PDF version 1.2.

Every line of text (the ones that contain PDF commands) must be terminated with the end of line marker: carriage return (CR, #13) and line feed (LF, #10) characters. If we were using the TStringList class to generate the PDF, things would be simpler because the TStringList class ends every string it contains with the abovementioned end of line marker. But since we're using the TMemoryStream class, we have to add the end of line marker to the stream manually.

To write strings to the memory stream and to automatically end them with the end of line marker, we need a simple method that will do the job for us. Here is the TPDFExport's overloaded Write method that allows us to write a string directly to the PDF or another stream:

```
type
  TPDFExport = class(TComponent)
  private
    procedure Write(const S: string); overload; inline;
    procedure Write(AStream: TStream; const S: string); overload;
  end;

procedure TPDFExport.Write(const S: string);
begin
  Write(PDF, S);
end;

procedure TPDFExport.Write(AStream: TStream; const S: string);
const
  EOLN = #13#10;
var
  pdfString: string;
begin
  pdfString := S + EOLN;
  AStream.Write(pdfString[1], Length(pdfString));
end;
```

Now that the Write method is done, we have to create two more methods: StartDocument and SaveToFile. The SaveToFile method is the most important method because it calls all other methods and actually generates the document. We also need an FCurrentObject private integer field to help us keep track of the object we're working with. For now, simply declare the field and set it to 0 in the StartDocument method.

Here are the StartDocument and SaveToFile methods:

```
type
  TPDFExport = class(TComponent)
  private
    FCurrentObject: Integer;
    procedure StartDocument;
  public
    procedure SaveToFile(const AFileName: string);
  end;

procedure TPDFExport.SaveToFile(const AFileName: string);
begin
  { 1. Write the document header. }
  StartDocument;

  { Finally, save the stream to disk. }
  PDF.SaveToFile(AFileName);
end;

procedure TPDFExport.StartDocument;
begin
  { remove the old pdf document from the stream
    and reset the cross-reference (offset) list }
  PDF.Clear;
  FOffsetList.Clear;
  FCurrentObject := 0;

  { Write the PDF document header. }
  Write('%PDF-1.2');
end;
```

Indirect Objects

Indirect objects in PDF are objects with a unique identifier (a positive integer) and a generation number. In new PDF documents, the generation number is 0, but in updated documents, the generation number can be larger than 0. In this component, we'll only be creating objects with a generation number of 0.

Indirect objects are extremely useful because they can reference each other in the document. As you'll see in a moment, almost everything we write to the PDF document will be an indirect object or an indirect object reference.

The syntax for creating an indirect object looks like this:

```
UniqueID GenerationNumber obj
  ObjectData
endobj
```

Here's a real indirect object from a PDF document:

```
1 0 obj
<<
/Author (Santa Claus)
/Producer (Inside Delphi Book - TPDFExport Component)
/Subject ()
/Title (Christmas Presents)
>>
endobj
```

Each time an indirect object is created, like the document information dictionary above, the TPDFExport component has to create its entry in the cross-reference table. To solve these two problems, let's create the StartObject function. The StartObject function should first add the offset of the new object to the reference list, acquire a unique number for the new object, and then write the object's header to the PDF stream.

Entries in the cross-reference table are strictly defined. Each entry in the table has to be 20 bytes long (18 bytes of data and the end of line marker). The format of the cross-reference table entry is:

```
nnnnnnnnnn ggggg n eoln
```

The nnnnnnnnnn part is a 10-digit byte offset, the ggggg part is the 5-digit generation number (which in the TPDFExport component is always 00000), and the n part is actually the letter "n". The letter "n" is used to specify that this object is in use. In PDF documents, you can also use an "f" letter here. The letter "f" specifies that the location is free, but we aren't going to be using or creating free entries in the TPDFExport component.

The Format function can help us generate the required 10-digit offset string. When you pass a %10d format string to the Format function, the function will convert the number into a 10-character string. However, the function will fill the string with spaces, so we need to change the spaces into zeros after the Format call. The FormatOffset function, which follows, is used by the TPDFExport component to generate the 10-digit object offset:

```
function FormatOffset(Index: Integer): string;
var
  i: Integer;
begin
  Result := Format('%10d', [Index]);
  { replace spaces with zeros }
  for i := 1 to Length(Result) do
    if Result[i] = ' '  then Result[i] := '0';
end;
```

Now that the biggest problem is solved, we can create the StartObject and EndObject methods. The TPDFExport component uses these two methods and the Write method to generate the PDF document. Note that the EndObject procedure only outputs the `endobj` reserved word to the PDF stream and that its only purpose beyond outputting the reserved word is to improve the readability of the source code that generates PDF objects.

