Quick PDF Library 7

Developer Guide

Debenu

About

Quick PDF Library is a popular PDF SDK for manipulating PDF files on all levels. It supports a variety of different programming languages and is used by thousands of developers around the world.

Quick PDF Library is a powerful royalty-free PDF developer SDK used by thousands of developers for working with PDFs on all levels. Including a robust API with over 500 functions for use with C, C++, C#, Delphi, PHP, Visual Basic, VB.NET, ASP, PowerBASIC, Pascal and more, Quick PDF Library truly is the ultimate toolkit for project where you need to create, edit, secure, print, render, split, merge or manipulate PDF documents.

The library is available in ActiveX, DLL, Delphi LIB and TCP editions. Single, multiple developer and source code license are available.

Quick PDF Library is a Debenu (www.debenu.com) product and can be found on the web at www.quickpdflibrary.com.

Features

The Quick PDF Library API consists of approximately 600 functions which cover a wide range of features from the PDF specification. Some of the SDKs features include:

- Create PDFs on the fly
- Render and print PDFs
- Secure, sign and protect PDFs
- Create, fill and edit PDF forms
- Split, merge, append and combine PDFs
- Extract text and images from PDFs
- Edit PDFs initial view and document properties
- Add text, images and barcodes to PDFs
- Add and manipulate JavaScript, bookmarks and links
- Support for Annotations, vector graphics, GeoPDF
- ...and much more (check out the function reference for the full list)

Programming Languages

Quick PDF Library is available as an ActiveX, a DLL and a native Delphi library. There is also a LIB edition for C++ and a TCP edition. You can use Quick PDF Library with any programming language that supports these technologies.

Some well-known programming languages that support these technologies are:

- ASP NET
- C#
- C++ Builder
- C/C++
- Classic ASP
- Delphi
- Java
- Pascal
- PHP
- PowerBASIC
- PowerBuilder
- Python
- VBScript
- Visual Basic
- Visual Basic NET
- Visual C++
- And more...

License terms

The full end user license agreement for Quick PDF Library can be read online here, but to give you a rough idea of how you can and can't use Quick PDF Library, here's a few key points:

- **Per developer.** Licenses for Quick PDF Library are sold on a per developer basis. No run-time licenses, no server licenses and no annual payments.
- **Royalty free.** You can use Quick PDF Library in your applications without needing to pay any royalty fees for distribution.
- **No limits on number of applications.** We sell licenses based on the number of your developers who will be using the library, not the number of applications your company intends to build using our library.
- **Servers.** You are permitted to use Quick PDF Library in a server environment with the small provision that access to Quick PDF Library by third parties must be via your own software and your software cannot expose functionality of the library via an API.
- **Compiled applications only.** You are not permitted to create your own PDF software libraries or to use Quick PDF Library in any development component or toolkit.
- **No reselling.** You are not allowed to resell Quick PDF Library or your license key, but you can embed the software in your application, or distribute it with your system.

Setup

System requirements

Desktop

- Windows XP (32-bit)
- Windows Vista (32-bit)
- Windows 7 (32-bit)

Server

- Windows Server 2003 (32-bit)
- Windows Server 2008 (32-bit)

64-bit

- Quick PDF Library can be successfully run on 64-bit machines using WoW64 or the TCP edition of the library. More information here.
- Note: we are developing a 64-bit solution; however, the release date is not confirmed. Email us at support@quickpdflibrary.com if you're interested in testing a 64-bit solution.

Evaluation Version and Full Version

The evaluation version and the full version of Quick PDF Library use the same installer, so you only ever need to download one version of Quick PDF Library.

The full version is unlocked using your commercial license key.

The evaluation version is time limited to 30 days; however, during those 30 days the library is not limited in any way. You get full access to all of the functions without any nag ware or trial watermarks.

License Key

There are two different types of license key for Quick PDF Library: a trial license key and a commercial license key. The trial license key is provided with every download and installation of the library and the commercial license key is provided to you after you have purchased the library.

