PDF File Writer C# Class Library (Version 1.27.0)

Uzi Granot

9 Sep 2020 CPOL

PDF File Writer is a C# .NET class library allowing applications to create PDF files. Latest update is support for Metadata and QR Code ECI Assignment Number.

Download PdfFileWriter_dll.zip - 300.4 KB

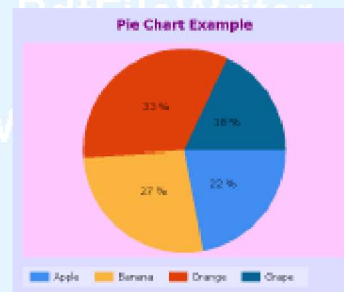
Download demo - 5.7 MB

Download source - 4.9 MB

Download PdfFileWriter_examples.zip - 5 MB

PDF FILE WRITER

Example



This area is an example of displaying text that is too long to fit within a fixed width area. The text is displayed justified to right edge. You define a text box with the required width and first line indent. You add text to this box. The box will divide the text into lines. Each line is made of segments of text. For each segment, you define font, font size, drawing style and color. After loading all the text, the program will draw the formatted text.

Example of multiple fonts: Times New Roman, **Comic Sans MS**, Example of regular, **bold**, *italic*, ***bold plus italic***. Arial size 7, size 8, size 9, size 10. Underline, ~~Strikeout.~~ Subscript H_2O. Superscript $A^2+B^2=C^2$

Some color, green, blue, orange, and purple.

Support for non-Latin letters: עברית АБВГДЕαβγδεζ

In the examples above this area the text box was set for first line indent of 0.25 inches. This paragraph has zero first line indent and no right justify.

This paragraph is set to first line hanging indent of 0.5 inches. The left margin of this paragraph is 0.5 inches.

Example of PdfTable support

Description	Price	Qty	Total
Current Trends in Theoritical Computer Science: Algorithms and Complexity. By: Paun	123.50	2	247.00
Theoretical Computer Science: Introduction to Automata, Computability, Complexity, Algorithmics, Randomization. By: Juraj Hromkovic	76.81	1	76.81
Discrete Mathematics for Computer Science (with Student Solutions Manual CD-ROM). By: Gary Haggard, John Schlipf and Sue Whitesides	229.88	1	229.88
		Total before tax	553.69
		Tax (13%)	71.98
		Total payable	625.67

PdfFileWriter • PDF File Writer C# Class Library • Author: Uzi Granot   Click to view article

1. Introduction

The PDF File Writer C# class library `PdfFileWriter` allows you to create PDF files directly from your .net application. The library shields you from the details of the PDF file structure. To use the library, you need to add a reference to the attached `PdfFileWriter.dll` class library file, add a `using PdfFileWriter` statement in every source file that uses the library and include the `PdfFileWriter.dll` with your distribution. For more details go to 4. Installation. The code was developed using .NET Framework 4.6.2 and Visual Studio 2019.

Version 1.26.0 enhancements: Support for PDF XMP Metadata and support for QR Code ECI Assinment number.

The PDF File Writer C# class library supports the following PDF document's features:

- Graphics: drawing lines, rectangles, polygons, Bezier curves, foreground and background color, patterns and shading. Section 2.1 Coordinate System.
- Decimal separator note for users in world regions using comma to denote fraction. See section 2.2 Decimal Separator.
- Drawing text. Section 2.3 Language Support, fonts and character sets.
- Drawing images: drawing raster (Bitmap) images and vector (Metafile) images. Section 2.4. Image Support.
- Barcode: support for Barcode 128, Barcode 39, Barcode interleaved 2 of 5, Barcode EAN13 and Barcode UPC-A. Section 2.5 Barcode Support.
- QR Code: support for two dimensions barcode. Section 2.6 QR Code Support.
- PDF417 barcode. Section 2.7 PDF417 Barcode.
- Web Link: Web link interactive support. Section 2.8 Web Link Support.
- Bookmarks: Support for document outlines. Section 2.9 Bookmark Support.
- Charts: Support for Microsoft Charting. Section 2.10 Charting Support.
- Print to PDF: Create a PDF document from `PrintDocument` process. Section 2.11 PrintDocument Support.
- Display data tables. Section 2.12 Data Table Support
- Play video files. Section 2.13 Play Video Files
- Play sound files. Section 2.14 Play Sound Files
- Attach data files. Section 2.15 Attach Data Files
- Reorder pages. Section 2.16 Reorder Pages
- PDF document output to a file or to a stream. Section 2.17 PDF Document Output.
- PDF document information dictionary. The PDF reader displays this information in the Description tab of the document properties. The information includes: Title, Author, Subject, Keywords, Created date and time, Modified date and time, the Application that produced the file, the PDF Producer. Section 2.18 Document Information Dictionary.
- Memory control: Write contents information of completed pages to output file and free unused memory with garbage collector. Section 2.19. Memory Control.
- Draw artwork defined by `System.Windows.Media.PathGeometry` class. Input argument can be text string or `PathGeometry` class. Section 2.20 Windows Presentation Foundation WPF
- Transparency or opaqueness is available for painting shapes, lines, text and images. Your application can set the alpha component of color for all graphics and text. Section 2.21 Transparency, Opacity, Alpha Color Component and Blending
- Blend. The library supports the PDF color blending scheme. Blending defines how the color of a new item painted over a previous item is handled. Section 2.21 Transparency, Opacity, Alpha Color Component and Blending
- Document Links and Named Destination. Section 2.22 Document Links and Named Destination.
- Encryption: support for AES-128 encryption. Section 2.23 Encryption Support.
- Sticky Notes 2.24 Sticky Notes or Text Annotation.
- Layers or Optional Content. 2.25 Layers or Optional Content.
- Initial document display. 2.26 Initial Document Display.
- XMP Metadata. 2.27 XMP Metadata.

Creating a PDF is a six steps process.

- Step 1: Create one document object `PdfDocument`.
- Step 2: Create resource objects such as fonts or images (i.e. `PdfFont` or `PdfImage`).
- Step 3: Create page object `PdfPage`.
- Step 4: Create contents object `PdfContents`.
- Step 5: Add text and graphics to the contents object (using `PdfContents` methods).
- Repeat steps 3, 4 and 5 for additional pages
- Step 6: Create your PDF document file by calling `CreateFile` method of `PdfDocument`.

Step 5 is where most of your programming effort will be spent. Adding contents is achieved by calling the methods of `PdfContents` class to render graphics and text. The contents class has a rich set (about 100) of methods for adding text and graphics to your document.

`PdfDocument` implements the `IDisposable` interface to release unmanaged resources. The `CreateFile` method calls `Document.Dispose()` after the PDF file is created. However, to ensure the release of resources you should wrap the `PdfDocument` creation and the final `CreateFile` with either a `using` statement or a `try/catch` block./p>

The demo program attached to this article is the test program developed to debug the library. The `TestPdfFileWriter` has six buttons on the main screen. Five buttons to produce examples of PDF files and one button to display all fonts available on your computer. The first button "Article Example" creates the PDF file displayed at the top of this article. Section 3. Development Guide by Example.

As stated before, the `PdfFileWriter` C# class library shields you from the complexities of the PDF file structure. However, good understanding of PDF file is always an advantage. Adobe PDF file specification document available from Adobe website: "PDF Reference, Sixth Edition, Adobe Portable Document Format Version 1.7 November 2006". It is an intimidating 1310 pages document. I would strongly recommend reading Chapter 4 Graphics and sections 5.2 and 5.3 of the Text chapter 5.

If you want to analyze the PDF files created by this project, or if you want to understand PDF file structure in general, you can use the demo program attached to my previous article "PDF File Analyzer With C# Parsing Classes". This article provides a concise overview of the PDF specifications.

2. PDF File Writer Library General Notes

2.1. Coordinate system and Unit of Measure

The PDF coordinate system origin is at the bottom left corner of the page. The X-axis is pointing to the right. The Y-axis is pointing in upward direction.

The PDF unit of measure is point. There are 72 points in one inch. The PDF File writer allows you to select your own unit of measure. All methods arguments representing position, width or height must be in your unit of measure. There are two exceptions: font size and resolution. Font size is always in points. Resolution is always in pixels per inch. The PDF File Writer converts all input arguments to points. All internal measurement values and calculations are done with double precision. At the final step when the PDF file is created, the values are converted to text strings. The conversion precision is six digits. The conversion formula used is:

```
// Value is Double
if(Math.Abs(Value) < 0.0001) Value = 0.0;
String Result = ((Single) Value).ToString();
```

2.2. Decimal Separator

PDF readers such as Adobe Acrobat expect real numbers with a fraction to use period as the decimal separator. Some of the world regions use other decimal separators such as comma. Since Version 1.1 of the PDF File Writer library will use period as decimal separator regardless of regional setting of your computer.

2.3. Language support, fonts and character sets

The PDF File Writer library supports most of the fonts installed on your computer. The only exception is device fonts. The supported fonts follow the OpenType font specifications. More information is available at Microsoft Typography - OpenType Specification. The text to be drawn is stored in a String made of Unicode characters. The library will accept any character (0 to 65536) except control codes 0 to 31 and 128 to 159. Every character is translated into a glyph. The glyphs are drawn on the page left to right in the same order as they are stored in the string. Most font files support only a subset of all possible Unicode characters. In other words, you must select a font that supports the language of your project or the symbols you are trying to display. If the input String contains unsupported glyphs, the PDF reader will display the "undefined glyph". Normally it is a small rectangle. The test program attached to this article has a "Font Families" button. If you click it you can see all available fonts on your computer and within each font all available characters. If the language of your project is a left to right language and each character is translated into one glyph and the glyph is defined in the font, the result should be what you expect. If the result is not what you expect, here are some additional notes:

Unicode control characters. Unicode control characters are used to control the interpretation or display of text, but these characters themselves have no visual or spatial representation. The PDF File writer does not identify these characters. The library assumes that every

character is a display character. They will be displayed as undefined character.

Right to left language. Normally the order of characters in a text string is the order a person would read them. Since the library draws left to right the text will be written backwards. The `ReverseString` method reverses the character order. This will solve the problem if the text is made only of right to left characters. If the text is a mix of right to left, left to right, numbers and some characters such as brackets ()[]<>{} it will not produce the desired results. Another limitation is `TextBox` class cannot break long right to left text into lines.

Ligature. In some languages a sequence of two or more characters are grouped together to display one glyph. Your software can identify these sequences and replaced them with the proper glyph.

Dotted circle. If you look at the Glyph column of Glyph Metrics screen you can see that some glyphs have a small dotted circle (i.e. character codes 2364 and 2367). These characters are part of a sequence of characters. The dotted circle is not displayed. If the advance width is zero and the bounding box is on the left side of the Y axis, this glyph will be drawen ok. It will be displayed on top of the previous character. If The advance width is not zero, this glyph should be displayed before the previous character. Your software can achieve it by reversing the two characters.

2.4. Image Support

Displaying images in the PDF document is handled by the PdfImage class. This class is a PDF resource. Image sources can be:

- Image file
- .NET Image derived class such as Bitmap
- A Boolean array of black and white pixels
- QRCode barcode represented by QREncoder class
- PDF 417 barcode represented by Pdf417Encoder class
- PdfChart is a derived class of PdfImage
- PdfPrintDocument is using internally PdfImage to capture print pages

Images are saved to the PDF file in one of the following formats:

- Jpeg format (lossy compression)
- Indexed bitmap (Lossless compression)
- Gray bitmap (Lossless compression)
- Black and white bitmap (Lossless compression)

Color pictures should be saved in Jpeg format. To control the size of your image, you can reduce the resolution or change the image quality. Color pictures can be saved in shades of gray. Data size is cut by three, but you lose the color. If the image was created programmatically as in charts, and the number of colors is less than 256 the image can be saved as an indexed bitmap. Each color is represented by one byte (or less) compare to 3 bytes. This can result in a very significant file size reduction. For example, the ChartExample.pdf file is reduced from 642KB to 72KB. If the image is black and white as in PdfPrintDocument images of text, the image can be saved as BWImage. In the case of PrintExample.pdf the Jpeg file is 1795KB and the black and white version is 66KB.

Adding an image to your PDF File

Create a PdfImage class.

```
// create PdfImage object
PdfImage MyImage = new PdfImage(Document);
```

Set optional parameters if required. All the parameters have a default value.

```
// saved image format (default SaveImageAs.Jpeg)
// other choices are: IndexedImage, GrayImage, BWImage
MyImage.SaveAs = SaveImageAs.Jpeg;

// Crop rectangle is the image area to be cropped.
// The default is no crop (empty rectangle).
// The origin is at top left and Y axis is pointing down.
// The dimensions are in pixels.
// The crop rectangle must be contained within the image.
MyImage.CropRec = new Rectangle(x, y, width, height);
```

```
// Crop percent rectangle is the image area to be cropped.
// The default is no crop (empty rectangle).
// The origin is at top left and Y axis is pointing down.
// The dimensions are in percent of image.
// The crop rectangle must be contained within the image.
MyImage.CropPercent = new RectangleF(x, y, width, height);

// Image Quality is an integer in the range of 0 to 100.
// representing poor to best quality image.
// The default (-1) is save image with Bitmap default quality.
// For your information Bitmap class default image quality is 75.
// Lower image quality means smaller PDF file.
MyImage.ImageQuality = PdfImage.DefaultQuality;

// Image resolution sets the image resolution provided
// that it is smaller than the resolution of source image.
// Default of 0, is keeping the original image resolution.
// Resolution is specified in pixels per inch.
// Reduced resolution means smaller PDF file.
MyImage.Resolution = 0;

// Cutoff value for gray conversion to black and white.
// The range is 1 to 99. The default is 50.
// If shade of gray is below cutoff, the result is black.
// If shade of gray is above cutoff, the result is white.
MyImage.GrayToBWCutoff = 50;

// Reverse black and white
// Default is false
// In Gray or BW images the color is reversed
MyImage.ReverseBW = false;

// Attach to layer control
// Image visibility will be controlled by viewer application.
MyImage.LayerControl = PdfLayer;
```

Load image into PdfImage object. NOTE: the loading method saves the image in the PDF output file. Once this step is done the image cannot be changed.

```
// use one of 5 methods to load the image.
// image source can be a file, Bitmap,
// BW bool array, QRCode or Pdf417 barcode
MyImage.LoadImage(image_source);
```

Draw the image into the PDF Document.

```
// draw the image
Contents.DrawImage(MyImage, PosX, PosY, Width, Height);
```

If you want the image to maintain correct aspect ratio use `ImageSize` or `ImageSizePosition` to calculate the width and height. If the ratio of width and height is not the same as the image, the image will look stretched in one of the directions.

```
// calculate the largest rectangle with the correct
// aspect ratio
SizeD MyImage.ImageSize(Double Width, Double Height);
```

```
// calculate the largest rectangle with
// correct aspect ratio that will fit in a given area and
// position. It based on <code>ContentAlignment</code> enumeration.
ImageSizePos ImageSizePosition(Double Width, Double Height, ContentAlignment Alignment);
```

2.5. Barcode Support

The code below illustrates how to include UPC-A barcode in a PDF document.

```
// create barcode object
BarcodeEAN13 Barcode = new BarcodeEAN13("123456789010");

// draw the barcode including text under the barcode
Contents.DrawBarcode(PosX, PosY, BarWidth, BarcodeHeight, Barcode, Font, FontSize);
```

In this case the class is BarcodeEAN13 with 12 digits input string. The result is UPC-A barcode.