Listing 27-5: StartObject and EndObject methods

```
type
  TPDFExport = class(TComponent)
  private
    function StartObject: Integer;
    procedure EndObject;
  end;

function TPDFExport.StartObject: Integer;
const
  NEW_OBJECT = '%d 0 obj';
begin
  { Remember the offset of the object and
    automatically create its entry in the xref list }
  FOffsetList.Add(FormatOffset(PDF.Size) + ' 00000 n');

  { Increment object count and write the object's
    header to the PDF stream }
  Inc(FCurrentObject);
  Write(Format(NEW_OBJECT, [FCurrentObject]));

  { return the object's ID }
  Result := FCurrentObject;
end;

procedure TPDFExport.EndObject;
begin
  Write('endobj');
end;
```

Document Information Dictionary

The document information dictionary is one of the simplest objects you can create in a PDF. Even though this dictionary is completely optional, we're going to create it as the very first one because this dictionary allows us to add metadata to the PDF document, such as the document's title, the name of the author, etc. Dictionaries are sequences of key-value pairs enclosed in "<<" and ">>".

Again, here is what the document information dictionary looks like:

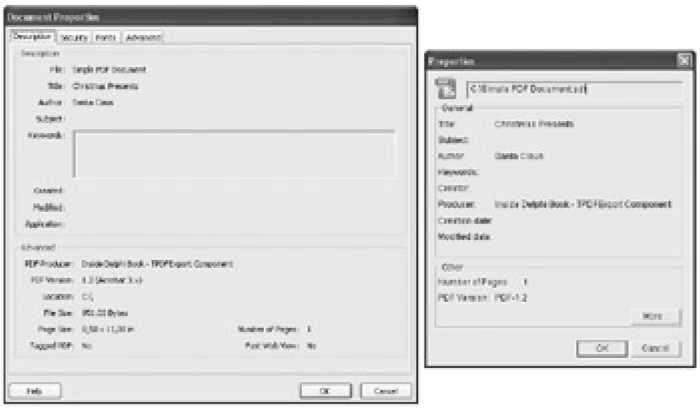
```
1 0 obj
<<
/Author (Santa Claus)
/Producer (Inside Delphi Book - TPDFExport Component)
/Subject ()
/Title (Christmas Presents)
>>
endobj
```

To allow the user to customize the PDF document's metadata, you need to at least create the Author, Producer, Subject, and Title string properties and use the StartObject, Write, and EndObject methods to generate the document information dictionary from this data. Listing 27-6 shows the CreateInfoObject method that generates the document information dictionary. To create the information dictionary as the first object, call the CreateInfoObject method in the SaveToFile method, immediately after the StartDocument method that generates the document header.

Listing 27-6: Generating the document information dictionary

```
type
  TPDFExport = class(TComponent)
  private
    FAuthor: string;
    FProducer: string;
    FSubject: string;
    FTitle: string;
    procedure CreateInfoObject;
  published
    property Author: string read FAuthor write FAuthor;
    property Producer: string read FProducer write FProducer;
    property Subject: string read FSubject write FSubject;
    property Title: string read FTitle write FTitle;
  end;

procedure TPDFExport.CreateInfoObject;
begin
  StartObject;
    Write('<<');
    Write('/Author (' + FAuthor + ')');
    Write('/Producer (' + FProducer + ')');
    Write('/Subject (' + FSubject + ')');
    Write('/Title (' + FTitle + ')');
    Write('>>');
  EndObject;
end;

procedure TPDFExport.SaveToFile(const AFileName: string);
begin
  { 1. Write the document header. }
  StartDocument;
  { 2. Create the info object, always 1 }
  CreateInfoObject;
  { Other objects will be generated here. }

  { Finally, save the stream to disk. }
  PDF.SaveToFile(AFileName);
end;
```

The metadata from the information dictionary can be viewed in any PDF reader, such as Adobe Reader or Foxit Reader (see Figure 27-17).