By default the trial license key is located in a text file called TRIAL_LICENSE_KEY.TXT which is located in the default installation directory for the library. This text file also includes the expiration date for the trial license key that you have been issued.

Installation

Quick PDF Library is delivered to customers by way of an electronic download. When installing Quick PDF Library the default installation location is the *Program Files* directory.

32-bit machine

C:\Program Files\Quick PDF Library

64-bit machine
C:\Program Files (x86)\Quick PDF Library

The installation directory will contain five different editions of the library: ActiveX, Delphi, DLL, LIB and TCP. Each edition has its own sub-directory which includes a getting started guide.

For customers who are evaluating Quick PDF Library the installation directory also contains a file called *TRIAL_LICENSE_KEY.TXT* which contains your 30 day trial license key.

The installation directory also contains Quick PDF Library Function Reference, the Developer Guide, a samples folder and the license agreement for this software.

Resources

Getting started guides

There is a special getting started guide for each edition of Quick PDF Library available online:

- Getting Started ActiveX Edition
- Getting Started DLL Edition
- Getting Started Delphi Edition
- Getting Started LIB Edition
- Getting Started TCP Edition

The getting started guides are also provided as PDFs in the product download.

Function reference

The function reference for Quick PDF Library is provided as a PDF in the product download but it is also available for viewing online: Quick PDF Library Function Reference.

Tutorials

Tutorials for Quick PDF Library in a variety of different programming languages can be found on Quick PDF Library's Tutorials page. These tutorials provide all of the information required to get up and running in a few different programming environments and also a step-by-step guide on how to create your first PDF application with Quick PDF Library.

Sample code

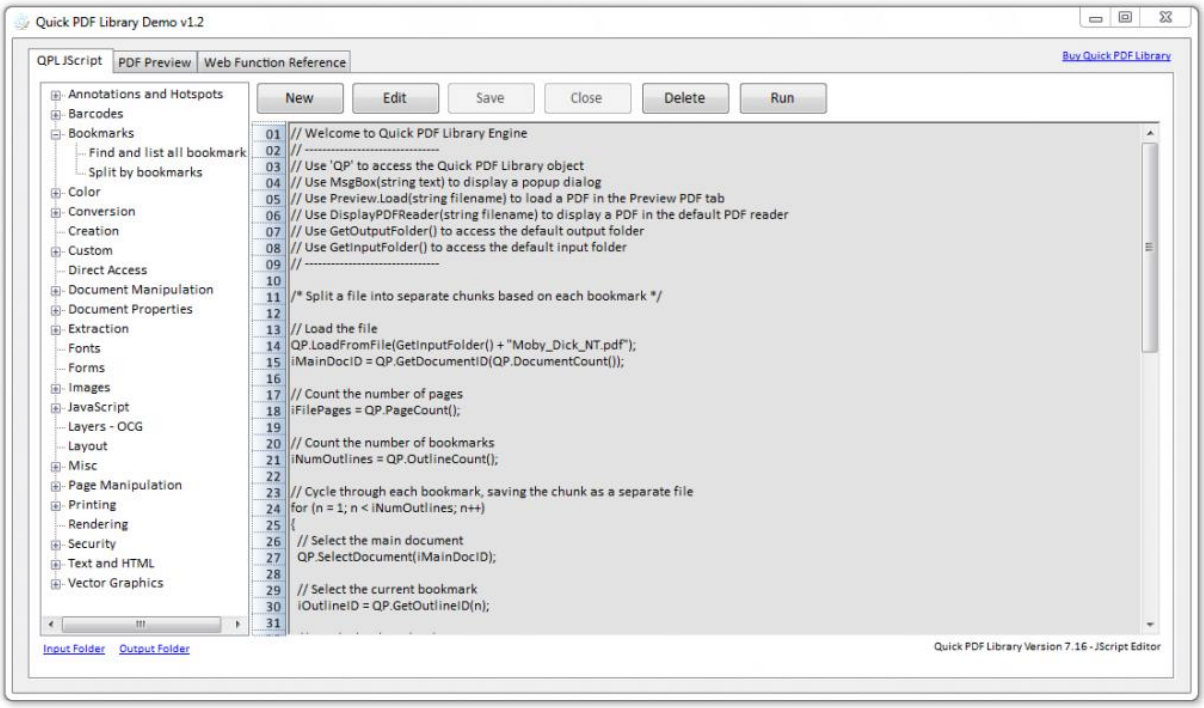
Sample code for Quick PDF Library is available in a few different locations.