The PDF File Writer library includes a base class `Barcode`. For each supported barcode one needs a derived class. The class library includes four derived classes: `Barcode128`, `Barcode39`, `BarcodeInterleaved2of5` and `BarcodeEAN13`. The `BarcodeEAN13` produces EAN-13 barcode if the input string is 13 digits and UPC-A if the input string is 12 digits. Input string with 13 digit and a leading zero is considered UPC-A.

The `DrawBarcode` method has a number of overloads. You specify the position of the bottom left corner of the barcode, the width of the narrow bar, the height of the barcode and the derived barcode class. There are optional arguments: justification (left, center, right) color and font to display the text. Quiet zone around the barcode is your responsibility. Optional text is displayed below the barcode. If you select color other than black you should make sure the contrast to the background is significant. Usage examples are given in 3.7 Draw Barcodes, `ArticleExample.cs` and `OtherExample.cs`.

If you want to create a derived class for another barcode, use the source code for the three included classes as an example.

2.6. QR Code Support

The PDF File Writer library provides support for QR Code. It is based on article QR Code Encoder and Decoder .NET(Framework, Standard, Core) Class Library Written in C#. The program supports three characters sets: numeric, alphanumeric and eight-bit byte. The program does not support Kanji characters. The program will scan the data string input and selects the most effective character set. If your data can be divided into segments with just digits or just alphanumeric characters you can create a QR Code object with array of data strings.

Adding QRCode barcode to your PDF document must follow the steps below.

- Create `QREncoder` object.
- Set encoding options. All encoding options have default values.
- Encode a data string or a data bytes array.
- Create `PdfImage`.
- Draw the barcode image with `PdfContent.DrawImage`

QR Code example

```
// create QRCode barcode
QREncoder QREncoder = new QREncoder();

// set error correction code (default is M)
QREncoder.ErrorCorrection = ErrorCorrection.M;

// set module size in pixels (default is 2)
QREncoder.ModuleSize = 1;

// set quiet zone in pixels (default is 8)
QREncoder.QuietZone = 4;

// ECI Assignment Value (default is -1 not used)
// The ECI value is a number in the range of 0 to 999999.
// or -1 if it is not used
Encoder.ECIAssignValue = -1;
```

```
// encode your text or byte array
QREncoder.Encode(ArticleLink);

// convert QRCode to PdfImage in black and white
PdfImage BarcodeImage = new PdfImage(Document, QREncoder);

// draw image (height is the same as width for QRCode)
Contents.DrawImage(BarcodeImage, 6.0, 6.8, 1.2);
```

For coding examples please review 3.7 Draw Barcodes, ArticleExample.cs and OtherExample.cs source code.

2.7. PDF417 Barcode

PDF417 barcode support software is based on article PDF417 Barcode Encoder Class Library and Demo App. The PDF417 barcode documentation and specification can be found in the following websites. Wikipedia provides a good introduction to PDF417. Click here to access the page. The PDF417 standard can be purchased from the ISO organization at this website. An early version of the specifications can be downloaded from this website for free. I strongly recommend that you download this document if you want to fully understand the encoding options.

The PDF417 barcode encodes array of bytes into an image of black and white bars. Encoding Unicode text requires converting Unicode characters into bytes. The decoder must do the reverse process to recover the text. The bytes are converted to codewords. This conversion process compresses the bytes into codewords. The encoder adds error correction codewords for error detection and recovery. Once the total number of data codewords and error correction codewords is known , the encoder divides the codewords into data rows and data columns. The final step is the creation of a black and white image.

Adding PDF417 barcode to your PDF document must follow the steps below.

- Create `Pdf417Encoder` object.
- Set encoding options. All encoding options have default values.
- Encode a data string or a data bytes array.
- Check the image width and height or the number of data columns and data rows to make sure image size is appropriate for your application. If it is not, adjust the layout.
- Create `PdfImage`.
- Draw the barcode image with `PdfContent.DrawImage`

Example of PDF417 Barcode Drawing

```
private void DrawPdf417Barcode()
{
// save graphics state
Contents.SaveGraphicsState();

// create PDF417 barcode
Pdf417Encoder Pdf417 = new Pdf417Encoder();

string ArticleLink = "http://www.codeproject.com/Articles/570682/PDF-File-Writer-Csharp-Class-
Library-Version";

// encode text
Pdf417.Encode(ArticleLink);
Pdf417.WidthToHeightRatio(2.5);

// convert Pdf417 to black and white image
PdfImage BarcodeImage = new PdfImage(Document, Pdf417);

// draw image
Contents.DrawImage(BarcodeImage, 1.1, 5.2, 2.5);

// restore graphics sate
Contents.RestoreGraphicsState();
```

```
return;
}
```

PDF417 Barcode Object

Create PDF417 barcode object. This object can be reused serially to produce multiple barcodes.

```
// create PDF417 barcode
Pdf417Encoder Pdf417 = new Pdf417Encoder();
```

Set optional parameters to control the encoding process.

Encoding control

The PDF417 encoder encodes Input bytes into codewords. There are three types of codewords: byte, text and numeric. The program has an algorithm to divide the data input into these three types to compress the data. The default is Auto. However, you can restrict the encoding to only bytes or only text and bytes.

```
Pdf417.EncodingControl = Pdf417EncodingControl.Auto;
// or
Pdf417.EncodingControl = Pdf417EncodingControl.ByteOnly;
// or
Pdf417.EncodingControl = Pdf417EncodingControl.TextAndByte;
```

Error Correction Level

The PDF417 adds error correction codewords to detect errors and correct them. More error correction codewords improves the reliability of the barcode. However, it makes the barcode bigger. Error correction level allows you to control the quality of the barcode. The `ErrorCorrectionLevel` enumeration has two types of values. Fixed levels from 0 to 8. And levels that are recommended values based on the number of data codewords. The default value in `ErrorCorrectionLevel.AutoNormal`. For more details look at Table 6 and Table 7 in the PDF417 Specification.

```
Pdf417.ErrorCorrection = ErrorCorrectionLevel.Level_0; // up to Level_8
// or
Pdf417.ErrorCorrection = ErrorCorrectionLevel.AutoNormal;
// or
Pdf417.ErrorCorrection = ErrorCorrectionLevel.AutoLow; // one less than normal
// or
Pdf417.ErrorCorrection = ErrorCorrectionLevel.AutoMedium; // one more than normal
// or
Pdf417.ErrorCorrection = ErrorCorrectionLevel.AutoHigh; // two more than normal
```

Narrow Bar Width

The width in pixels of a narrow barcode bar. The default is 2. If this value is changed, the program makes sure that `RowHeight` is at least three times that value. And that `QuiteZone` is at least twice that value.

```
Pdf417.NarrowBarWidth = value;
```

Row Height

The height in pixels of one row. This value must be greater than or equal to 3 times the `NarrowBarWidth` value. The default is 6.

```
Pdf417.RowHeight = value;
```

Quiet Zone

The width of the quiet zone all around the barcode. The quiet zone is white. This value must be greater than or equal to 2 times the `NarrowBarWidth` value. The default is 4.

```
Pdf417.QuietZone = value;
```

Default Data Columns

The default data columns value. The value must be in the range of 1 to 30. The default is 3. After the input data is encoded, the software sets the number of data columns to the default data columns and calculates the number of data rows. If the number of data rows exceeds the maximum allowed (90), the software sets the number of rows to the maximum allowed and recalculate the number of data columns. If the result is greater than the maximum columns allowed, an exception is thrown.

```
Pdf417.DefaultDataColumns = value;
```

Global Label ID Character Set

Set Global Label ID Character Set to ISO 8859 standard. The n can be 1 to 9 or 13 or 15. If the string is null, the default of ISO-8859-1 is used. Language support is defined here.

```
Pdf417.GlobalLabelIDCharacterSet = "ISO-8859-n";
```

Global Label ID User Defined

Set Global Label ID User Defined value. The default is not used. I did not find any reference explaining the usage of this value. User defined value must be between 810900 and 811799.

```
Pdf417.GlobalLabelIDUserDefined = UserDefinedValue;
```

Global Label ID General Purpose

Set Global Label ID General Purpose value. The default is not used. I did not find any reference explaining the usage of this value. User defined value must be between 900 and 810899.

```
Pdf417.GlobalLabelIDGeneralPurpose = UserDefinedValue;
```

Encoding data

There are two encoding methods. One accepts text string as an input and the other one accepts byte array as an input.

```
Pdf417.Encode(string StringData);
// or
Pdf417.Encode(byte[] BinaryData);
```

The barcode was designed for binary data. Therefore, the first method above must encode the string from 16 bit characters to byte array. The encode method with string input has the following conversion logic. The Global Label ID Character Set property control the

conversion. It is done in two steps. Step one string to UTF8 and step two UTF8 to ISO-8859-n. If you want to encode Hebrew you should set the character set to ISO-8859-8.

```
public void Encode(string StringData)
{
// convert string to UTF8 byte array
byte[] UtfBytes = Encoding.UTF8.GetBytes(StringData);

// convert UTF8 byte array to ISO-8859-n byte array
Encoding ISO = Encoding.GetEncoding(_GlobalLabelIDCharacterSet ?? "ISO-8859-1");
byte[] IsoBytes = Encoding.Convert(Encoding.UTF8, ISO, UtfBytes);

// call the encode binary data method
Encode(IsoBytes);
Return;
}
```

Final Barcode Layout

After the data was encoded, you can check the layout of the barcode by examining these values: `ImageWidth`, `ImageHeight`, `DataColumns` or `DataRows`. If you want to readjust the width and the height of the barcode you can used one of the methods below. In addition, you can readjust the optional parameters: `NarrowBarWidth`, `RowHeight` or `QuietZone`values.

Width to Height Ratio

This method will calculate the number of data rows and data columns to achieve a desired width to height ratio. The ratio includes the quiet zone. Check the return value for success.

```
Bool Pdf417.WidthToHeightRatio(double Ratio);
```

Set Data Columns

This method will calculate the number of data rows based on the desired number of data columns. Check the return value for success.

```
Bool Pdf417.SetDataColumns(int Columns);
```

Set Data Rows

This method will calculate the number of data columns based on the desired number of data rows. Check the return value for success.

```
Bool Pdf417.SetDataRows(int Rows);
```

Create PDF Document Image Resource

In this step we create a PDF document image resource from the PDF417 barcode.

```
PdfImage BarcodeImage = new PdfImage(Document, Pdf417);
```

Draw the Barcode

The last step is adding the barcode to the content of a PDF document’s page.

```
// PosX and PosY are the page coordinates in user units.
// Width is the width of the barcode in user units.
// The height of the barcode is calculated to preserve aspect ratio.
// Height = Width * BarcodeImage.ImageHeight / BarcodeImage.ImageWidth
Contents.DrawImage(BarcodeImage, PosX, PosY, Width);
```

2.8. Web Link Support

The PDF File Writer library provides support to web linking. This feature is one of the PDF interactive features described in the PDF reference manual in Section 8 Interactive Features. It is a combination of annotation and action. Annotation associates a web link with an area on the page. When the user clicks on the area, the PDF reader will activate the default web browser navigating to the desired web page.

The annotation area is a rectangle area defined by absolute coordinates relative to the bottom left corner of the page. To add a web link call `AddWebLink` method of the `PdfPage` class.

```
Page.AddWebLink(Double LeftPos, Double BottomPos, Double RightPos, Double TopPos, String WebLink);
```

Annotations are not part of the page contents. In order or the reader of your PDF document to know where to click you need to display appropriate text or graphics in the same area on the page. In other words you need to call two methods. The `AddWebLink` method associated with the page and a second method associated with the contents. The second method can be a graphic object such as image or a rectangle, or text. Because `AddWebLink` requires coordinates relative to the bottom left corner of the page, the coordinates of your graphic object must be the same. In other words, do not use translation, scaling or rotation. If you do, you need to make sure that the two areas will coincide.

The PDF File Writer has several `PdfContents` methods supporting text annotation.

Draw a line of text with associated web link. The text will be left justified, underlined and blue. Text position is relative to bottom left corner of the page.

```
// font size in points
PdfContents.DrawWebLink(PdfPage Page, PdfFont Font, Double FontSize,
    Double TextAbsPosX, Double TextAbsPosY, String Text, String WebLink);
```

Draw a line of text with associated web link. Text position is relative to bottom left corner of the page.

```
// font size in points
PdfContents.DrawWebLink(PdfPage Page, PdfFont Font, Double FontSize, Double TextAbsPosX, Double
TextAbsPosY,
    TextJustify Justify, DrawStyle DrawStyle, Color TextColor, String Text, String WebLink);
```

Drawing web link within `TextBox` is a two step process. First you add the text and the web link string to the box using one of the `AddText` methods of `TextBox` class. Second you draw the `TextBox` to the page contents using one of the `DrawText` methods of `PdfContents`.