Document Properties

Description

File: Single PDF Document
Title: Christmas Presents
Author: Santa Claus
Subject:
Keywords:

Created:
Modified:
Application:

Advanced

PDF Producer: Inside Delphi Book - TPDFExport Component
PDF Version: 1.2 (Acrobat 3.x)
Location: C:\
File Size: 951.00 Bytes
Page Size: 8,50 x 11,00 in
Number of Pages: 1
Tagged PDF: No
Fast Web View: No

Properties

C:\Simple PDF Document.pdf

General

Title: Christmas Presents
Subject:
Author: Santa Claus
Keywords:
Creator:
Producer: Inside Delphi Book - TPDFExport Component
Creation date:
Modified date:

Other

Number of Pages: 1
PDF Version: PDF-1.2

Figure 27-17: Viewing document properties in Adobe Reader and Foxit Reader

Creating the Font Object

The PDF specification requires viewer applications to support 14 standard (Type 1) fonts. Thus, you can freely use these 14 fonts without having to worry about the document size or how the document will be displayed. PDFs can also use non-standard fonts. Applications that support non-standard fonts embed them in the document, which increases the size of the PDF document but enables it to be displayed on machines that don't have the specific font.

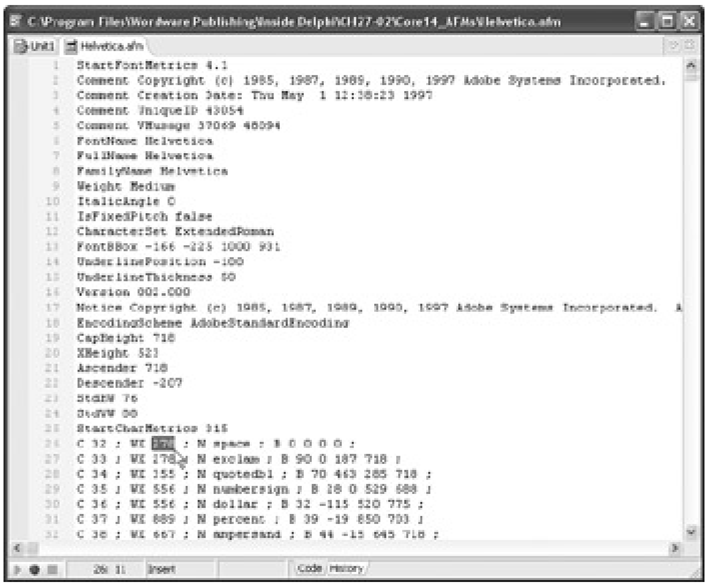
The following table shows the 14 standard fonts that can be safely used in PDF documents.

Table 27-2: Standard fonts

Times-Roman	Helvetica	Courier	Symbol
Times-Bold	Helvetica-Bold	Courier-Bold	Zapf-Dingbats
Times-Italic	Helvetica-Oblique	Courier-Oblique	
Times-BoldItalic	Helvetica-BoldOblique	Courier-BoldOblique	

The TPDFExport component uses only one font — Courier — for several reasons:

- Since we are only working with plain text files, we don't need to use more than one font.
- Courier is one of the standard fonts, which means that every PDF document created by the component will be small and will look almost the same on different operating systems and in different viewers.
- Every character in the Courier font has the same width, which allows us to create simple algorithms for text output. Other standard fonts, not to mention non-standard ones, have variable character widths. To find out about character widths in other standard fonts, take a look at the Adobe font metrics (AFM) files (available on the companion CD). Figure 27-18 shows a piece of the Helvetica.afm file, which contains information about the Helvetica font. The WX column, in which the value 278 is highlighted, contains character widths of the Helvetica font.



```
StartFontMetrics 4.1
Comment Copyright (c) 1985, 1987, 1989, 1990, 1997 Adobe Systems Incorporated.
Comment Creation Date: Thu May  1 12:38:23 1997
Comment UniqueID 43054
Comment VMusage 37069 48094
FontName Helvetica
FullName Helvetica
FamilyName Helvetica
Weight Medium
ItalicAngle 0
IsFixedPitch false
CharacterSet ExtendedRoman
FontBBox -166 -225 1000 931
UnderlinePosition -100
UnderlineThickness 50
Version 002.000
Notice Copyright (c) 1985, 1987, 1989, 1990, 1997 Adobe Systems Incorporated.  A
EncodingScheme AdobeStandardEncoding
CapHeight 718
XHeight 523
Ascender 718
Descender -207
StdHW 76
StdVW 88
StartCharMetrics 315
C 32 ; WX 278 ; N space ; B 0 0 0 0 ;
C 33 ; WX 278 ; N exclam ; B 90 0 187 718 ;
C 34 ; WX 355 ; N quotedbl ; B 70 463 285 718 ;
C 35 ; WX 556 ; N numbersign ; B 28 0 529 688 ;
C 36 ; WX 556 ; N dollar ; B 32 -115 520 775 ;
C 37 ; WX 889 ; N percent ; B 39 -19 850 703 ;
C 38 ; WX 667 ; N ampersand ; B 44 -15 645 718 ;
```

Figure 27-18: The Helvetica font's font metrics file

To use a font in a PDF document, you have to describe it in a dictionary and then reference the font object when you need it. When creating a font object, you need to give the font a name (in this case the name of the font is F1, for font1), use the BaseFont key to specify the name of the font you want to use (in this case it's Courier), and optionally specify the encoding like MacRomanEncoding, MacExpertEncoding, or WinAnsiEncoding.