- Official samples
- User samples on forum
- Knowledge base samples
- Tasks section of this Developer Guide

Demo

The Quick PDF Library Demo is a customizable demo application for Quick PDF Library.

A demo application that shows you how to perform a wide range of different tasks — such as split pages by bookmarks, convert PDFs to images, extract text and much more — through customizable scripts.



But to begin with, you don't have to customize anything; you can just select one of the scripts and click on the Run button to see the library in action. No effort required. Then later, if you want to take the library for a spin, you can either customize an existing script or create a new script of your own and save it within the application for future use.

This demo is for new and existing customers alike. The default scripts provided demonstrate how to perform some of the most common PDF related tasks and will be updated frequently in the future with more useful scripts.

Forum

Quick PDF Library has its very own user-to-user forum for Quick PDF Library at QuickPDF.org/Forum. This forum is maintained by fans of Quick PDF Library and has over 5 years of very useful content and discussions.

Knowledge base

The knowledge base tries to answer all of the basic (and some not so basic) questions that developers often task. Currently it has almost 400 articles and that list is growing every day as more and more customers ask questions.

API Overview

Quick PDF Library has almost 600 hundred functions that let you do about just about everything with PDF files. The API uses a flat structure, so to help you find the functions you need the functions are broken down into groups in our documentation.

Function groups

In our documentation functions are broken down into different groups -- although functions can belong to multiple groups -- to make it easier to keep track of them. It's important to know that these function groups don't necessarily reveal much about the structure of PDF since they're just useful groupings that we thought of.

The different function groups can be seen in the online function reference and as bookmarks in the PDF version of the function reference.

What you should know

To save you time we've compiled a list of things you might want to be familiar with prior to using Quick PDF Library. You don't necessarily need to know all of this information if you just want to do something similar, but if you're going to be making active use of a bunch of different functions in the library then this information could be useful to know.

Unlock the library

The UnlockKey function needs to be called and the return value checked otherwise most other functions called later will fail. More info here

Check functions for return values

Always check the return value of the important functions such as LoadFromFile because if this function fails the subsequent function calls will fail to. Every single function returns a value. Checking for return values is not a requirement, but it can be immensely useful in ensuring the robustness of your code and debugging.

If a function is not documented as having a return value then you can assume that a return value of one (1) indicates successful. Zero (0) or other values could indicate an error or it could be returning a valid handle or ID.

Memory and direct access functions

Function names that start with DA indicate that the function is a direct access function. Functions that do not have this pre-fix are memory functions. Direct access and memory functions cannot be used together. Combining them in your code will result in your code not working correctly. More info here.

The DA functions are primarily used for PDF documents that are very large and contain thousands of pages. They are generally faster for these larger documents because the document does not need to be loaded into memory. For file sizes under 500 MB or under a few thousand pages the speed differences are negligible.

Blank document automatically loaded

When you initialize the library there is always a one page blank document in memory. It is selected and ready to use by default. This is due to the design of the library. There must always be at least one document in memory, so if you try to delete that document using the DeletePages function, the library will automatically re-create a one page blank document.