Add web link to `TextBox`. The text will be displayed underlined and blue.

```
TextBox.AddText(PdfFont Font, Double FontSize, String Text, String WebLink);
```

Add web link to `TextBox`. The text attributes are defined by `DrawStyle` and `FontColor`.

```
TextBox.AddText(PdfFont Font, Double FontSize, DrawStyle DrawStyle, Color FontColor, String Text,
String WebLink);
```

Second step draw text to contents. This method assumes no extra line or paragraph spacing. Note, if you call `DrawText` without `PdfPage` argument on a `TextBox` with `WebLink` information, `ApplicationException` will be thrown.

```
// note: PosYTop is by reference.
// On exit from the method the PosYTop will have the next Y position
PdfContents.DrawText(Double PosX, ref Double PosYTop, Double PosYBottom, Int32 LineNo, TextBox Box,
PdfPage Page);
```

This method lets you define extra line or paragraph spacing. Note, if you call `DrawText` without `PdfPage` argument on a `TextBox` with `WebLink` information, `ApplicationException` will be thrown.

```
// note: PosYTop is by reference.
// On exit from the method the PosYTop will have the next Y position
PdfContents.DrawText(Double PosX, ref Double PosYTop, Double PosYBottom, Int32 LineNo,
    Double LineExtraSpace, Double ParagraphExtraSpace, Boolean FitTextToWidth, TextBox Box, PdfPage
Page);
```

For coding examples please review 3.4 Draw Frame, ArticleExample.cs and OtherExample.cs source code.

2.9. Bookmarks Support

Bookmarks are described in the PDF specification (section 8.2.2 Document Outline) as follows: "A PDF Document may optionally display a document outline on the screen, allowing the user to navigate interactively from one part of the document to another. The outline consists of a tree-structured hierarchy of outline items (sometimes called bookmarks), which serve as a visual table of contents to display the document's structure to the user. The user can interactively open and close individual item by clicking them with the mouse."

The OtherExample.cs source code has an example of bookmarks. At one location there is a hierarchy of three levels. You can see the result in OtherExample.pdf file.

The first step in adding bookmarks to your application is:

```
// set the program to display bookmarks
// and get the bookmark root object
PdfBookmark BookmarkRoot = Document.GetBookmarksRoot();
```

This step activates bookmarks in your document and returns the root node.

Adding bookmarks is similar to adding controls to a windows form. The first level bookmarks are added to the root. Subsequent levels are added to existing bookmarks. At minimum you have to define a title, page, vertical position on the page and an open entries flag. Page is the PdfPage object of the page to go to. YPos is the vertical position relative to the bottom left corner of the page. Open entries flag is true if the lower level bookmarks are visible and false if the lower level are hidden. The first level is always visible by default.

```
// hierarchy example
PdfBookmark FirstLevel_1 = BookmarkRoot.AddBookmark("Chapter 1", Page, YPos, false);
PdfBookmark SecondLevel_11 = FirstLevel_1.AddBookmark("Section 1.1", Page, YPos, false);
PdfBookmark SecondLevel_12 = FirstLevel_1.AddBookmark("Section 1.2", Page, YPos, false);
PdfBookmark ThirdLevel_121 = SecondLevel_12.AddBookmark("Section 1.2.1", Page, YPos, false);
PdfBookmark ThirdLevel_122 = SecondLevel_12.AddBookmark("Section 1.2.2", Page, YPos, false);
PdfBookmark SecondLevel_13 = FirstLevel_1.AddBookmark("Section 1.3", Page, YPos, false);
PdfBookmark FirstLevel_2 = BookmarkRoot.AddBookmark("Chapter 2", Page, YPos, false);
PdfBookmark SecondLevel_21 = FirstLevel_2.AddBookmark("Section 2.1", Page, YPos, false);
PdfBookmark SecondLevel_22 = FirstLevel_2.AddBookmark("Section 2.2", Page, YPos, false);
```

AddBookmark() method has four overloading variations:

```
// basic
public PdfBookmark AddBookmark
    (
    String  Title,      // bookmark title
    PdfPage Page,       // bookmark page
    Double  YPos,       // bookmark vertical position relative to bottom left corner of the page
    Boolean OpenEntries // true is display children. false hide children
    )
```

```
// title color and style
public PdfBookmark AddBookmark
    (
    String      Title,      // bookmark title
    PdfPage     Page,       // bookmark page
    Double      YPos,       // bookmark vertical position relative to bottom left corner of the
page
    Color       Paint,      // bookmark color. Coloe.Empty is display title in default color
    TextStyle       TextStyle,  // bookmark text style: normal, bold, italic, bold-italic
    Boolean     OpenEntries // true is display children. false hide children
    )

// XPos and zoom
public PdfBookmark AddBookmark
    (
    String      Title,      // bookmark title
    PdfPage     Page,       // bookmark page
    Double      XPos,       // bookmark horizontal position relative to bottom left corner of the
page
    Double      YPos,       // bookmark vertical position relative to bottom left corner of the
page
    Double      Zoom,       // Zoom factor. 1.0 is 100%. 0.0 is ignore zoom.
    Boolean     OpenEntries // true is display children. false hide children
    )

// all options
public PdfBookmark AddBookmark
    (
    String      Title,      // bookmark title
    PdfPage     Page,       // bookmark page
    Double      XPos,       // bookmark horizontal position relative to bottom left corner of the
page
    Double      YPos,       // bookmark vertical position relative to bottom left corner of the
page
    Double      Zoom,       // Zoom factor. 1.0 is 100%. 0.0 is ignore zoom.
    Color       Paint,      // bookmark color. Coloe.Empty is display title in default color
    TextStyle       TextStyle,  // bookmark text style: normal, bold, italic, bold-italic
    Boolean     OpenEntries // true is display children. false hide children
    )
```

`PdfBookmark` class exposes one more method `GetChild`. You can get any bookmark by calling `GetChild` with one or more integer arguments. Each argument is a zero base argument of the child position in the level. For example `GetChild(2)` is the third item of the first level. `GetChild(2, 3)` is the forth second level item of the third first level item.

2.10. Charting Support

The PDF specification has no specific support for charting. The PDF File Writer library provides charting support by allowing the developer to create a Microsoft Charting object and draw this object as an image into the PDF file. For more information about Microsoft Chart control note MSDN library documentation Visual Studio 2012 Chart Controls. The documentation for the charting name space is available at Data Visualization Charting Namespace. The attached `ChartExample.cs` has four examples of charts. If you intend to use charting, you need to add `System.Windows.Forms.Visualization` reference to your project. In each source module using `Chart` you need to add `using System.Windows.Forms.DataVisualization.Charting;`.

Adding a chart to a PDF document is four steps process.

- Create Chart object.
- Create PdfChart object.
- Build the chart.
- Draw the PdfChart to PdfContents.

The recommended way to create a chart is to use a static method of `PdfChart` object.

```
// Document is your PdfDocument object.
// Width and height are in user units.
// Resolution is in pixels per inch.
// Resolution is optional. If not included the library will take the .net default.
// Library will create Chart object with width and height in pixels and set resolution in pixels
per inch
Chart MyChart = PdfChart.CreateChart(PdfDocument Document, Double Width, Double Height, Double
Resolution);
```

You can instantiate `Chart` class yourself.

```
Chart MyChart = new Chart();
MyChart.RenderingDpiY = 300;    // example of 300 pixels per inch
MyChart.Width = 1950;       // example of 6.5 inches in pixels
Mychart.Height = 1350;      // example of 4.6 inches in pixels
```

Next you create a `PdfChart` from the `Chart` created above. Optionally, you can override the resolution.

```
// resolution is optional. It will override the resolution set above.
PdfChart MyPdfChart = new PdfChart(PdfDocument Document, Chart MyChart, Double Resolution);
```

Next you build your chart. `ChartExample.cs` has four examples. The documentation for building a chart is beyond the scope of this article. There plenty of examples on the Internet.

`PdfChart` has a `CreateFont` method to simplify the creation of fonts. It will calculate font size based on chart's resolution.

```
// FontSizeUnit is an enumeration
// Available units: pixel, point, UserUnit, Inch, cm, mm
Font CreateFont(String FontFamilyName, FontStyle Style, Double FontSize, FontSizeUnit Unit);
```

The last step is drawing the chart.

```
// Draw the chart at OrigX, OrigY in user units
// The width and height of the chart are taken from the Chart object.
// They are calculated from the size in pixels and resolution of the chart.
public void PdfContents.DrawChart(PdfChart MyPdfChart, Double OrigX, Double OrigY);

// Draw the chart at OrigX, OrigY with Width and Height as specified, all in user units.
// NOTE: Width and Height should be selected to agree with the aspect ratio of the chart object.
public void PdfContents.DrawChart(PdfChart MyPdfChart, Double OrigX, Double OrigY, Double Width,
Double Height);
```

The `PdfChart` class provides some optional methods to control image positioning.

The `ImageSize` method returns the largest rectangle with correct aspect ratio that will fit in a given area.

```
SizeD ImageSize(Double Width, Double Height);
```

The `ImageSizePosition` method returns the largest rectangle with correct aspect ratio that will fit in a given area and position it based on `ContentAlignment` enumeration.

```
ImageSizePos ImageSizePosition(Double Width, Double Height, ContentAlignment Alignment);
```

2.11. Print Document Support

Print document support allows you to print a report in the same way as printing to a printer and producing a PDF document. The difference between this method of producing a PDF file and using `PdfContents` to produce a PDF file is the difference between

raster graphics to vector graphics. Print document support creates one jpeg image per page. `PrintExample.cs` has an example of creating a three page document.

Normally each page is a full image of the page. If your page is letter size and the resolution is 300 pixels per inch, each pixel is 3 bytes, the bit map of the page will be 25.245MB long. `PrintPdfDocument` has a method `CropRect` that can reduce the size of the bit map significantly. Assuming one inch margin is used, the active size of the bit map will be reduced to 15.795 MB. That is 37.4% reduction.

```
// main program
// Create empty document
Document = new PdfDocument(PaperType.Letter, false, UnitOfMeasure.Inch);

// create PrintPdfDocument producing an image with 300 pixels per inch
PdfImageControl ImageControl = new PdfImageControl();
ImageControl.Resolution = 300.0;
PrintPdfDocument Print = new PrintPdfDocument(Document, ImageControl);

// PrintPage in the delegate method PrintPageEventHandler
// This method will print one page at a time to PrintDocument
Print.PrintPage += PrintPage;

// set margins in user units (Left, top, right, bottom)
// note the margins order are per .net standard and not PDF standard
Print.SetMargins(1.0, 1.0, 1.0, 1.0);

// crop the page image result to reduce PDF file size
// the crop rectangle is per .net standard.
// The origin is top left.
Print.CropRect = new RectangleF(0.95F, 0.95F, 6.6F, 9.1F);

// initiate the printing process (calling the PrintPage method)
// after the document is printed, add each page as an image to PDF file.
Print.AddPagesToPdfDocument();

// dispose of the PrintDocument object
Print.Dispose();

// create the PDF file
Document.CreateFile(FileName);
```

Example of PrintPage method

```
// Print each page of the document to PrintDocument class
// You can use standard PrintDocument.PrintPage(...) method.
// NOTE: The graphics origin is top left and Y axis is pointing down.
// In other words, this is not PdfContents printing.
public void PrintPage(object sender, PrintPageEventArgs e)
{
// graphics object short cut
Graphics G = e.Graphics;

// Set everything to high quality
G.SmoothingMode = SmoothingMode.HighQuality;
G.InterpolationMode = InterpolationMode.HighQualityBicubic;
G.PixelOffsetMode = PixelOffsetMode.HighQuality;
G.CompositingQuality = CompositingQuality.HighQuality;

// print area within margins
Rectangle PrintArea = e.MarginBounds;

// draw rectangle around print area
G.DrawRectangle(Pens.DarkBlue, PrintArea);
```

```
// line height
Int32 LineHeight = DefaultFont.Height + 8;
Rectangle TextRect = new Rectangle(PrintArea.X + 4, PrintArea.Y + 4, PrintArea.Width - 8,
LineHeight);

// display page bounds
// DefaultFont is defined somewhere else
String text = String.Format("Page Bounds: Left {0}, Top {1}, Right {2}, Bottom {3}",
    e.PageBounds.Left, e.PageBounds.Top, e.PageBounds.Right, e.PageBounds.Bottom);
G.DrawString(text, DefaultFont, Brushes.Black, TextRect);
TextRect.Y += LineHeight;

// display print area
text = String.Format("Page Margins: Left {0}, Top {1}, Right {2}, Bottom {3}",
    PrintArea.Left, PrintArea.Top, PrintArea.Right, PrintArea.Bottom);
G.DrawString(text, DefaultFont, Brushes.Black, TextRect);
TextRect.Y += LineHeight;

// print some lines
for(Int32 LineNo = 1; ; LineNo++)
    {
    text = String.Format("Page {0}, Line {1}", PageNo, LineNo);
    G.DrawString(text, DefaultFont, Brushes.Black, TextRect);
    TextRect.Y += LineHeight;
    if(TextRect.Bottom > PrintArea.Bottom) break;
    }

// move on to next page
PageNo++;
e.HasMorePages = PageNo <= 3;
return;
}
```

2.12. Data Table Support

The data table classes allow you to display data tables in your PDF document. `PdfTable` is the main class controlling the display of one table. A table is made out of a header row and data rows. Each row is divided into cells. `PdfTableCell` controls the display of one header cell or one data cell. If header is used it will be displayed at the top of the table. Optionally it will be displayed at the top of each additional page. To display data in a cell, you load the data into the `Value` property of `PdfTableCell`. Data can be text string, basic numeric value, Boolean, Char, `TextBox`, image, QR code or barcode. Independently of data, you can load the cell with document link, web link, video, audio or embedded file. Clicking anywhere within the cell's area will cause the PDF reader to activate the document link, web link, video, audio or embedded file. The display of the data is controlled by `PdfTableStyle` class. `PdfTable` class contains a default cell style and a default header style. You can override the default styles with private styles within `PdfTableCell`. To display a table, you create a `PdfTable` object. Next you initialize the table, header cells, data cells and styles objects. Finally, you set a loop and load the cell values of one row and then draw this row. This loop continues until all data was displayed. Below you will find the necessary sequence of steps to produce a table.