The following listing shows the CreateFontObject method that creates the Courier font object. In the TPDFExport component, the font object is assumed to be the second object.

Listing 27-7: Generating the font object

```
procedure TPDFExport.CreateFontObject;
begin
  StartObject;
    Write('<< /Type /Font');
    Write('/Subtype /Type1');
    Write('/Name /F1');
    Write('/BaseFont /Courier');
    Write('/Encoding /WinAnsiEncoding');
    Write('>>');
  EndObject;
end;

procedure TPDFExport.SaveToFile(const AFileName: string);
begin
  ...
  { 3. Create the font object, always 2 }
  CreateFontObject;
  ...
end;
```

The Resource Dictionary

To use the indirect font object we created earlier, we need to create a resource dictionary that will contain a reference to the font. The pages that use the font (and other resources) will then reference the resource dictionary.

A resource dictionary looks like this:

```
3 0 obj
<< /ProcSet [ /PDF /Text]
/Font <<
/F1 2 0 R
>>
>>
```

To create the resource dictionary in this component, you pretty much only have to output what you see above. The ProcSet key in the resource dictionary is an array that holds PostScript procedure sets, which are only used when the PDF document is sent to a PostScript output device to allow the device to understand PDF operators.

Besides the procedure sets, the above resource dictionary also contains a reference to the F1 font. The syntax for an object reference in PDF is ObjectID GenerationNumber and the letter "R" for reference.

Here's the TPDFExport component's CreateResourcesObject method that generates the resource dictionary:

```
{ The resource dictionary is always 3 in this component. }
procedure TPDFExport.CreateResourcesObject;
begin
  { The font object (F1) is always 2 in this component. }
  StartObject;
    Write('<< /ProcSet [ /PDF /Text]');
    Write('/Font <<');
    Write('/F1 2 0 R');
    Write('>>');
    Write('>>');
  EndObject;
end;
procedure TPDFExport.SaveToFile(const AFileName: string);
begin
  ...
  { 4. Create resources object, always 3 }
  CreateResourcesObject;
  ...
end;
```

Preparing Strings for Output

Before we can store text into a PDF document using the TPDFExport component, we need to finish the implementation of the Strings property. The write method of the Strings property must not blindly assign the strings it acquires to the Strings property. To reduce the amount of code that needs to be written to actually generate PDF output, the write method of the Strings property should also reformat the strings to fit inside the margins.

To find out how many characters fit inside the margins, we need to divide the space inside the margins by the character width. Since the character width in PDF is specified in units that are the thousandth part of the default unit, we need to divide the result by 1000 to get the number of characters.

Here's how the TPDFExport component determines how many characters fit inside the margins (this is an excerpt from the Strings property's write method):

```
var
  charCount: Integer;
begin
  { 144 are left and right one-inch margins }
  { 600 is the constants width of all Courier characters }
  charCount := Round((FPageWidth - 144) / (600 * FFontSize / 1000));
end;
```

Once the number of characters per line is known, the SetStrings method needs to search for delimiter characters that can't be directly written to the PDF: (,), \, and the tab character (#9). The (,), and \ characters need to be escaped with another \ character to create a sequence that can be written to the PDF stream. So, to display a left parenthesis, you need to output \(to the PDF stream, and to display the \ character, you need to output \\ to the PDF stream. The tab character (#9) should be replaced with \t, although not replacing the tab character wreaks much less havoc on the PDF than the above three characters do.

Listing 27-8 shows the SetStrings write method that reformats the strings while assigning them to the Strings property. The code is thoroughly commented so you should have no problems in figuring out what's going on.