The blank one page document uses a Letter page size which is 8.5 x 11 inches or 215.9 mm × 279.4 mm. The page size can be changed using the SetPageSize

New documents automatically selected

Whether you load an existing document using the LoadFromFile function or create a new document using the NewDocument function, it will automatically be selected in memory and the documents ID can be retrieved using the SelectedDocument function.

Multiple documents in memory permitted

You can have more than one document in memory and can swap between them using the ID returned from calls such functions as NewDocument and SelectedDocument. You can also count all documents in memory using the DocumentCount function and then retrieve each documents ID or filename using GetDocumentID or GetDocumentFileName. All of the document management related functions can be seen in the document management section in the function reference.

Origin point for drawing operations

The origin has coordinates of “0,0” and is the starting point for finding all other points. The origin point for a page in a PDF typically starts at a page corner. The default origin for Quick PDF Library is the bottom left page corner.

Using the SetOrigin function in Quick PDF Library you can change the point of origin to be any page corner (bottom left, top left, top right, bottom right).

By default calling DrawText(10, 10, “Test”) will result in the text being drawn at 10 points in from the left of the page and 10 points up from the bottom of the page, but if you call SetOrigin(1) prior to DrawText then the text will be drawn at 10 points in from the left of the page and 10 points down from the top of the page because passing the value 1 to the Origin parameter for the SetOrigin function changes the origin to top left of the page.

The default point of origin in Adobe Acrobat was the bottom left page corner up until Acrobat 8, at which point Adobe switched the point of origin to the top left page corner. As mentioned above, you can set Quick PDF Library to use any page corner in a PDF.

Measurement units

In PDF the coordinate system is called default user space. The default for the size of the unit in default user space (1/72 inch) is approximately the same as a point, a unit widely used in the printing industry. It is not exactly the same, however; there is no universal definition of a point.

Using the SetMeasurementUnits function you can change the units for all measurements given to and returned from the library. The available options are default user space, millimeters and inches.

Unicode, UTF-8 and the DLL and Delphi Editions

There are many different ways to encode Unicode characters. One way is to use strings with 16-bit characters. COM/ActiveX uses 16-bit characters, so adding Unicode support for the ActiveX edition of the library was easy.

For the Delphi and DLL editions, the strings have always been 8-bit characters. Unfortunately we can't change the definition of functions as this would cause issues with backwards compatibility.

This means that when using the Delphi and DLL editions and working with Unicode characters, you need to encode your file names with UTF8 encoding, as mentioned in the function reference. Make sure that you pay attention to each function description as it will specifically mention if you need to encode or decode the input or output.

Different languages will have different functions to do the UTF8 encoding.

Fonts

There are three different ways that fonts can be referenced or stored in PDF files. They are:

Full Font Embedding = Larger file size
Recipient doesn't need the same font to view or edit the file

Subset Font Embedding = Smaller file size
Recipient doesn't need the same font to view but does need the same font installed in order to edit the file

No Font Embedding = Smallest file size
Recipient needs to have same fonts installed

Each option has its merits. As long as option 1 or 2 above when you are building your PDF files, then you can be sure that when your PDF is rendered or printed the font you've specified will be used. If you use option 3, then the PDF viewer will attempt to locate the specified font on the local machine but if it cannot be found then it will use a substitute font during the viewing/printing process.

When a PDF is displayed on your screen it is rendered in exactly the same fashion as it would be prior to being printed.

Processing digitally signed PDF files

Any changes that you make to a PDF after it has been digitally signed will invalidate the digital signature. PDF files that contain digital signatures must be incrementally updated -- meaning that the contents of the PDF are updated without rewriting the entire file by way of appending changes to the end of the file. If digitally signed PDF files are not updated this way then the digital signature will not only be invalidated, but it will be completely broken too.

When using Quick PDF Library it's important to note that the combination of LoadFromFile and SaveToFile does not incrementally update the file. Instead it completely rewrites the contents of the PDF and breaks the digital signature. If you're working with a PDF that has a digital signature and you do not want to break or remove the digital signature, then you should not use the SaveToFile function.