When `DrawRow` method is called, the software calculates the required row height. Row height is the height of the tallest cell. The row will be drawn if there is sufficient space within the table. When the available space at the bottom is too small, a new page is called, and optional heading and the current row are displayed at the top of the table. If the required row height is so large that it will not fit in full empty table, an exception is raised. In order to accommodate long multi-line Strings or `TextBoxes`, the software can handle these cases in a flexible way. Multi-line String is converted by `PdfTable` into a `TextBox`. The `PdfTableStyle` class has a `TextBoxPageBreakLines` property. If this property is set to zero (default), the `TextBox` is treated as other data values. `TextBox` height must fit the page. If `TextBoxPageBreakLines` is set to a positive integer, the system will calculate cell's height as `TextBox` height or the height the first few lines as specified by `TextBoxPageBreakLines`. The system will draw the row with as many lines that fit the page. A new page will be created, and the rest of the lines will be drawn. In other words, the first block of lines of a long `TextBox` will be at least `TextBoxPageBreakLines` long. TableExample.cs source contains an example of long `TextBox` cells.

Create a `PdfTable` object.

```
// create table
PdfTable Table = new PdfTable(Page, Contents, Font, FontSize);
```

Page is the current PdfPage. Contents is the current PdfContents. Font is the table default font. FontSize is the default font size in points.

Define table's area on the page.

```
// table's area on the page
Table.TableArea = new PdfRectangle(Left, Bottom, Right, Top);

// first page starting vertical position
Table.RowTopPosition = StartingTopPosition;
```

The four arguments are the four sides of the table relative to bottom left corner and in user units. If on the first page the table-top position is not at the top of the page set `RowTopPosition` to the starting top position. On subsequent pages the table will always start at the top. If `TableArea` is not specified, the library will set it to default page size less one inch margin.

Divide the table width into columns.

```
// divide table area width into columns
StockTable.SetColumnWidth(Width1, Width2, Width3, ...);
```

The number of arguments is the number of columns. The table width less total border lines will be divided in proportion to these arguments.

Once the number of columns is set with `SetColumnWidth` method the library creates two `PdfTableCell` arrays. One array for header cells and one array for data cells.

Rows and columns of the data table can be separated by border lines. Border lines properties are defined by `PdfTableBorder` and `PdfTableBorderStyle`. There are four horizontal border lines: `TopBorder`, `BottomBorder`, `HeaderHorBorder` between the header row and first data row and `CellHorBorder` between data rows. There are two sets of vertical border lines: `HeaderVertBorder` array for vertical border lines within the header row, and `CellVertBorder` array for vertical border lines between columns within the data part of the table. Arrays size is the number of columns plus one. Array element zero is the table's left border. Array element Columns is the table's right border. All other elements are lines separating columns. Each of these lines can be defined individually. There are methods to define all border lines at once or define each individual border line.

Methods to define all border lines:

```
// clear all border lines
Table.Borders.ClearAllBorders();

// set all border lines to default values (no need to call)
// All frame lines are one point (1/72") wide
// All grid lines are 0.2 of one point wide
// All borders are black
Table.Borders.SetDefaultBorders();

// set all borders to same width and black color
Table.Borders.SetAllBorders(Double Width);

// set all borders to same width and a specified color
Table.Borders.SetAllBorders(Double Width, Color LineColor);

// set all borders to one width and all grid lines to another width all lines are black
Table.Borders.SetAllBorders(Double FrameWidth, Double GridWidth);

// set all borders to one width and color and all grid lines to another width and color
Table.Borders.SetAllBorders(Double FrameWidth, Color FrameColor, Double GridWidth, Color
GridColor);

// set all frame borders to same width and black color and clear all grid lines
```

```
Table.Borders.SetFrame(Double Width);

// set all frame borders to same width and a specified color and clear all grid lines
Table.Borders.SetFrame(Double Width, Color LineColor);
```

Each horizontal border line can be cleared or set. The example is for top border line:

```
// clear border
Table.Borders.ClearTopBorder();

// set border with default color set to black
// Zero width means one pixel of the output device.
Table.Borders.SetTopBorder(Double LineWidth);

// set border
Table.Borders.SetTopBorder(Double LineWidth, Color LineColor);
```

Each vertical border line can be cleared or set. The example is for cell's vertical border lines:

```
// clear border
Table.Borders.ClearCellVertBorder(Int32 Index);

// set border with default color set to black
Table.Borders.SetCellVertBorder(Int32 Index, Double LineWidth);

// set border
Table.Borders.SetCellVertBorder(Int32 Index, Double LineWidth, Color LineColor);
```

Set other optional table properties. The values given in the example below are the defaults.

```
// header on each page
HeaderOnEachPage = true;

// minimum row height
MinRowHeight = 0.0;
```

Table information is processed one row at a time. Each row is made of cells. One cell per column. The display of cell's information is controlled by `PdfTableStyle` class. There are about 20 style properties. For the complete list view the source code or the help file. Some of these styles are specific to the type of information to be displayed. Here is an example

```
// make some changes to default header style
Table.DefaultHeaderStyle.Alignment = ContentAlignment.BottomRight;

// create private style for header first column
Table.Header[0].Style = Table.HeaderStyle;
Table.Header[0].Style.Alignment = ContentAlignment.MiddleLeft;

// load header value
Table.Header[0].Value = "Date";

// make some changes to default cell style
Table.DefaultCellStyle.Alignment = ContentAlignment.MiddleRight;
Table.DefaultCellStyle.Format = "#,##0.00";

// create private style for date column
Table.Cell[0].Style = StockTable.CellStyle;
Table.Cell[0].Style.Alignment = ContentAlignment.MiddleLeft;
Table.Cell[0].Style.Format = null;
```

After initialization is done it is time to display the data. The example below is from `TableExample.cs`. It is a table of stock prices. There are 6 columns.

```
// open stock daily price
StreamReader Reader = new StreamReader("SP500.csv");

// ignore header
Reader.ReadLine();

// read all daily prices
for(;;)
	{
	String TextLine = Reader.ReadLine();
	if(TextLine == null) break;

	String[] Fld = TextLine.Split(new Char[] {','});

	Table.Cell[ColDate].Value = Fld[ColDate];
	Table.Cell[ColOpen].Value = Double.Parse(Fld[ColOpen], NFI.PeriodDecSep);
	Table.Cell[ColHigh].Value = Double.Parse(Fld[ColHigh], NFI.PeriodDecSep);
	Table.Cell[ColLow].Value = Double.Parse(Fld[ColLow], NFI.PeriodDecSep);
	Table.Cell[ColClose].Value = Double.Parse(Fld[ColClose], NFI.PeriodDecSep);
	Table.Cell[ColVolume].Value = Int32.Parse(Fld[ColVolume]);
	StockTable.DrawRow();
	}

StockTable.Close();
```

The `DrawRow(NewPage)` method has an optional argument `Boolean NewPage = false`. The default value is `false`. If you want the next row to be printed at the top of the next page, set the argument to `true`.

Interactive features example.

```
// set cell number 6 with web link
BookList.Cell[6].WebLink = WebLinkString;

// another way to set weblink
BookList.Cell[6].AnnotAction = new AnnotWebLink(WebLinkString);

// set cell with document link to chapter 3
BookList.Cell[6].AnnotAction = new AnnotLinkAction("Chapter3");

// play video
PdfDisplayMedia Omega = new PdfDisplayMedia(PdfEmbeddedFile.CreateEmbeddedFile(Document,
"Omega.mp4"));
BookList.Cell[6].AnnotAction = new AnnotDisplayMedia(Omega);

// play audio
PdfDisplayMedia RingSound = new PdfDisplayMedia(PdfEmbeddedFile.CreateEmbeddedFile(Document,
"Ring01.wav"));
BookList.Cell[6].AnnotAction = new AnnotDisplayMedia(RingSound);

// allow user to save or view embedded file
PdfEmbeddedFile EmbeddedFile = PdfEmbeddedFile.CreateEmbeddedFile(Document, "BookList.txt");
BookList.Cell[6].AnnotAction = new AnnotFileAttachment(EmbeddedFile, FileAttachIcon.NoIcon);
```

For more examples of data table source code look into `ArticleExample.cs` and `TableExample.cs`. For more detail documentation of `PdfTable`, `PdfTableCell`, `PdfTableStyle` and `PdfTableBorder` classes look into `PdfFileWriter.chm` help file.

2.13. Play Video Files

The `PdfFileWriter` supports embedding video files in the PDF document. Full examples of playing video files are given in the page 7 of `OtherExample.cs`. Adding a video file requires the use of three classes. First you need to embed the video file in the PDF

document.

Second you need to define how the video is to be played. The `PdfDisplayMedia` class has a number of methods to control the video display. Please refer to the class' source code and the documentation help file. For example: `RepeatCount` or `ScaleMedia`. If you want to play the video in a floating window you must use `SetMediaWindow` method.

Third you need to define the area on the PDF page that the user must click in order to activate the video. If you want to activate the video when the annotation area is visible, use `ActivateActionWhenPageIsVisible`.

```
// define annotation rectangle that has the same aspect ratio as the video
PdfRectangle AnnotRect = ImageSizePos.ImageArea(480, 360,
AreaLeft, AreaBottom, AreaRight - AreaLeft, AreaTop - AreaBottom, ContentAlignment.MiddleCenter);

// create display media object
PdfDisplayMedia DisplayMedia = new PdfDisplayMedia(PdfEmbeddedFile.CreateEmbeddedFile(Document,
"LooneyTunes.mp4"));

// create annotation object
PdfAnnotation Annotation = Page.AddScreenAction(AnnotRect, DisplayMedia);

// activate the video when the page becomes visible
// Annotation.ActivateActionWhenPageIsVisible(true);

// define X Object to paint the annotation area when the video is not playing
PdfXObject AnnotArea = AnnotationArea(AnnotRect.Width, AnnotRect.Height, Color.Lavender,
Color.Indigo, "Click here to play the video");
Annotation.Appearance(AnnotArea);
```

Floating window video display

```
// create display media object
PdfDisplayMedia DisplayMedia = new PdfDisplayMedia(PdfEmbeddedFile.CreateEmbeddedFile(Document,
"Omega.mp4"));

// activate display controls
DisplayMedia.DisplayControls(true);

// repeat video indefinitly
DisplayMedia.RepeatCount(0);

// display in floating window
DisplayMedia.SetMediaWindow(MediaWindow.Floating, 640, 360, WindowPosition.Center,
    WindowTitleBar.TitleBarWithCloseButton, WindowResize.KeepAspectRatio, "Floating Window
Example");

Double LineSpacing = ArialNormal.LineSpacing(12.0);
Double TextPosX = PosX + 0.5 * AreaWidth;
Double TextPosY = PosY + 0.5 * AreaHeight + LineSpacing;
Double TextWidth = Contents.DrawText(ArialNormal, 12.0, TextPosX, TextPosY, TextJustify.Center,
"Click this text to play video");
TextPosY -= LineSpacing;
Contents.DrawText(ArialNormal, 12.0, TextPosX, TextPosY, TextJustify.Center, "in floating window");

// create annotation object
PdfRectangle AnnotRect = new PdfRectangle(TextPosX - 0.5 * TextWidth, TextPosY -
ArialNormal.DescentPlusLeading(12.0),
    TextPosX + 0.5 * TextWidth, TextPosY + ArialNormal.AscentPlusLeading(12.0) + LineSpacing);
Page.AddScreenAction(AnnotRect, DisplayMedia);
```

2.14. Play Sound Files

The `PdfFileWriter` supports embedding sound files in the PDF document. Full example of playing sound file is given in the page 7 of `OtherExample.cs`. Embedding sound files is essentially the same as video files. The only obvious difference is that there is nothing to display.

```
// create embedded media file
PdfDisplayMedia DisplayMedia = new PdfDisplayMedia(PdfEmbeddedFile.CreateEmbeddedFile(Document,
"Ring01.wav"));
DisplayMedia.SetMediaWindow(MediaWindow.Hidden);
AnnotDisplayMedia RingSound = new AnnotDisplayMedia(DisplayMedia);

// display text area to activate the sound
Double LineSpacing = ArialNormal.LineSpacing(12.0);
Double TextPosX = PosX + 0.5 * AreaWidth;
Double TextPosY = PosY + 0.5 * AreaHeight + LineSpacing;
Contents.DrawTextWithAnnotation(Page, ArialNormal, 12.0, TextPosX, TextPosY, TextJustify.Center,
  DrawStyle.Normal, Color.Red, "Click this text to play", RingSound);
TextPosY -= LineSpacing;
Contents.DrawTextWithAnnotation(Page, ArialNormal, 12.0, TextPosX, TextPosY, TextJustify.Center,
  DrawStyle.Normal, Color.Red, "Ringing sound", RingSound);
```

2.15. Attach Data Files

The `PdfFileWriter` supports embedding data files in the PDF document. Full example of embedding file is given in the page 7 of `OtherExample.cs`. The user can save the files or display the files.

```
// create embedded media file
PdfEmbeddedFile EmbeddedFile = PdfEmbeddedFile.CreateEmbeddedFile(Document, "BookList.txt");
AnnotFileAttachment FileIcon = new AnnotFileAttachment(EmbeddedFile, FileAttachIcon.Paperclip);

// display text area to activate the file attachment
Double LineSpacing = ArialNormal.LineSpacing(12.0);
Double TextPosX = PosX + 0.5 * AreaWidth;
Double TextPosY = PosY + 0.5 * AreaHeight + LineSpacing;
Contents.DrawText(ArialNormal, 12.0, TextPosX, TextPosY, TextJustify.Center, "Right click on the
paper clip");
TextPosY -= LineSpacing;
Double TextWidth = Contents.DrawText(ArialNormal, 12.0, TextPosX, TextPosY, TextJustify.Center, "to
open or save the attached file");

// annotation
Double IconPosX = TextPosX + 0.5 * TextWidth + 0.1;
Double IconPosY = TextPosY;
PdfRectangle AnnotRect = new PdfRectangle(IconPosX, IconPosY, IconPosX + 0.15, IconPosY + 0.4);
Page.AddFileAttachment(AnnotRect, EmbeddedFile, FileAttachIcon.Paperclip);
TextPosY -= 2 * LineSpacing;
AnnotFileAttachment FileText = new AnnotFileAttachment(EmbeddedFile, FileAttachIcon.NoIcon);
Contents.DrawTextWithAnnotation(Page, ArialNormal, 12.0, TextPosX, TextPosY, TextJustify.Center,
  DrawStyle.Underline, Color.Red, "File attachment (right click)", FileText);
```