Listing 27-8: Reformatting the strings in the component's Strings property

```
type
  TPDFExport = class(TComponent)
  private
    procedure SetStrings(Value: TStrings);
  published
    property Strings: TStrings read FStrings write SetStrings;
  end;

procedure TPDFExport.SetStrings(Value: TStrings);
var
  charCount: Integer;
  line: string;
  i: Integer;
  lastSpace: Integer;
  delimiter: Integer;
  lineFits: Boolean;
begin
  { 144 are left and right one-inch margins }
  { 600 is the constants width of all Courier characters }
  charCount := Round((FPageWidth - 144) / (600 * FFontSize / 1000));
  FStrings.Clear;

  for i := 0 to Pred(Value.Count) do
  begin
    if Value[i] = '' then
      FStrings.Add('')
    else begin
      line := Value[i] + ' ';
      { if line fits into margins, copy the entire line into
        FStrings, but after the treatment of delimiter characters }
      lineFits := Pred(Length(line)) <= charCount;

      { if one of the special characters is found, replace it
        with its escape sequence, which can be used in the string }
      delimiter := 1;
      while delimiter < Length(line) do
      begin
        if line[delimiter] in ['\', '(', ')'] then
        begin
          Insert('\', line, delimiter);
          Inc(delimiter);     // skip inserted char
        end else if line[delimiter] = #9 {Tab} then
        begin
          { remove tab }
          Delete(line, delimiter, 1);
          { insert the proper tab marker }
          Insert('\t', line, delimiter);
          Inc(delimiter);     // skip "\t"
        end;
        Inc(delimiter);
      end;                   // while delimiter

      { if the line fits into margins, copy the
        entire line and move to the next one }
      if lineFits = True then
      begin
        FStrings.Add(line);
        Continue;
      end;
      { if the line doesn't fit into the margins, try to
        find the last space and break the line there }
      while line <> '' do
      begin
        { start at max length char }
        lastSpace := charCount;
        while (lastSpace > 1) and
           (line[lastSpace] <> ' ') do Dec(lastSpace);

        { if there are no spaces, break the string at charCount }
        if lastSpace = 1 then
        begin
          FStrings.Add(Copy(line, 1, charCount));
          Delete(line, 1, charCount);
        end else
        begin
          { if space is found, break it at the last full word }
          FStrings.Add(Copy(line, 1, lastSpace - 1));
          Delete(line, 1, lastSpace);
        end;     // if lastSpace = 1
      end;       // while line <> ''
    end;         // the main if, if value[i] <> ''
  end;           // for i
end;
```

Generating Page Contents (Content Streams)

An interesting thing about PDF documents is that they store pages and page contents as separate objects. The TPDFExport component first generates all page contents objects and then creates all page objects with references to the appropriate contents stream.

Here is a very simple page contents object (content stream):

```
4 0 obj
<< /Length 76
>>
stream
BT
/F1 12 Tf
72 720 Td
(A very simple PDF document...) Tj
0 -12 TD
ET
endstream
endobj
```

Every stream has to have a Length entry that tells the viewer application how many bytes the stream contains. If the stream is compressed, Length should specify the compressed size. The length of the stream is the amount of data between the `stream` and `endstream` reserved words.

The BT operator in PDF documents is used to start text output. The end of text output is marked with the ET operator. Between the two operators are additional PDF commands that display text.

The /F1 12 Tf line shows how to use the Tf operator to select a font and define the font's size. In this case, this line tells the viewer application to use our F1 font and to display the text in size 12.

The 72 720 Td line marks the beginning of text output; these are the coordinates of the first line. The Td operator is a text-positioning operator used to specify where the text should be displayed. These two values result in text being displayed in the top-left corner of the page, next to the left margin (72) and beneath the top margin (720, page height – top margin (72)). The (A very simple PDF document...) Tj line uses the Tj operator to display the text in parentheses on the page. Now that you've seen that parentheses are used in PDF to contain strings, you understand why these characters had to be escaped with \ in the SetStrings write method.

The final line inside the stream, 0 –12 TD, uses the TD operator to offset text from the current location. In this case, the horizontal coordinate isn't modified (0). The –12 value moves text to the next line; actually it moves the text by FontSize toward the bottom of the page.

To create page contents objects, the TPDFExport component uses the CreatePageContents method that is able to create both uncompressed and zlib compresssed streams.

Compressing data in Delphi cannot be simpler. To compress a stream that contains uncompressed data, you have to use the TCompressionStream class declared in the ZLib unit. The actual compression of data occurs when you copy data from an uncompressed stream to the compression stream.

The constructor of the TCompressionStream class accepts two parameters: the level of compression and the destination stream where the compressed data will be written. The following excerpt shows how to compress a stream using the TCompressionStream class. In the excerpt, the buff variable is a TMemoryStream that contains uncompressed data, C is the TCompressionStream used to compress data, and destStream is another TMemoryStream whose purpose is to accept the compressed output of the TCompressionStream object:

```
 { create the dest stream to hold compressed data }
 destStream := TMemoryStream.Create;
 try
   C := TCompressionStream.Create(clDefault, destStream);
   try
    { compress data by copying it from another stream }
    C.CopyFrom(buff, 0);
   finally
    C.Free;
   end;       // C
finally
   destStream.Free;
end;          // destStream
```

Listing 27-9 shows the entire CreatePageContents method. Notice that the method has two out parameters (which are only used to output values from the method to the outside world). FirstObject returns the ID of the first page contents object and the ObjectCount parameter returns the number of page contents objects created. These two values are later used by the method that creates page objects.