If you need to modify a PDF that has been digitally signed then you should use the DAOpenFile and DAAppendFile functions as the DAAppend function does incrementally update PDF files, which means that the digital signature will be invalidated, but not broken.

Optional Content Groups and Layers: What is the difference?

In Quick PDF Library layers and optional content groups do not refer to the same technology, however, in Adobe Acrobat, they do. Confused? There's a simple explanation for this anomaly:

A page can have one or more content streams. These content streams are combined into one long string by the PDF viewer.

When Quick PDF Library was first developed (about 10 years ago) we called these individual content streams "layers" as they enabled you to move groups of things on top of and below other things and each new function in this area included the term layer in its name (NewLayer, LayerCount, MoveLayer, etc.).

However, in version 1.5 of the PDF specification (after we'd already begun using the term layer to describe content streams for a few years) Adobe added a new feature called optional content groups. This new feature was technically referred to as optional content groups (OCGs) in the PDF specification, but in Adobe Acrobat and Adobe Reader, Adobe elected to use the term layers instead. And thus the confusion began:

We use the term "layers" for content streams and Adobe uses the term "layers" for optional content groups.

So what do we call optional content groups in Quick PDF Library? Well, we call them optional content groups (NewOptionalContentGroup, OptionalContentGroupCount, etc.). We didn't think

it was necessary to change our terminology, since we've taken our terminology straight from the PDF specification.

In addition, if we were to rename our functions in order to accommodate Adobe's slightly different terminology, then it would break the backwards compatibility of our library for our customers, and so renaming the functions for clarity is unfortunately not an option at this stage.

Need Appearances and Form Fields

A form field in a PDF is an interactive element that floats above the actual content of the PDF. In order for a form field to be displayed it must have an associated appearance stream that tells the conforming PDF viewer how to render the form fields.

Appearance streams can be generated in advance, using the UpdateAppearanceStream function, or you can tell the conforming PDF viewer to automatically generate the necessary appearance streams when the PDF is opened by using the SetNeedAppearances function to set the NeedAppearances flag to true.

If the NeedAppearances flag is set to false and no appearance stream has been generated in advance for the form fields, then the form fields will not be shown in the conforming PDF viewer.

After existing form fields have been updated the appearance stream must be updated, so if NeedAppearances flag is set to false then you will need to make a new call to the UpdateAppearanceStream function.

Tasks

These are the most common tasks for using Quick PDF Library, but it is not an exhaustive list, our PDF SDK is capable of far more than is covered here. The sample code show here is written in JScript which is very similar to JavaScript (and also C#) in syntax.

PDF Creation

PDF creation deals with creating PDF files programmatically from scratch.

Create a simple PDF

Quick PDF Library allows you to create new PDF files with very few lines of code. If you wanted to create a blank document with no content then all you would need to do is make one simple call to the SaveToFile function and that would save the blank document that is already in memory to disk. Making a simple PDF with just one line of text is not much harder.

```
/* Create a simple PDF */

// Set page origin to top left, default
// is the bottom left

QP.SetOrigin(1);

// Draw some text on the PDF

QP.DrawText(100, 100, "Hello World");

// Save the new PDF

QP.SaveToFile("my_new_doc.pdf");
```

That's it. This is your basic building block and you can make your PDF files as simple or as complex as you need them to be.