2.16. Reorder Pages

The `PdfFileWriter` appends new pages to the end of the page list. If you want to move a page from its current position to a new position use the following method.

```
// Source and destination index are zero based.
// Source must be 0 to PageCount - 1.
// Destination must be 0 to PageCount.
// If destination index is PageCount, the page will be the last page
```

```
// PageCount is a property of PdfDocument.
Document.MovePage(Int32 SourceIndex, Int32 DestinationIndex);
```

2.17. PDF Document Output

The PdfFileWriter creates PDF documents. The main class PdfDocument constructor gives you two choices to save the document. The first choice is to save the PDF file to a disk file. In this case you provide the constuctor with a file name. At the end of file creation, you call PdfDocument.CreateFile. This method writes the PDF to the file and closes the file.

```
// create main class
PdfDocument Document = new PdfDocument(PaperType.Letter, false, UnitOfMeasure.Inch, FileName);

// terminate
Document.CreateFile();
```

The second choice is a stream. You create a stream, either memory stream or a file stream, and you pass the stream as an argument to the PdfDocument constructor. After CreateFile method is executed, your stream contains the PDF document. Extract the document from the stream as appropriate to your application. You must close the stream in your application.

```
// create memory stream
MemoryStream PdfStream = new MemoryStream();

// create main class
PdfDocument Document = new PdfDocument(PaperType.Letter, false, UnitOfMeasure.Inch, PdfStream);

// terminate
Document.CreateFile();

// save the memory stream to a file
FileStream FS = new FileStream(FileName, FileMode.Create);
PdfStream.WriteTo(FS);
PdfStream.Close();
FS.Close();
```

2.18. Document Information Dictionary

The PDF document information dictionary is displayed by the PDF reader in the Description tab of the document properties. The information includes: Title, Author, Subject, Keywords, Created date and time, Modified date and time, the Application that produced the file, the PDF Producer. Including document information in your application is done as follows:

```
PdfInfo Info = new PdfInfo(Document);
Info.Title("Article Example");
Info.Author("Uzi Granot Granotech Limited");
Info.Keywords("PDF, .NET, C#, Library, Document Creator");
Info.Subject("PDF File Writer C# Class Library (Version 1.15.0)");
```

When `PdfInfo` object is created, four additional fields are added to the dictionary. You can override all of them in your code.

```
// set creation and modify dates
DateTime LocalTime = DateTime.Now;
Info.CreationDate(LocalTime);
Info.ModDate(LocalTime);

// set creator and producer
Info.Creator("PdfFileWriter C# Class Library Version " + PdfDocument.RevisionNumber);
Info.Producer("PdfFileWriter C# Class Library Version " + PdfDocument.RevisionNumber);
```

2.19. Memory Control

As a document is being built, the PDF File Writer accumulates all the information required to create the PDF file. The information is kept in memory except for images and embedded files. Images and embedded files are automatically written to the output file when they are declared. For very large documents the memory used keeps growing. The library offers methods (`CommitToPdfFile`) to write contents information to the output file and invoke the garbage collector to free the unused memory. The `GC.Collect` method takes time to execute. If execution time is an issue, set the `GCCollect` argument once every few pages. In other words, the `CommitToPdfFile` must run for every page but the cleanup is done once every few pages. Once a commit was executed, no additional information can be added. `PdfTable` automatically starts a new page when the next row cannot fit at the bottom of the current page. The `PdfTable` class has two members `CommitToPdfFile` and `CommitGCCollectFreq` to control memory usage while a table is being build. The `PdfChart` class generates an image from the .NET Chart class. The `DrawChart` method of `PdfContents` will perform the commit. Alternatively, you can call `CommitToPdfFile` method of `PdfChart`.

```
// PdfContents, PdfXObject and PdfTilingPattern
// Commit the contents to output file.
// Once committed, this PdfContents cannot be used.
// The argument is GCCollect
Contents.CommitToPdfFile(true);
```

2.20 Windows Presentation Foundation WPF

Drawing WPF graphics is a four steps process.

- Step 1: Create `DrawWPFPath` object. Input arguments are path and Y axis direction.
- Step 2: Optionally add a brush by calling one of seven `SetBrush` methods or call `UseCurrentBrush` method.
- Step 3: Optionally add a pen by calling one of two `SetPen` methods or call `UseCurrentPen` method.
- Step 4: Draw the graphics object by calling `PdfContents.DrawWPFPath`.

If `System.Windows.Media` reference is not available (i.e. your application is Windows Form), you need to add `PresentationCore` and `WindowsBase` assemblies to your application.

Programming example of drawing with WPF graphics classes is given in `OtherExample.cs` Example8e and Example8f.

If no brush and no pen are defined the draw graphics is taken as a clip path.

Below you will find more details how to use the DrawWPFPath class for both WPF applications and other applications.

`DrawWPFPath` constructor takes two arguments: path and Y axis direction. Path is either a string or `System.Windows.Media.PathGeometry`. The text string is defined in Path Markup Syntax. The path geometry class is described in PathGeometry Class. The Y axis direction is down when the path is defined for WPF environment. The Y axis is up for PDF environment.

NOTE to programmers in any part of the world that use number decimal separator other than period. The text string input representing a path must be constructed exactly as define in Path Markup Syntax. Decimal numbers with fraction must use period regardless of world region. If you use optional separator between x and y values, it must be a comma. If the string is generated by another application make sure that the PathGeometry ToString is called with: PathGeometry.ToString(System.Globalization.CultureInfo.InvariantCulture). In other words, The Microsoft's PathGeometry.Parse method will fail to read a text string produce in Italy, for example, by PathGeometry.ToString method without IFormatProvider set to InvariantCulture.

There are three methods to define a brush for WPF applications. All of these methods will set the brush opacity at the same time.

- `System.Windows.Media.SolidColorBrush` see SolidColorBrush Class
- `System.Windows.Media.LinearGradientBrush` see LinearGradientBrush Class
- `System.Windows.Media.RadialGradientBrush` see RadialGradientBrush Class

There are five methods to define a brush for all applications. All of these methods will set the brush opacity at the same time.

- Set brush to `System.Drawing.Color`.
- Set brush to `PdfAxialShading`
- Set brush to `PdfRadialShading`
- Set brush to `PdfTilingPattern`
- Set brush to `UseCurrentBrush`

If you want the `DrawWPFPath` class to set the brush to the currently selected brush, call `UseCurrentBrush` method.

There is one method to define a pen for WPF applications. Call `SetPen` method with `System.Windows.Media.Pen` class (See Pen Class). Note, the `Pen.Brush` property must be `SolidColorBrush`. The pen class contains all required information to draw a line, such as color and width.

There is one method to define a pen for all applications. Call `SetPen` with `System.Drawing.Color` argument. The color argument defines alpha, red, green and blue components. To set other pen characteristics set any or all of the following properties and methods:

- `SetPenWidth`
- `DashArray`
- `DashPhase`
- `LineCap`
- `LineJoin`
- `MiterLimit`

If you want the `DrawWPFPath` class to set the pen to the currently selected pen call `UseCurrentPen` method.

Once `DrawWPFPath` class is set with all information needed to draw the path, call `PdfContents.DrawWPFPath` method. The DrawWPFPath class calculates the transformation matrix required to convert input path to drawing rectangle based on the path bounding box and the drawing rectangle and alignment. Alignment of zero (the default) will stretch the path to fit the drawing area. All other alignment values position the path within the drawing area according to the argument enumeration value.

```
public void DrawWPFPath(
    DrawWPFPath Path,   // path to be drawn
    double OriginX,     // Drawing rectangle in user units left side
    double OriginY,     // Drawing rectangle in user units bottom side
    double Width,       // Drawing rectangle in user units width
    double Height,      // Drawing rectangle in user units height
                            // path alignment within drawing rectangle
                            // Alignment=0 means the path will be stretched
                            // in either horizontal or vertical direction
                            // to fit the drawing rectangle
    ContentAlignment Alignment = 0)
```

2.21 Transparency, Opacity, Alpha Color Component and Blending

By default the PDF imaging model paints objects (shapes, lines, text and images) opaquely onto the page. Each new object completely obscure the image under it. PDF has two mechanisms to change this behavior opacity and blending. The graphic state dictionary has opacity value for stroking (pen) and non-stroking (brush) operations. The opacity value of fully opaque is 1.0 and fully transparent is 0.0. The opacity value corresponds to the alpha component of a color structure such that 1.0 is 255 alpha and 0.0 is 0 alpha. If the opacity value is 0.5 a new object painted on the page will be 50% transparent. To set opacity call the `SetAlphaStroking` method for lines or `SetAlphaNonStroking` method for shapes. Blending is a process of combining the color on the page with the color of the new item being painted. To set a bland mode call the `SetBlendMode` method of `PdfContents`. The argument is `BlendMode` enumeration. For full description please refer to section 7.2.4 Blend Mode of the PDF specifications document. For example please refer to `OtherExample.cs` page 8.

2.22. Document Links and Named Destination

Document links allow PDF document users to click on the link and jump to another part of the document. Adding document links is done in two parts. The destination is defined as a location marker. Location marker must have a unique name, a scope (`LocalDest` or

NamedDest), and document location (page and position). NamedDest Scope can be used for either document link or for named destination or both . The second part is the link location. The two parts can be defined in any order. They are tied together by the name. The name is case sensitive. Many links can point to the same location marker.

Named destinations are targets within the PDF document. They are defined with location marker in the same way as document links. The scope has to be set to NamedDest. When a PDF reader such as Adobe Acrobat opens a PDF document it can open the document while displaying the target in the viewing window.

To embed a location marker call the AddLocationMarker method of PdfPage. Note: Name is case sensitive.

```
// Add location marker to the document (PdfPage method)
public void AddLocationMarker
  (
  string          LocMarkerName, // unique destination name (case sensitive)
  LocMarkerScope  Scope,         // eigther LocalDest or NamedDest
  DestFit         FitArg,        // fit argument (see below)
  params double[] SideArg        // 0, 1 or 4 side dimension argument (see below)
  )
```

To add link location call AddLinkLocation method of PdfPage.

```
public PdfAnnotation AddLinkAction
  (
  string LocMarkerName,    // location marker name
  PdfRectangle AnnotRect   // rectangle area on the page to activate the jump
  )
```

For more information about named destinations please refer to Adobe PDF file specification "PDF Reference, Sixth Edition, Adobe Portable Document Format Version 1.7 November 2006". Table 8.2 on page 582.

- DestFit.Fit (no arg): Display the page, with its contents magnified just enough to fit the entire page within the window both horizontally and vertically.
- DestFit.FitH (1 arg Top): Display the page, with the vertical coordinate top positioned at the top edge of the window and the contents of the page magnified just enough to fit the entire width of the page within the window.
- DestFit.FitV (1 arg Left): Display the page, with the horizontal coordinate left positioned at the left edge of the window and the contents of the page magnified just enough to fit the entire height of the page within the window.
- DestFit.FitR (4 args Left Bottom Right Top): Display the page, with its contents magnified just enough to fit the rectangle specified by the coordinates left, bottom, right, and topentirely within the window both horizontally and vertically.
- DestFit.FitB (No arg): Display the page, with its contents magnified just enough to fit its bounding box entirely within the window both horizontally and vertically.
- DestFit.FitBH (1 arg Top): Display the page, with the vertical coordinate top positioned at the top edge of the window and the contents of the page magnified just enough to fit the entire width of its bounding box within the window.
- DestFit.FitBV (1 arg Left): Display the page, with the horizontal coordinate left positioned at the left edge of the window and the contents of the page magnified just enough to fit the entire height of its bounding box within the window.

The PDF reader's calling parameters are defined in Parameters for Opening PDF Files by Adobe. If the PDF is opened on a desktop computer the calling line must be:

"path\AcroRd32.exe" /A "nameddest=ChapterXX" "path\Document.pdf"

If the PDF document is pointed by a link in a web page, the destination is appended to the link:

Target description

Or: Target description

2.23. Encryption Support

The PDF File Writer library provides support to AES 128 and Standard 128 (RC4) encryption. For more information please refer to PDF Reference sixth edition (Version 1.7) section 3.5 Encryption. The PDF File writer supports two types of encryption filters, the AES-128 and

Standard 128. The Standard 128 is RC4 encryption. It is considered unsafe. For new project do not use it. It does not support public key security to encode recipient list.

To encrypt your PDF document call one of four `SetEncryption` methods defined in `PdfDocument` class:

Set Encryption with no arguments.

The PDF File Writer library will encrypt the PDF document using AES-128 encryption. The PDF reader will open the document without requesting a password. Permissions flags are set to allow all.

```
Document.SetEncryption();
```

Set Encryption with one argument.

The PDF File Writer library will encrypt the PDF document using AES-128 encryption. The argument is permissions. Permission flags are defined below. You can or together more than one permission. The PDF reference manual has full description of permissions. The PDF reader will open the document without requesting a password.

```
Document.SetEncryption(Permission Permissions);
```

Set Encryption with two arguments.

The PDF File Writer library will encrypt the PDF document using AES-128 encryption. The two arguments are user password and permissions. The PDF reader will open the document with user password. Permissions will be set as per argument.

```
Document.SetEncryption(String UserPassword, Permission Permissions);
```

Set Encryption with four arguments.