Listing 27-9: Generating page contents objects

```
unit PDFExport;

interface

uses SysUtils, Classes, ZLib;

type
  TPDFExport = class(TComponent)
  private
    procedure CreatePageContents(out FirstObject,
      ObjectCount: Integer);
  end;

procedure TPDFExport.SaveToFile(const AFileName: string);
var
  contentsFirst: Integer;
  contentsCount: Integer;
begin
  { 5. Create contents objects }
  CreatePageContents(contentsFirst, contentsCount);
end;

procedure TPDFExport.CreatePageContents(out FirstObject,
  ObjectCount: Integer);
var
  i, j: Integer;
  linesPerPage: Integer;
  index: Integer;
  buff: TMemoryStream;
  destStream: TMemoryStream;
  C: TCompressionStream;
const
  CONST_LENGTH = '<< /Length %d';
begin
  { determine how many lines fit on a page by dividing
    the space between top and bottom margins by FontSize }
  { to get some white space and reduce the possibility of
    overlaping characters, increment the font size internally by 1 }
  { finally, because of the way PDF outputs text, we can fit
    two more lines inside the margins, so increment the result by 2 }
  linesPerPage := ((FPageHeight - 144) div (FFontSize + 1)) + 2;

  { return FirstObject and ObjectCount values because
    they are needed to create page objects later }
  FirstObject := Succ(FCurrentObject);
  { ObjectCount is actually page count }
  ObjectCount := Succ(FStrings.Count div linesPerPage);

  buff := TMemoryStream.Create;
  try
{ create the contents }
for i := 0 to ObjectCount - 1 do
```

```
begin
  buff.Clear;
  Write(buff, 'BT');
  Write(buff, '/F1 ' + IntToStr(FFontSize) + ' Tf');
  Write(buff, '72 ' + IntToStr(FPageHeight - 72) + ' Td');

  for j := 0 to linesPerPage - 1 do
  begin
    index := (i * linesPerPage) + j;
    if index > Pred(FStrings.Count) then Break;
    Write(buff, '(' + FStrings[index] + ') Tj');
    { move to next line, -FontSize }
    Write(buff, '0 -' + IntToStr(FFontSize + 1) + ' TD');
  end;         // for i

  Write(buff, 'ET'); { finish text output }

  { create the page }
  if FCompress then
  begin
    { create the dest stream to hold compressed data }
    destStream := TMemoryStream.Create;
    try
      C := TCompressionStream.Create(clDefault, destStream);
      try
        { compress data by copying it from another stream }
        C.CopyFrom(buff, 0);
      finally
        C.Free;
      end;      // C

      { Write the compressed object to the PDF stream }
      StartObject;
        Write(Format(CONST_LENGTH, [destStream.Size]));
        Write('/Filter [/FlateDecode]');
        Write('>>');
        Write('stream');
        destStream.SaveToStream(PDF);
        Write('endstream');
      EndObject;
   finally
      destStream.Free;
    end;        // destStream
  end else
  begin
    { Write uncompressed object to the PDF stream }
    StartObject;
      Write(Format(CONST_LENGTH, [buff.Size]));
      Write('>>');
      Write('stream');
      buff.SaveToStream(PDF);
      Write('endstream');
    EndObject;
      end;           // if FCompressed
    end;             // for i
  finally
    buff.Free;
  end;
end;
```

Creating the Page Tree and the Page Objects

The page tree is a very important object that contains the number of pages available in the document (the Count key) and the references to all document pages (the Kids array). The page tree is used by viewer applications to quickly access any page in the document, no matter how many pages there are.

Here's what a page tree object looks like in a document that has only one page (with ID 6):

```
5 0 obj
<< /Type /Pages
/Kids [
6 0 R
]
/Count 1
>>
endobj
```

Among other things, a page object needs to reference the page tree object, and because of that, the TPDFExport component first generates the page tree object (based on the count of page contents objects from the CreatePage- Contents method) and then creates the page objects.

Besides having to reference the page tree (Parent) object, a page object must have a MediaBox key that defines the page size, a Contents key that references the page contents object for the page, and a Resources key that references the resource dictionary.