Create a complex PDF

Quick PDF Library lets you add text, images, vector graphics, form fields, annotations and much more to PDF files. You won't necessarily need all of these elements in your PDF, but here's an example of how some of them fit together.

```
/* Create a complex PDF */

// Set page origin to top left, default is the bottom left

QP.SetOrigin(1);

// Invoice number text

QP.SetTextSize(20);
QP.DrawText(55, 65, "Invoice #");
QP.SetTextColor(0, 0.5, 1);
QP.DrawText(140, 65, "1564824");

// Company logo

QP.SetFillColor(0, 0, 0);
QP.SetLineColor(1, 1, 1);
QP.DrawCircle(475, 15, 5, 2);
QP.DrawCircle(490, 20, 10, 2);
QP.DrawCircle(505, 25, 15, 2);
QP.DrawCircle(520, 30, 20, 2);
QP.DrawCircle(535, 35, 25, 2);
QP.DrawCircle(550, 40, 30, 2);

QP.DrawHTMLTextBox(470, 38, 100, 100, "THE<br>CIRCLE<br>COMPANY.");
```

```
// Draw some lines to give the invoice structure

QP.SetLineColor(0, 0, 0);
QP.DrawLine(20, 150, 592, 150);
QP.DrawLine(20, 200, 592, 200);
QP.DrawLine(400, 250, 592, 250);

// Text color for body

QP.SetTextColor(0, 0, 0);

// Invoice headings

QP.SetTextSize(12);
QP.DrawText(50, 145, "Description");
QP.DrawText(225, 145, "Quantity");
QP.DrawText(375, 145, "Unit Price");
QP.DrawText(525, 145, "Amount");

// Invoice content

QP.SetTextSize(10);
QP.DrawText(50, 195, "Fictional Product 1.0");
QP.DrawText(245, 195, "5");
QP.DrawText(385, 195, "199.00");
QP.DrawText(532, 195, "995.00");

// Invoice totals

QP.DrawText(455, 225, "Sub-Total");
QP.DrawText(532, 225, "$995.00");
QP.DrawText(455, 245, "Tax");
QP.DrawText(532, 245, "$0.00");
QP.DrawHTMLText(455, 270, 100, "<b>Total</b>");
QP.DrawHTMLText(532, 270, 100, "<b>$995.00</b>");

// Payment due date

QP.DrawHTMLText(50, 400, 100, "<b>Due Date: Today</b>");

// Save the invoice

QP.SaveToFile("invoice.pdf");
```

Convert this code into the language of your choice and then run it. With not too much difficulty at all you can create a nifty invoice.

PDF Conversion

Quick PDF Library supports converting images to PDF, PDF to images and PDF to text.

Convert an image to PDF

Converting images to PDF files is quite a simple task.

```
/* Image to PDF conversion */

// Load your image into memory

QP.AddImageFromFile("sample123.jpg", 0);

// Get width, height of the image

lWidth = QP.ImageWidth();
lHeight = QP.ImageHeight();

// Reformat the size of the page in the selected document

QP.SetPageDimensions(lWidth, lHeight);

// Draw the image onto the page using the specified width/height

QP.DrawImage(0, lHeight, lWidth, lHeight);

// Store the updated PDF where you like

QP.SaveToFile("from image.pdf");
```

Convert a PDF to an image

When a PDF is displayed it is actually rendered as an image.

```
/* PDF to image conversion */

// Load the sample file

QP.LoadFromFile("sample.pdf");

// Calculate the number of pages

iNumPages = QP.PageCount();

// Render each page of the document to a separate file.
// To view the images open the output folder.

QP.RenderDocumentToFile(72, 1, iNumPages, 0, "image.bmp");
```

Convert a PDF to text

Extracting text from a PDF can at times be a difficult task. If a PDF has been scanned from paper and no OCR process has been performed then the page in a PDF might display what looks like text but in the objects in the PDF no text actually exists, just an image. You can use the HasFontResources function to check if a PDF has any text. Extracting text from a PDF that does indeed contain text objects is a relatively straight forward process using one of the text extraction functions.

```
/* PDF to text conversion */
```

```
// Load the test file and Iterate through each page in the
// PDF and append it to a text file.

strInputFilePath = "apply_ fingerprint.pdf";

QP.LoadFromFile(strInputFilePath);

// Calculate the number of pages

iNumPages = QP.PageCount();

strText = "";
nPage = 0;

// Loop through each page in the PDF and extract the text from the page

for(nPage = 1; nPage<=iNumPages; nPage++)
{
  strText = strText + QP.ExtractFilePageText(strInputFilePath, "", nPage, 0);
}

// Write all the data to a file

s = oFSO.CreateTextFile("extracted_text.txt", 1);
s.Writeline(strText);
s.Close();
```