The PDF File Writer library will encrypt the PDF document using either EncryptionType.Aes128 encryption or EncryptionType.Standard128 encryption. The four arguments are user password, owner password, permissions and encryption type. If user password is null, the default password will be taken. If owner password in null, the software will generate random number password. The Standard128 encryption is considered unsafe. It should not be used for new projects.

A PDF reader such as Acrobat will accept either user or owner password. If owner password is used to open document, the PDF reader will open it with all permissions set to allow operation.

```
Document.SetEncryption(String UserPassword, String OwnerPassword, Permission Permissions,
EncryptionType Type);
```

Permission flags are as follows:

```
// Full description is given in
// PDF Reference Version 1.7 Table 3.20
public enum Permission
    {
    None = 0,
    LowQalityPrint = 4, // bit 3
    ModifyContents = 8, // bit 4
    ExtractContents = 0x10, // bit 5
    Annotation = 0x20,  // bit 6
    Interactive = 0x100,    // bit 9
    Accessibility = 0x200,  // bit 10
    AssembleDoc = 0x400,    // bit 11
    Print = 0x804,      // bit 12 + bit 3
    All = 0xf3c,        // bits 3, 4, 5, 6, 9, 10, 11, 12
    }
```

2.24. Sticky Notes or Text Annotations.

The PDF reference document defines Sticky Notes or Text Annotation in Section 8.4 page 621. "A text annotation represents a "sticky note" attached to a point in the PDF document. When closed, the annotation appears as an icon; when open, it displays a pop-up window containing the text of the note in a font and size chosen by the viewer application. Text annotations do not scale and rotate with the page; they behave as if the NoZoom and NoRotate annotation flags (see Table 8.16 on page 608) were always set. Table 8.23 shows the annotation dictionary entries specific to this type of annotation."

Adding a sticky note to your document is very simple. You add one single line of code. The sticky note is added to a `PdfPage` object. It is not part of the page contents. The position of the sticky note is an absolute page location measured from the bottom left corner of the page to the top left corner of the sticky note icon. The text string is the content of the pop-up window. The stick note argument is one of enumeration items below.

```
// sticky note text annotation
Page.AddStickyNote(PageAbsPosX, PageAbsPosY,
    "My first sticky note", StickyNoteIcon.Note);
```

```
// Sticky note icon
public enum StickyNoteIcon
    {
    // Comment (note: no icon)
    Comment,

    // Key
    Key,

    // Note (default)
    Note,

    // Help
    Help,

    // New paragraph
    NewParagraph,

    // Paragraph
    Paragraph,

    // Insert
    Insert,
    }
```

2.25. Layers or Optional Content.

The PDF specification document (section 4.10) defines optional contents as follows. "Optional content (PDF 1.5) refers to sections of content in a PDF document that can be selectively viewed or hidden by document authors or consumers. This capability is useful in items such as CAD drawings, layered artwork, maps, and multi-language documents."

Adobe Acrobat viewer has navigation panels on the left side of the screen. One of them is the layers panel. If a PDF document uses layers, the layers control switches will be displayed in this panel. The user can display or hide items that were attached to these layer controls.

Adding layers to your PDF document. The full example is given in `LayersExample.cs`. In addition `OtherExample.cs` has examples of using layers to control images and annotations.

Create the main layers control object. One per document.

```
// create a layers control object and give it a name.
// only one object like that is allowed.
// The name will be displayed in the layer panel.
PdfLayers Layers = new PdfLayers(Document, "PDF layers group");
```

Set the list mode option. The default is all pages

```
// List mode
Layers.ListMode = ListMode.AllPages; // default

// or
Layers.ListMode = ListMode.VisiblePages;
```

Create one or more layers objects. Each one will correspond to one check box on the layer panel. Each one can control one or many displayed items.

```
// create one or more layer objects
PdfLayer LayerName = new PdfLayer(Layers, "Layer name");
```

A number of layers can be combined into radio buttons group. One group of radio buttons can be all off or just one layer on.

```
// Optionally combine three layers into
// one group of radio buttons
LayerName1.RadioButton = "Group1";
LayerName2.RadioButton = "Group1";
```

Set the order of layers in the layer pane. If DisplayOrder method is not used, the program will list all layers specified above on the same main level. If DisplayOrder method is used, all layers must be included

The list of layers can have sub-groups with optional name

```
// display order
Layers.DisplayOrder(LayerName1);
Layers.DisplayOrder(LayerName2);
Layers.DisplayOrder(LayerName3);
Layers.DisplayOrderStartGroup("Sub Group");
Layers.DisplayOrder(LayerName4);
Layers.DisplayOrder(LayerName5);
Layers.DisplayOrder(LayerName6);
Layers.DisplayOrderEndGroup();
```

Define an area within contents stream to be controlled by layer

```
// contents stream start layer marker
Contents.LayerStart(LayerName1);

// your contents methods to be
// controlled by LayerName1

// end of LayerName1 area
Contents.LayerEnd();
```

Control an image or annotation directly. The image can be anywhere within the document.

```
// image or annotation control
QREncoder QREncoder = new QREncoder();
QREncoder.ErrorCorrection = ErrorCorrection.M;
QREncoder.Encode("Some data");
PdfImage QRImage = new PdfImage(Document);
QRImage.LayerControl = LayerName1;
QRImage.LoadImage(QREncoder);
```

2.26. Initial Document Display.

The initial document display controls the appearance of your document when it is displayed by the PDF viewer (Acrobat). It controls the left pane of the screen.

For example, open the bookmarks pane.

```
Document.InitialDocDisplay = InitialDocDisplay.UseBookmarks;
```

```
// initial document display enumeration
public enum InitialDocDisplay
{
// keep the left pane closed
UseNone,

// open the bookmarks pane
UseBookmarks,

// open the page thumbnails
UseThumbs,

// full screen mode
FullScreen,

// open layers
UseLayers,

// open attachments
UseAttachments,
}
```

2.27. XMP Metadata.

The XMP file or byte array are embedded in a metadata stream contained in a PDF object. The XMP must be encoded as UTF-8. The PdfFileWriter includes the input file or input byte array as given by the user. The user must ensure the the XMP input is a valid metafile. The XMP stream is not compressed or encrypted. This allows readers to get the metadata information with little programming. You should include the XMP matadata shortly after the PdfDocument is created and before any image is loaded. By doing so the metadata will be at the start of the file and it will be readable by simple text editors.

```
// adding metadata
new PdfMatadata(Document, FileName);

// or
new PdfMetadata(Document, ByteArray);
```

3. Development Guide by Example

This section describes the integration of the PDF File Writer C# class library to your application. The test program `TestPdfFileWriter` program is a simulation of your own application. When you press on the “Article Example” button, the program executes the code in `ArticleExample.cs` source file. The image above displays the resulted PDF file. This method demonstrates the creation of one page document with some text and graphics. After going through this example, you should have a good understanding of the process. The other example buttons produce a variety of PDF documents. In total, practically every feature of this library is demonstrated by these examples.

The Debug check box, if checked, will create a PDF file that can be viewed with a text editor but cannot be loaded to a PDF reader. The resulted file is not compressed and images and font file are replaced with text place holder. The Debug check box should be used for debugging only.

The TestPdfFileWriter program was developed using Microsoft Visual C# 2012. It was tested for Windows XP, Vista, 7 and 8.

3.1. Document Creation Overview

The Test method below demonstrates the six steps described in the introduction for creating a PDF file. The method will be executed when you press on the “Article Example” button of the demo program. The following subsections describe in detail each step.

```
// Create article's example test PDF document
public void Test
        (
        bool Debug,
        string FileName
        )
{
    {
    // Step 1: Create empty document
    // Arguments: page width: 8.5", page height: 11", Unit of measure: inches
    // Return value: PdfDocument main class
    Document = new PdfDocument(PaperType.Letter, false, UnitOfMeasure.Inch, FileName);

    // for encryption test
    //      Document.SetEncryption(null, null, Permission.All & ~Permission.Print,
```

```
EncryptionType.Aes128);

    // Debug property
    // By default it is set to false. Use it for debugging only.
    // If this flag is set, PDF objects will not be compressed, font and images will be replaced
    // by text place holder. You can view the file with a text editor but you cannot open it with
PDF reader.
    Document.Debug = Debug;

    PdfInfo Info = PdfInfo.CreatePdfInfo(Document);
    Info.Title("Article Example");
    Info.Author("Uzi Granot");
    Info.Keywords("PDF, .NET, C#, Library, Document Creator");
    Info.Subject("PDF File Writer C# Class Library (Version 1.21.0)");

    // Step 2: create resources
    // define font resources
    DefineFontResources();

    // define tiling pattern resources
    DefineTilingPatternResource();

    // Step 3: Add new page
    Page = new PdfPage(Document);

    // Step 4:Add contents to page
    Contents = new PdfContents(Page);

    // Step 5: add graphices and text contents to the contents object
    DrawFrameAndBackgroundWaterMark();
    DrawTwoLinesOfHeading();
    DrawHappyFace();
    DrawBarcode();
    DrawPdf417Barcode();
    DrawImage();
    DrawChart();
    DrawTextBox();
    DrawBookOrderForm();

    // Step 6: create pdf file
    Document.CreateFile();

    // start default PDF reader and display the file
    Process Proc = new Process();
    Proc.StartInfo = new ProcessStartInfo(FileName);
    Proc.Start();

    // exit
    Return;
}
```

3.2. Font Resources

The `DefineFontResources` method creates all the font resources used in this example. To see all the characters available for any font, press the button “Font Families”. Select a family and view the glyphs defined for each character. To view individual glyph press view or double click.

```
// Define Font Resources
private void DefineFontResources()
{
    // Define font resources
    // Arguments: PdfDocument class, font family name, font style, embed flag
```

```
    // Font style (must be: Regular, Bold, Italic or Bold | Italic) All other styles are invalid.
    // Embed font. If true, the font file will be embedded in the PDF file.
    // If false, the font will not be embedded
    string FontName1 = "Arial";
    string FontName2 = "Times New Roman";

    ArialNormal = PdfFont.CreatePdfFont(Document, FontName1, FontStyle.Regular, true);
    ArialBold = PdfFont.CreatePdfFont(Document, FontName1, FontStyle.Bold, true);
    ArialItalic = PdfFont.CreatePdfFont(Document, FontName1, FontStyle.Italic, true);
    ArialBoldItalic = PdfFont.CreatePdfFont(Document, FontName1, FontStyle.Bold | FontStyle.Italic,
true);
    TimesNormal = PdfFont.CreatePdfFont(Document, FontName2, FontStyle.Regular, true);
    Comic = PdfFont.CreatePdfFont(Document, "Comic Sans MS", FontStyle.Bold, true);
    return;
}
```

3.3. Tiling Pattern Resource

ThThe `DefineTilingPatternResource` method defines background pattern resource for the example area. The pattern is the word “PdfFileWriter” in white over light blue background. The pattern is made of two lines of repeating the key word. The two lines are skewed by half word length.

If you want to find interesting patterns, search the internet for catalogs of companies making floor or wall tiles.p>

```
// Define Tiling Pattern Resource
private void DefineTilingPatternResource()
{
    // create empty tiling pattern
    WaterMark = new PdfTilingPattern(Document);

    // the pattern will be PdfFileWriter laied out in brick pattern
    string Mark = "PdfFileWriter";

    // text width and height for Arial bold size 18 points
    double FontSize = 18.0;
    double TextWidth = ArialBold.TextWidth(FontSize, Mark);
    double TextHeight = ArialBold.LineSpacing(FontSize);

    // text base line
    double BaseLine = ArialBold.DescentPlusLeading(FontSize);

    // the overall pattern box (we add text height value as left and right text margin)
    double BoxWidth = TextWidth + 2 * TextHeight;
    double BoxHeight = 4 * TextHeight;
    WaterMark.SetTileBox(BoxWidth, BoxHeight);

    // save graphics state
    WaterMark.SaveGraphicsState();

    // fill the pattern box with background light blue color
    WaterMark.SetColorNonStroking(Color.FromArgb(230, 244, 255));
    WaterMark.DrawRectangle(0, 0, BoxWidth, BoxHeight, PaintOp.Fill);

    // set fill color for water mark text to white
    WaterMark.SetColorNonStroking(Color.White);

    // draw PdfFileWriter at the bottom center of the box
    WaterMark.DrawText(ArialBold, FontSize, BoxWidth / 2, BaseLine, TextJustify.Center, Mark);

    // adjust base line upward by half height
    BaseLine += BoxHeight / 2;

    // draw the right half of PdfFileWriter shifted left by half width
```

```
    WaterMark.DrawText(ArialBold, FontSize, 0.0, BaseLine, TextJustify.Center, Mark);

    // draw the left half of PdfFileWriter shifted right by half width
    WaterMark.DrawText(ArialBold, FontSize, BoxWidth, BaseLine, TextJustify.Center, Mark);

    // restore graphics state
    WaterMark.RestoreGraphicsState();
    return;
}
```

3.4. Draw Frame with Background Pattern

The `DrawFrameAndBackgroundWaterMark` method draws a frame around the example area with background water mark pattern. The pattern resource was define in the previous subsection.

```
// Draw frame around example area
private void DrawFrameAndBackgroundWaterMark()
{
    // save graphics state
    Contents.SaveGraphicsState();

    // Draw frame around the page
    // Set line width to 0.02"
    Contents.SetLineWidth(0.02);

    // set frame color dark blue
    Contents.SetColorStroking(Color.DarkBlue);

    // use water mark tiling pattern to fill the frame
    Contents.SetPatternNonStroking(WaterMark);

    // rectangle position: x=1.0", y=1.0", width=6.5", height=9.0"
    Contents.DrawRectangle(1.0, 1.0, 6.5, 9.0, PaintOp.CloseFillStroke);

    // restore graphics sate
    Contents.RestoreGraphicsState();

    // draw article name under the frame
    // Note: the \u00a4 is character 164 that was substituted during Font resource definition
    // this character is a solid circle it is normally Unicode 9679 or \u25cf in the Arial family
    Contents.DrawText(ArialNormal, 9.0, 1.1, 0.85, "PdfFileWriter \u25cf PDF File Writer C# Class
Library \u25cf Author: Uzi Granot");

    // draw web link to the article
    Contents.DrawWebLink(Page, ArialNormal, 9.0, 7.4, 0.85, TextJustify.Right,
    DrawStyle.Underline, Color.Blue, "Click to view article", ArticleLink);

    return;
}
```