Here's what a page object looks like:

```
6 0 obj
<< /Type /Page
/Parent 5 0 R
/MediaBox [0 0 612 792]
/Contents 4 0 R
/Resources 3 0 R
>>
endobj
```

The TPDFExport component uses the CreatePages method to generate both the page tree object and all document pages. To achieve this, the method has to accept the ID of the first page contents object and the number of page contents objects available in the document (both values are returned by the CreatePageContents method). The method must also return the ID of the page tree object because it's needed later. Listing 27-10 shows the CreatePages method.

Listing 27-10: Generating the page tree and the page objects

```
procedure TPDFExport.SaveToFile(const AFileName: string);
var
  PageTreeID: Integer;
begin
   { 6. Create page tree and actual page objects }
  CreatePages(contentsFirst, contentsCount, PageTreeID);
end;


procedure TPDFExport.CreatePages(FirstRef,
  CountRef: Integer; out PageTreeID: Integer);
var
  i: Integer;
begin
  { First create the page tree object }
  PageTreeID := StartObject;
    Write('<< /Type /Pages');
    Write('/Kids [');

    for i := 1 to CountRef do
      Write(IntToStr(PageTreeID + i) + ' 0  R');

    Write(']');
    Write('/Count ' + IntToStr(CountRef)); { page count }
    Write('>>');
  EndObject;

  { Then create page objects }
  for i := 0 to CountRef - 1 do
  begin
    StartObject;
      Write('<< /Type /Page');
      Write('/Parent ' + IntToStr(PageTreeID) + ' 0  R');
      Write('/MediaBox [0 0 ' +
        IntToStr(FPageWidth) + ' ' + IntToStr(PageHeight) + ']');
      Write('/Contents ' + IntToStr(FirstRef + i) + ' 0  R');

      { Resource object is always 3 in this component. }
      Write('/Resources 3 0 R');
      Write('>>');
    EndObject;
  end;       // for i
end;
```

Creating the Document Catalog

The document catalog is the root object whose main purpose is to reference other objects that define the contents of the document. In this case, the document catalog only needs to reference the page tree object.

Here's what a document catalog looks like:

```
7 0 obj
<< /Type /Catalog
/Pages 5 0 R
>>
endobj
```

To generate the document catalog, the CreateCatalog method needs to accept the ID of the page tree object. The CreateCatalog method also needs to return the document catalog's ID:

```
procedure TPDFExport.SaveToFile(const AFileName: string);
var
  CatalogID: Integer;
begin
  { 7. Create the catalog object }
  CatalogID := CreateCatalog(PageTreeID);
end;

function TPDFExport.CreateCatalog(PageTreeRef: Integer): Integer;
begin
  Result := StartObject;
    Write('<< /Type /Catalog');
    Write('/Pages ' + IntToStr(PageTreeRef) + ' 0 R');
    Write('>>');
  EndObject;
end;
```

The Cross-Reference Table and the File Trailer

At this point, it's extremely easy to finish the cross-reference table because most of it has already been constructed with every call to the StartObject method. To finish the cross-reference table, we need to output the `xref` reserved word to the PDF stream. The `xref` reserved word has to be followed by two numbers: the ID of the first object (0) and the number of objects in the table. The ID of the first object in the following cross-reference table is 0, and there are eight objects in the table. The "0000000000 65535 f" entry is an obligatory placeholder for other entries:

```
xref
0 8
0000000000 65535 f
0000000010 00000 n
0000000156 00000 n
0000000269 00000 n
0000000342 00000 n
0000000473 00000 n
0000000540 00000 n
0000000652 00000 n
```

The file trailer starts with the trailer dictionary and enables viewer applications to quickly find the cross-reference table and objects like the document catalog. The trailer dictionary should have a Size key that specifies the total number of entries in the xref table + 1. It also must point to the document catalog and optionally to the info object, if there is one. The trailer dictionary has to be followed by the `startxref` reserved word, which must be followed by a number that is the byte offset of the xref table. The only thing that should follow this byte offset is the end of file marker, %%EOF.

Here's what a file trailer looks like:

```
trailer
<< /Size 9
/Root 7 0 R
/Info 1 0 R
>>
startxref
706
%%EOF
```