PDF Rendering

Rendering a PDF so that it can be viewed on screen is quite a difficult process. To render a PDF page the following steps need to be taken:

1. Locate the page object in the PDF.
2. Extract the page's content streams (one or more strings).
3. Parse the content stream to get the page description commands - these are similar to PostScript language commands.
4. The content streams commands refer to one or more resources (fonts, images, etc.) - these resource objects are located in the PDF and images are decoded and fonts are processed to obtain the glyph outlines. Text is simply a collection of shapes just like other graphic content.
5. As the page content is followed there are operations that adjust the user matrix which change the position, scaling and rotation.
6. These commands / objects collectively describe the page in mathematical terms and are then converted into an appropriate output - either an image (using GDI+ in our case) or a different graphical language like EPS, EMF/WMF etc.
7. The image is then loaded into the PDF viewer.

Luckily the only item from above that you have to worry about is loading the image into your application. Quick PDF Library provides a variety of different functions for rendering a page from a PDF as an image, some of which can be seen when viewing the PDF rendering tag in our knowledge base. Here is a simple example using the RenderDocumentToFile function.

```
/* Render a PDF file */

// Load the sample file

QP.LoadFromFile("sample.pdf");

// Calculate the number of pages
iNumPages = QP.PageCount();

// Render each page of the document to a separate file.
// To view the images open the output folder.

QP.RenderDocumentToFile(72, 1, iNumPages, 0, "image.bmp");
```

PDF Editing

There are many ways in which a PDF file can be edited (not all of which Quick PDF Library supports):

- Text manipulations: edit text in a PDF.
- Image editing: edit an image in a PDF file or replace it.
- Page related changes: delete pages, change their order or rotate them.
- Altering a PDF file: merge pages from separate PDF files, edit password protected files, edit metadata, add new content, etc.
- Color editing: adjust the color of text objects, shapes, images, etc.

It is extremely important to realize that PDF is not a very good “authoring” format and was never designed to be. If you need to make significant changes to a document -- i.e. re-write paragraphs of text, change the layout, etc -- then it is best to go back to the original source document (often Microsoft Word or similar authoring tools) and make the changes there and then create the PDF again.

PDF is a fixed format and thus best suited to being used as a presentation format. Always keep in mind that PDF files are an accurate representation of a document. They are meant for output or on-screen viewing.

So what PDF editing does Quick PDF Library support?

- **Text manipulations.** Quick PDF Library does not let you edit existing text in a PDF but it does let you add new text.
- **Image editing**. Quick PDF Library does not let you edit an image -- i.e. you cannot resize the image or convert an RGB image to a CMYK image -- but it does let you delete images and replace images with a different image.
- **Page related changes**. Quick PDF Library provides an extensive range of functions for manipulating pages in such ways as rotating, replacing, extracting, deleting, moving and cropping pages. Check out the page manipulation section in the function reference for all of the relevant functions.
- **Altering a PDF file**. Quick PDF Library provides a variety of different ways of altering PDF files. From merging PDF files together or extracting pages from a PDF to encrypting PDF files or editing a PDFs metadata and much more.
- **Color editing**. Quick PDF Library provides a lot of control over color when adding new text or vector graphics into a PDF but it does not let you editing existing colors in a PDF.

PDF Printing

The same process that goes into preparing a PDF for viewing is required for preparing a PDF for printing. The PDF must be rendered an image (PostScript and similar formats are also permitted by certain printers) prior to being sent to the printer or displayed on the screen.