3.5. Draw Two Lines of Heading

The `DrawTwoLinesOfHeading` method draws two heading lines at the center of the page. The first line is drawing text with outline special effect.

```
// Draw heading
private void DrawTwoLinesOfHeading()
{
    // page heading
    // Arguments: Font: ArialBold, size: 36 points, Position: X = 4.25", Y = 9.5"
```

```
    // Text Justify: Center (text center will be at X position)
    // Stoking color: R=128, G=0, B=255 (text outline)
    // Nonstroking color: R=255, G=0, B=128 (text body)
    Contents.DrawText(Comic, 40.0, 4.25, 9.25, TextJustify.Center, 0.02, Color.FromArgb(128, 0,
255), Color.FromArgb(255, 0, 128), "PDF FILE WRITER");

    // save graphics state
    Contents.SaveGraphicsState();

    // change nonstroking (fill) color to purple
    Contents.SetColorNonStroking(Color.Purple);

    // Draw second line of heading text
    // arguments: Handwriting font, Font size 30 point, Position X=4.25", Y=9.0"
    // Text Justify: Center (text center will be at X position)
    Contents.DrawText(Comic, 30.0, 4.25, 8.75, TextJustify.Center, "Example");

    // restore graphics sate (non stroking color will be restored to default)
    Contents.RestoreGraphicsState();
    return;
}
```

3.6. Draw Happy Face

The `DrawHappyFace` method is an example of drawing oval and constructing path from a line and Bezier curve.

```
// Draw Happy Face
private void DrawHappyFace()
{
    // save graphics state
    Contents.SaveGraphicsState();

    // translate coordinate origin to the center of the happy face
    Contents.Translate(4.25, 7.5);

    // change nonstroking (fill) color to yellow
    Contents.SetColorNonStroking(Color.Yellow);

    // draw happy face yellow oval
    Contents.DrawOval(-1.5, -1.0, 3.0, 2.0, PaintOp.Fill);

    // set line width to 0.2" this is the black circle around the eye
    Contents.SetLineWidth(0.2);

    // eye color is white with black outline circle
    Contents.SetColorNonStroking(Color.White);
    Contents.SetColorStroking(Color.Black);

    // draw eyes
    Contents.DrawOval(-0.75, 0.0, 0.5, 0.5, PaintOp.CloseFillStroke);
    Contents.DrawOval(0.25, 0.0, 0.5, 0.5, PaintOp.CloseFillStroke);

    // mouth color is black
    Contents.SetColorNonStroking(Color.Black);

    // draw mouth by creating a path made of one line and one Bezier curve
    Contents.MoveTo(-0.6, -0.4);
    Contents.LineTo(0.6, -0.4);
    Contents.DrawBezier(0.0, -0.8, 0, -0.8, -0.6, -0.4);

    // fill the path with black color
    Contents.SetPaintOp(PaintOp.Fill);
```

```
    // restore graphics sate
    Contents.RestoreGraphicsState();
    return;
}
```

3.7. Draw Barcodes

The `DrawBarcode` method is an example of drawing two barcodes EAN-13 and Code-128

```
// Draw Barcode
private void DrawBarcode()
{
    // save graphics state
    Contents.SaveGraphicsState();

    // draw EAN13 barcode
    BarcodeEAN13 Barcode1 = new BarcodeEAN13("1234567890128");
    Contents.DrawBarcode(1.3, 7.05, 0.012, 0.75, Barcode1, ArialNormal, 8.0);

    // create QRCode barcode
    QREncoder QREncoder = new QREncoder();

    // set error correction code
    QREncoder.ErrorCorrection = ErrorCorrection.M;

    // set module size in pixels
    QREncoder.ModuleSize = 1;

    // set quiet zone in pixels
    QREncoder.QuietZone = 4;

    // encode your text or byte array
    QREncoder.Encode(ArticleLink);

    // convert QRCode to black and white image
    PdfImage BarcodeImage = new PdfImage(Document);
    BarcodeImage.LoadImage(QREncoder);

    // draw image (height is the same as width for QRCode)
    Contents.DrawImage(BarcodeImage, 6.0, 6.8, 1.2);

    // define a web link area coinsiding with the qr code
    Page.AddWebLink(6.0, 6.8, 7.2, 8.0, ArticleLink);

    // restore graphics sate
    Contents.RestoreGraphicsState();
    return;
}
```

3.8. Draw PDF417 Barcode

```
// Draw Barcode
private void DrawPdf417Barcode()
{
    // save graphics state
    Contents.SaveGraphicsState();

    // create PDF417 barcode
    Pdf417Encoder Pdf417 = new Pdf417Encoder();
    Pdf417.DefaultDataColumns = 3;
```

```
    Pdf417.Encode(ArticleLink);
    Pdf417.WidthToHeightRatio(2.5);

    // convert Pdf417 to black and white image
    PdfImage BarcodeImage = new PdfImage(Document);
    BarcodeImage.LoadImage(Pdf417);

    // draw image
    Contents.DrawImage(BarcodeImage, 1.1, 5.2, 2.5);

    // define a web link area coinsiding with the qr code
    double Height = 2.5 * Pdf417.ImageHeight / Pdf417.ImageWidth;
    Page.AddWebLink(1.1, 5.2, 1.1 + 2.5, 5.2 + Height, ArticleLink);

    // restore graphics sate
    Contents.RestoreGraphicsState();
    return;
}
```

3.9. Draw Image and Clip it

The `Draw`Image method is an example of drawing an image. The `PdfFileWriter` support drawing images stored in all image files supported by `Bitmap` class and `Metafile` class. The `ImageFormat` class defines all image types. The JPEG image file type is the native image format of the PDF file. If you call the `PdfImage` constructor with JPEG file, the program copies the file as is into the PDF file. If you call the `PdfImage` constructor with any other type of image file, the program converts it into JPEG file. In order to keep the PDF file size as small as possible, make sure your image file resolution is not unreasonably high.

The `PdfImage` class loads the image and calculates maximum size that can fit a given image size in user coordinates and preserve the original aspect ratio. Before drawing the image we create an oval clipping path to clip the image.

```
// Draw image and clip it
private void DrawImage()
{
    // define local image resources
    // resolution 96 pixels per inch, image quality 50%
    PdfImage Image1 = new PdfImage(Document);
    Image1.Resolution = 96.0;
    Image1.ImageQuality = 50;
    Image1.LoadImage("TestImage.jpg");

    // save graphics state
    Contents.SaveGraphicsState();

    // translate coordinate origin to the center of the picture
    Contents.Translate(3.75, 5.0);

    // adjust image size and preserve aspect ratio
    PdfRectangle NewSize = Image1.ImageSizePosition(1.75, 1.5, ContentAlignment.MiddleCenter);

    // clipping path
    Contents.DrawOval(NewSize.Left, NewSize.Bottom, NewSize.Width, NewSize.Height,
PaintOp.ClipPathEor);

    // draw image
    Contents.DrawImage(Image1, NewSize.Left, NewSize.Bottom, NewSize.Width, NewSize.Height);

    // restore graphics state
    Contents.RestoreGraphicsState();
    return;
}
```

3.10. Draw Pie Chart

The `DrawChart` method is an example of defining a chart and drawing it to the PDF document.

```
// Draw chart
private void DrawChart()
{
    // save graphics state
    Contents.SaveGraphicsState();

    // create chart
    Chart PieChart = PdfChart.CreateChart(Document, 1.8, 1.5, 300.0);

    // create PdfChart object with Chart object
    PdfChart PiePdfChart = new PdfChart(Document, PieChart);
    PiePdfChart.SaveAs = SaveImageAs.IndexedImage;

    // make sure we have good quality image
    PieChart.AntiAliasing = AntiAliasingStyles.None; //.All;

    // set colors
    PieChart.BackColor = Color.FromArgb(220, 220, 255);
    PieChart.Palette = ChartColorPalette.BrightPastel;

    // default font
    Font DefaultFont = PiePdfChart.CreateFont("Verdana", FontStyle.Regular, 0.05,
FontSizeUnit.UserUnit);
    Font TitleFont = PiePdfChart.CreateFont("Verdana", FontStyle.Bold, 0.07,
FontSizeUnit.UserUnit);

    // title (font size is 0.25 inches)
    Title Title1 = new Title("Pie Chart Example", Docking.Top, TitleFont, Color.Purple);
    PieChart.Titles.Add(Title1);

    // legend
    Legend Legend1 = new Legend();
    PieChart.Legends.Add(Legend1);
    Legend1.BackColor = Color.FromArgb(230, 230, 255);
    Legend1.Docking = Docking.Bottom;
    Legend1.Font = DefaultFont;

    // chart area
    ChartArea ChartArea1 = new ChartArea();
    PieChart.ChartAreas.Add(ChartArea1);

    // chart area background color
    ChartArea1.BackColor = Color.FromArgb(255, 200, 255);

    // series 1
    Series Series1 = new Series();
    PieChart.Series.Add(Series1);
    Series1.ChartType = SeriesChartType.Pie;
    Series1.Font = DefaultFont;
    Series1.IsValueShownAsLabel = true;
    Series1.LabelFormat = "{0} %";

    // series values
    Series1.Points.Add(22.0);
    Series1.Points[0].LegendText = "Apple";
    Series1.Points.Add(27.0);
    Series1.Points[1].LegendText = "Banana";
    Series1.Points.Add(33.0);
    Series1.Points[2].LegendText = "Orange";
```

```
    Series1.Points.Add(18.0);
    Series1.Points[3].LegendText = "Grape";

    Contents.DrawChart(PiePdfChart, 5.6, 5.0);

    // restore graphics state
    Contents.RestoreGraphicsState();
    return;
}
```

3.11. Draw Text Box

The `DrawTextBox` method is an example of using the `TextBox` class. The `TextBox` class formats text to fit within a column. The text can be drawn using a verity of font's styles and sizes.

```
// Draw example of a text box
private void DrawTextBox()
{
    // save graphics state
    Contents.SaveGraphicsState();

    // translate origin to PosX=1.1" and PosY=1.1" this is the bottom left corner of the text box
example
    Contents.Translate(1.1, 1.1);

    // Define constants
    // Box width 3.25"
    // Box height is 3.65"
    // Normal font size is 9.0 points.
    const double Width = 3.15;
    const double Height = 3.65;
    const double FontSize = 9.0;

    // Create text box object width 3.25"
    // First line indent of 0.25"
    TextBox Box = new TextBox(Width, 0.25);

    // add text to the text box
    Box.AddText(ArialNormal, FontSize,
    "This area is an example of displaying text that is too long to fit within a fixed width " +
    "area. The text is displayed justified to right edge. You define a text box with the required "
+
    "width and first line indent. You add text to this box. The box will divide the text into " +
    "lines. Each line is made of segments of text. For each segment, you define font, font " +
    "size, drawing style and color. After loading all the text, the program will draw the formatted
text.\n");
    Box.AddText(TimesNormal, FontSize + 1.0, "Example of multiple fonts: Times New Roman, ");
    Box.AddText(Comic, FontSize, "Comic Sans MS, ");
    Box.AddText(ArialNormal, FontSize, "Example of regular, ");
    Box.AddText(ArialBold, FontSize, "bold, ");
    Box.AddText(ArialItalic, FontSize, "italic, ");
    Box.AddText(ArialBoldItalic, FontSize, "bold plus italic. ");
    Box.AddText(ArialNormal, FontSize - 2.0, "Arial size 7, ");
    Box.AddText(ArialNormal, FontSize - 1.0, "size 8, ");
    Box.AddText(ArialNormal, FontSize, "size 9, ");
    Box.AddText(ArialNormal, FontSize + 1.0, "size 10. ");
    Box.AddText(ArialNormal, FontSize, DrawStyle.Underline, "Underline, ");
    Box.AddText(ArialNormal, FontSize, DrawStyle.Strikeout, "Strikeout. ");
    Box.AddText(ArialNormal, FontSize, "Subscript H");
    Box.AddText(ArialNormal, FontSize, DrawStyle.Subscript, "2");
    Box.AddText(ArialNormal, FontSize, "O. Superscript A");
    Box.AddText(ArialNormal, FontSize, DrawStyle.Superscript, "2");
    Box.AddText(ArialNormal, FontSize, "+B");
```

```
    Box.AddText(ArialNormal, FontSize, DrawStyle.Superscript, "2");
    Box.AddText(ArialNormal, FontSize, "=C");
    Box.AddText(ArialNormal, FontSize, DrawStyle.Superscript, "2");
    Box.AddText(ArialNormal, FontSize, "\n");
    Box.AddText(Comic, FontSize, Color.Red, "Some color, ");
    Box.AddText(Comic, FontSize, Color.Green, "green, ");
    Box.AddText(Comic, FontSize, Color.Blue, "blue, ");
    Box.AddText(Comic, FontSize, Color.Orange, "orange, ");
    Box.AddText(Comic, FontSize, DrawStyle.Underline, Color.Purple, "and purple.\n");
    Box.AddText(ArialNormal, FontSize, "Support for non-Latin letters: ");
    Box.AddText(ArialNormal, FontSize, Contents.ReverseString("עברית"));
    Box.AddText(ArialNormal, FontSize, "АБВГДЕ");
    Box.AddText(ArialNormal, FontSize, "αβγδεζ");

    Box.AddText(ArialNormal, FontSize, "\n");

    // Draw the text box
    // Text left edge is at zero (note: origin was translated to 1.1")
    // The top text base line is at Height less first line ascent.
    // Text drawing is limited to vertical coordinate of zero.
    // First line to be drawn is line zero.
    // After each line add extra 0.015".
    // After each paragraph add extra 0.05"
    // Stretch all lines to make smooth right edge at box width of 3.15"
    // After all lines are drawn, PosY will be set to the next text line after the box's last
paragraph
    double PosY = Height;
    Contents.DrawText(0.0, ref PosY, 0.0, 0, 0.015, 0.05, TextBoxJustify.FitToWidth, Box);

    // Create text box object width 3.25"
    // No first line indent
    Box = new TextBox(Width);

    // Add text as before.
    // No extra line spacing.
    // No right edge adjustment
    Box.AddText(ArialNormal, FontSize,
    "In the examples above this area the text box was set for first line indent of " +
    "0.25 inches. This paragraph has zero first line indent and no right justify.");
    Contents.DrawText(0.0, ref PosY, 0.0, 0, 0.01, 0.05, TextBoxJustify.Left, Box);

    // Create text box object width 2.75
    // First line hanging indent of 0.5"
    Box = new TextBox(Width - 0.5, -0.5);

    // Add text
    Box.AddText(ArialNormal, FontSize,
    "This paragraph is set to first line hanging indent of 0.5 inches. " +
    "The left margin of this paragraph is 0.5 inches.");

    // Draw the text
    // left edge at 0.5"
    Contents.DrawText(0.5, ref PosY, 0.0, 0, 0.01, 0.05, TextBoxJustify.Left, Box);

    // restore graphics state
    Contents.RestoreGraphicsState();
    return;
}
```