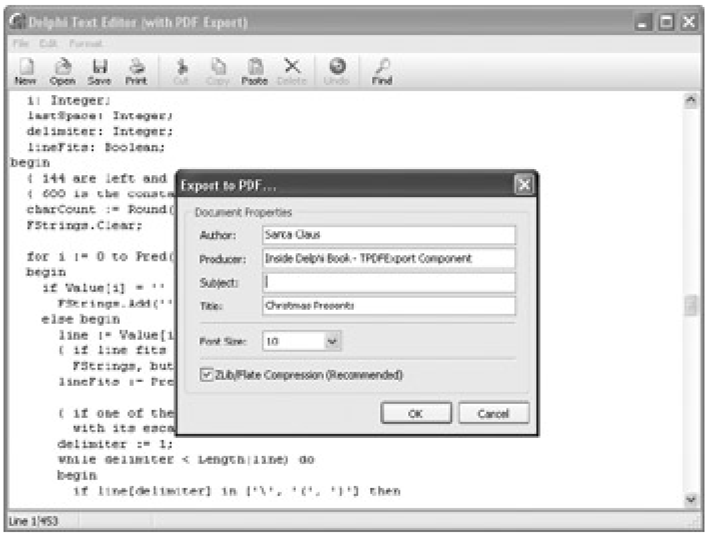
Listing 27-11 shows the EndDocument method that generates both the cross-reference table and the file trailer.

Listing 27-11: Generating the cross-reference table and the file trailer

```
procedure TPDFExport.SaveToFile(const AFileName: string);
begin
  { 8. Write the cross-reference table and the trailer }
  EndDocument(CatalogID);
end;

procedure TPDFExport.EndDocument(CatalogRef: Integer);
var
  xrefOffset: Integer;
  refCount: Integer;
begin
  xrefOffset := PDF.Size;
  Write('xref');

  // refCount = objectCount + the constant xref entry
  refCount := Succ(FCurrentObject);
  Write('0 ' + IntToStr(refCount));
  Write('0000000000 65535 f');
  FOffsetList.SaveToStream(PDF);

  // Write the trailer
  Write('trailer');
  Write('<< /Size ' + IntToStr(refCount + 1));
  Write('/Root ' + IntToStr(CatalogRef) + ' 0  R');
  Write('/Info 1 0 R');
  Write('>>');
  Write('startxref');
  Write(IntToStr(xrefOffset));
  Write('%%EOF');
end;
```

Testing the Component

The TPDFExport component is really easy to use. The only thing you have to remember is to set all component properties like page height, page width, and font size before assigning strings to the Strings property. Listing 27-12 shows an example of how to use the TPDFExport component. The code first displays a simple dialog box (see Figure 27-19) that allows the user to customize the PDF document and then generates the document based on the user's settings.


Delphi Text Editor (with PDF Export)
File Edit Format
New Open Save Print Cut Copy Paste Delete Undo Find
Export to PDF...
OK
Cancel
Line 1/453

```
  i: Integer;
  lastSpace: Integer;
  delimiter: Integer;
  lineFits: Boolean;
begin
  { 144 are left and
  { 600 is the consta
  charCount := Round(
  FStrings.Clear;

  for i := 0 to Pred(
  begin
    if Value[i] = ''
      FStrings.Add(''
    else begin
      line := Value[i
      { if line fits
        FStrings, but
      lineFits := Pre

      { if one of the
        with its esca
      delimiter := 1;
      while delimiter < Length(line) do
      begin
        if line[delimiter] in ['\', '(', ')'] then
```

Document Properties

Author: Santa Claus

Producer: Inside Delphi Book - TPDFExport Component

Subject:

Title: Christmas Presents

Font Size: 10

[✓] ZLib/Flate Compression (Recommended)

Figure 27-19: A dialog box that allows the user to customize PDF output

Listing 27-12: Using the TPDFExport component to save TMemo text to PDF

```
procedure TMainForm.ExportPDFActionExecute(Sender: TObject);
var
  origWordWrap: Boolean;
begin
  PDFForm := TPDFForm.Create(Self);
  try
    if PDFForm.ShowModal = mrOK then
    begin
      origWordWrap := Editor.WordWrap;
      try
        Editor.WordWrap := False;
        if PDFSaveDialog.Execute then
        begin
          PDF.Title := PDFForm.TitleEdit.Text;
          PDF.Author := PDFForm.AuthorEdit.Text;
          PDF.Producer := PDFForm.ProducerEdit.Text;
          PDF.Subject := PDFForm.SubjectEdit.Text;
          PDF.Title := PDFForm.TitleEdit.Text;
          PDF.Compress := PDFForm.CompressLabel.Checked;
          PDF.FontSize := StrToInt(PDFForm.FontCombo.Text);

          PDF.Strings := Editor.Lines;
          PDF.SaveToFile(PDFSaveDialog.FileName);
        end;    // if PDFSaveDialog
      finally
        Editor.WordWrap := origWordWrap;
      end;        // try..finally (Editor.WordWrap)
    end;          // if ShowModal
  finally
    PDFForm.Free;
  end;           // try..finally (PDFForm)
end;
```