3.12. Draw Book Order Form

The `DrawBookOrderForm` method is an example of an order entry form or an invoice. It is an example for data table support. It demonstrate the use of `PdfTable`, `PdfTableCell` and `PdfTableStyle` classes.

```
// Draw example of order form
private void DrawBookOrderForm()
{
    // Define constants to make the code readable
    const double Left = 4.35;
    const double Top = 4.65;
    const double Bottom = 1.1;
    const double Right = 7.4;
    const double FontSize = 9.0;
    const double MarginHor = 0.04;
    const double MarginVer = 0.04;
    const double FrameWidth = 0.015;
    const double GridWidth = 0.01;

    // column widths
    double ColWidthPrice = ArialNormal.TextWidth(FontSize, "9999.99") + 2.0 * MarginHor;
    double ColWidthQty = ArialNormal.TextWidth(FontSize, "Qty") + 2.0 * MarginHor;
    double ColWidthDesc = Right - Left - FrameWidth - 3 * GridWidth - 2 * ColWidthPrice -
ColWidthQty;

    // define table
    PdfTable Table = new PdfTable(Page, Contents, ArialNormal, FontSize);
    Table.TableArea = new PdfRectangle(Left, Bottom, Right, Top);
    Table.SetColumnWidth(new double[] { ColWidthDesc, ColWidthPrice, ColWidthQty, ColWidthPrice });

    // define borders
    Table.Borders.SetAllBorders(FrameWidth, GridWidth);

    // margin
    PdfRectangle Margin = new PdfRectangle(MarginHor, MarginVer);

    // default header style
    Table.DefaultHeaderStyle.Margin = Margin;
    Table.DefaultHeaderStyle.BackgroundColor = Color.FromArgb(255, 196, 255);
    Table.DefaultHeaderStyle.Alignment = ContentAlignment.MiddleRight;

    // private header style for description
    Table.Header[0].Style = Table.HeaderStyle;
    Table.Header[0].Style.Alignment = ContentAlignment.MiddleLeft;

    // table heading
    Table.Header[0].Value = "Description";
    Table.Header[1].Value = "Price";
    Table.Header[2].Value = "Qty";
    Table.Header[3].Value = "Total";

    // default style
    Table.DefaultCellStyle.Margin = Margin;

    // description column style
    Table.Cell[0].Style = Table.CellStyle;
    Table.Cell[0].Style.MultiLineText = true;

    // qty column style
    Table.Cell[2].Style = Table.CellStyle;
    Table.Cell[2].Style.Alignment = ContentAlignment.BottomRight;

    Table.DefaultCellStyle.Format = "#,##0.00";
    Table.DefaultCellStyle.Alignment = ContentAlignment.BottomRight;

    Contents.DrawText(ArialBold, FontSize, 0.5 * (Left + Right), Top + MarginVer +
Table.DefaultCellStyle.FontDescent,
    TextJustify.Center, DrawStyle.Normal, Color.Purple, "Example of PdfTable support");
```

```
    // reset order total
    double Total = 0;

    // loop for all items in the order
    // Order class is a atabase simulation for this example
    foreach(Order Book in Order.OrderList)
        {
        Table.Cell[0].Value = Book.Title + ". By: " + Book.Authors;
        Table.Cell[1].Value = Book.Price;
        Table.Cell[2].Value = Book.Qty;
        Table.Cell[3].Value = Book.Total;
        Table.DrawRow();

        // accumulate total
        Total += Book.Total;
        }
    Table.Close();

    // save graphics state
    Contents.SaveGraphicsState();

    // form line width 0.01"
    Contents.SetLineWidth(FrameWidth);
    Contents.SetLineCap(PdfLineCap.Square);

    // draw total before tax
    double[] ColumnPosition = Table.ColumnPosition;
    double TotalDesc = ColumnPosition[3] - MarginHor;
    double TotalValue = ColumnPosition[4] - MarginHor;
    double PosY = Table.RowTopPosition - 2.0 * MarginVer - Table.DefaultCellStyle.FontAscent;
    Contents.DrawText(ArialNormal, FontSize, TotalDesc, PosY, TextJustify.Right, "Total before
tax");
    Contents.DrawText(ArialNormal, FontSize, TotalValue, PosY, TextJustify.Right,
Total.ToString("#.00"));

    // draw tax (Ontario Canada HST)
    PosY -= Table.DefaultCellStyle.FontLineSpacing;
    Contents.DrawText(ArialNormal, FontSize, TotalDesc, PosY, TextJustify.Right, "Tax (13%)");
    double Tax = Math.Round(0.13 * Total, 2, MidpointRounding.AwayFromZero);
    Contents.DrawText(ArialNormal, FontSize, TotalValue, PosY, TextJustify.Right,
Tax.ToString("#.00"));

    // draw total line
    PosY -= Table.DefaultCellStyle.FontDescent + 0.5 * MarginVer;
    Contents.DrawLine(ColumnPosition[3], PosY, ColumnPosition[4], PosY);

    // draw final total
    PosY -= Table.DefaultCellStyle.FontAscent + 0.5 * MarginVer;
    Contents.DrawText(ArialNormal, FontSize, TotalDesc, PosY, TextJustify.Right, "Total payable");
    Total += Tax;
    Contents.DrawText(ArialNormal, FontSize, TotalValue, PosY, TextJustify.Right,
Total.ToString("#.00"));

    PosY -= Table.DefaultCellStyle.FontDescent + MarginVer;
    Contents.DrawLine(ColumnPosition[0], Table.RowTopPosition, ColumnPosition[0], PosY);
    Contents.DrawLine(ColumnPosition[0], PosY, ColumnPosition[4], PosY);
    Contents.DrawLine(ColumnPosition[4], Table.RowTopPosition, ColumnPosition[4], PosY);

    // restore graphics state
    Contents.RestoreGraphicsState();
    return;
}
```

4. Installation

Integrating `PdfFileWriter` to your application requires the following steps. Install the attached `PdfFileWriter.dll` file in your development area. Start the Visual C# program and open your application. Go to the Solution Explorer, right click on References and select Add Reference. Select the Browse tab and navigate your file system to the location of the `PdfFileWriter.dll`. When your application is published, the `PdfFileWriter.dll` must be included.

Source code documentation is available in as a help file `PdfFileWriter.chm`. The file is produced by Sandcastle. The result looks like Microsoft documentation pages.

If you want access to the source code of the `PdfFileWriter` project, install the `PdfFileWriter` project in your development area. The `PdfFileWriter.dll` will be in `PdfFileWriter\bin\Release` directory.

Add the following statement to all source modules using this library.

```
using PdfFileWriter;
```

If you intend to use charting, you need to add reference to: `System.Windows.Forms.Visualization`. In each source module using `Chart` you need to add

```
using System.Windows.Forms.DataVisualization.Charting;
```

5. References

- Adobe PDF file specification document available from Adobe website: “PDF Reference, Sixth Edition, Adobe Portable Document Format Version 1.7 November 2006”.
- Information about OpenType font specifications can be found at Microsoft Typography - OpenType Specification.
- Source for a decompression class matching the PDF Deflate compression class is available at "PDF File Analyzer With C# Parsing Classes" article.

6. History

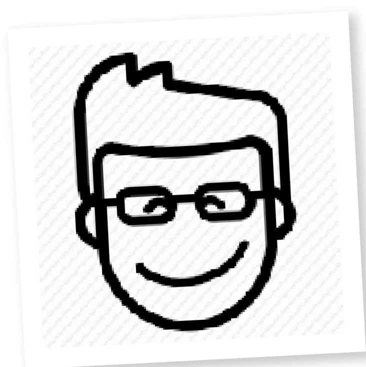
- 2013/04/01: Version 1.0 Original Version.
- 2013/04/09: Version 1.1 Support for countries with decimal separator other than period.
- 2013/07/21: Version 1.2 The original revision supported image resources with jpeg file format only. Version 1.2 support all image files acceptable to Bitmap class. See ImageFormat class. The program was tested with: Bmp, Gif, Icon, Jpeg, Png and Tiff. See Section 2.3 and Section 3.8 above.
- 2014/02/07: Version 1.3 Fix bug in PdfContents.DrawBezierNoP2(PointD P1, PointD P3).
- 2014/03/01: Version 1.4 Improved support for character substitution. Improved support for image inclusion. Some fixes related to PdfXObject.
- 2014/05/05: Version 1.5 Barcode support without use of fonts. Four barcodes are included: Code-128, Code-39, UPC-A and EAN-13. See Section 2.5 and Section 3.7.
- 2014/07/09: Version 1.6 (1) The CreateFile method resets the PdfDocument to initial condition after file creation. (2) The PdfFont object releases the unmanaged code resources properly.
- 2014/08/25: Version 1.7 Support for document encryption, web-links and QR Code.
- 2014/09/12: Version 1.8 Support for bookmarks.
- 2014/10/06: Version 1.9 Support for charting, PrintDocument and image Metafiles.
- 2014/10/12: Version 1.9.1 Fix to ChartExample. Parse numeric fields in regions with decimal separator other than period.
- 2014/12/02: Version 1.10.0 Support for data tables. Add source code documentation. Increase maximum number of images per document.
- 2015/01/12: Version 1.11.0 Support for video, sound and attachment files. Add support for Interleave 2 of 5 barcode.
- 2015/04/13: Version 1.12.0 Support for reordering pages and enhance data table border lines support.
- 2015/05/05: Version 1.13.0 PDF document output to a stream. PDF table insert page break. Image quality enhancement. Support for Standard-128 (RC4) encryption.

- 2015/06/08: Version 1.14.0 Support for long text blocks or TextBox within PDF Table.
- 2015/06/09: Version 1.14.1 one line change to Copy method of PdfTableStyle class.
- 2015/06/17: Version 1.15.0 Document information dictionary. PdfImage rewrite. Additional image saving options.
- 2015/06/18: Version 1.15.1 Remove unused source from solution explorer.
- 2015/07/27: Version 1.16.0 Unicode support. Commit page method.
- 2015/08/07: Version 1.16.1. Fix for small (<0.0001) real numbers conversion to string.
- 2015/09/01: Version 1.16.2. Fix for undefined characters. The selected font does not support characters used.
- 2015/09/22: Version 1.16.3. PdfTable constructor uses current page size to calculate the default table area rectangle. When PdfTable starts a new page the page type and orientation is taken from the previous page.
- 2015/09/30: Version 1.16.4 Consistent use of IDisposable interface to release unmanaged resources.
- 2016/01/26: Version 1.17.0 WPF graphics, transparency, color blending, elliptical arcs and quadratic Bezier curves.
- 2016/02/29: Version 1.17.1 PdfTable will display headers properly when the first column header is a TextBox.
- 2016/03/22: Version 1.17.2 PdfInfo PDF documents properties will be displayed properly .
- 2016/04/14: Version 1.17.3 Fix problem with non integer font size in regions that define decimal separator to be non period (comma).
- 2016/05/24: Version 1.18.0 Named destinations and creation of PdfFont resource.
- 2016/06/02: Version 1.18.1 Re-apply 1.17.3 fix.
- 2016/06/13: Version 1.19.0 Document links. Changes to named destination. Interactive features support to TextBox and PdfTable.
- 2016/07/27: Version 1.19.1 Fix: AddLocationMarker fix for regions with decimal separator not period.
- 2017/08/31: Version 1.19.2 Fix: Debug working directory is not saved as part of the project
- 2018/06/26: Version 1.19.3 Fix PdfFontFile.BuildLocaTable method. Long format buffer pointer initialization. Fix PdfTableCell add value type of DBNull.
- 2018/07/15: Version 1.20.0 Modify QR Code support by adding number of pixels per module.
- 2019/02/06: Version 1.21.0 Support for PDF417 barcode.
- 2019/02/13: Version 1.21.1 Fix for PDF417 barcode quiet zone.
- 2019/02/18: Version 1.22.0 Support for sticky notes.
- 2019/05/26: Version 1.23.0 Support for layers and changes to QRCode and Pdf417 barcode.
- 2019/06/06: Version 1.24.0 Support for layers control of images and annotations.
- 2019/06/20: Version 1.24.1 Support for meter as unit of measure.
- 2019/07/15: Version 1.25.0 Support for font collections (mainly CJK fonts) and for non ASCII font names.
- 2019/07/28: Version 1.26.0 Support for XMP Metadata and QR Code ECI Assignment Number.
- 2020/09/09: Version 1.27.0 Fix a out of memory problem related to PDF417 barcode. The broblem only occured under unusuual circumstances.

License

This article, along with any associated source code and files, is licensed under The Code Project Open License (CPOL)

About the Author



Uzi Granot

Canada

No Biography provided

Comments and Discussions

1500 messages have been posted for this article Visit **https://www.codeproject.com/Articles/570682/PDF-File-Writer-Csharp-Class-Library-Version-1-27** to post and view comments on this article, or click **here** to get a print view with messages.

Permalink
Advertise
Privacy
Cookies
Terms of Use

Article Copyright 2013 by Uzi Granot
Everything else Copyright © CodeProject, 1999-2020

Web03 2.8.2009011.1