All of the printing functionality in Quick PDF Library enables you to create a printing interface with a GUI or to create a printing process that silently prints documents in the background without any user interaction.

Standard PDF printing

The standard printing option in Quick PDF Library provides a quick and simple way to print a document without having to worry about too many settings.

```
/* Simple PDF printing */

// Load a local sample file from the input folder

QP.LoadFromFile("JavaScript.pdf");

// Configure print options

iPrintOptions = QP.PrintOptions(0, 0, "Printing Sample")

// Print the current document to the default
// printing using the options as configured above

QP.PrintDocument(QP.GetDefaultPrinterName(), 1, 1, iPrintOptions);
```

Custom PDF printing

The custom printing options give you far greater control over the printing process. These options give you as much control over printing as you usually see in the print dialog of most applications.

```
/* Advanced PDF printing */


// Load a sample file from the input folder

QP.LoadFromFile("JavaScript.pdf");

// Create the custom printer

CustomPrinter = QP.NewCustomPrinter("Microsoft XPS Document Writer");

// Setup the settings for the customer printer
// Medium quality

QP.SetupCustomPrinter(CustomPrinter, 5, 2);

// Monochrome

QP.SetupCustomPrinter(CustomPrinter, 6, 1);

// Horizontal Duplex

QP.SetupCustomPrinter(CustomPrinter, 7, 3);

// Configure print options

iPrintOptions = QP.PrintOptions(0, 0, "Printing Sample");

// Print the current document to the default printing
// using the options as configured above

QP.PrintDocument(CustomPrinter, 1, 1, iPrintOptions);
```

The custom printer functions include all of the functionality required to create your own print dialog.

PDF Security

PDF provides a variety of different security options. From a simple open password -- where the user must type in a password prior to being able to view the PDF -- to the more complicated options where a master password can be used to apply document restrictions, such as no printing, no editing, etc, to the document. It is also possible to apply digital signatures to PDF files.

There are two different types of passwords used in PDF files but also two different terms for each of these types: an owner or master password and an open or user password. The owner or master password is required for setting document restrictions and the open or user password is required for making the user enter in a password prior to viewing the document.

Open password

If you want to require that the user type in a password prior to being able to view the PDF file then you need to encrypt the PDF with an open password.

```
/* Apply an open password to a PDF */

// Load a sample file from the input folder

QP.LoadFromFile("secure_me.pdf");

// For adding document restrictions we'll use the
// EncodePermissions function. For this example we will
// not restrict any actions.

EncPerm = QP.EncodePermissions(1, 1, 1, 1, 1, 1, 1, 1);

// Encrypting the document must be the last
// function called prior to saving

QP.Encrypt("locked_down", "", 1, EncPerm);

// Save the updated file to the output folder

QP.SaveToFile("secured.pdf");
```

Document restrictions

If you want to prevent the user from performing certain actions with your document then you need to add document restrictions to your PDF along with an owner password.

```
/* Apply a master document and document restrictions to a PDF */

// Load a sample file from the input folder

QP.LoadFromFile("secure_me.pdf");

// For adding document restrictions we'll use the
// EncodePermissions function. Look at this function
// in the reference to see what the available options are.

// In this sample the only permission we'll give the user
// to the ability to print the document. The user won't be
// able to copy text, modify the document, etc.

EncPerm = QP.EncodePermissions(1, 0, 0, 0, 0, 0, 0, 0);
```

```
// Encrypting the document must be the last
// function called prior to saving

QP.Encrypt("locked_down", "", 1, EncPerm);

// Save the updated file to the output folder

QP.SaveToFile("secured.pdf");
```

Digital signatures

Quick PDF Library allows you to add digital signatures to PDF files at the document level. It does not allow you to sign digital signature form fields.

```
/* Apply a digital signature to a PDF */

// Use the SignFile function to apply the digital signature.
// This function handles loading the file, signing it, and then
// saving it to disk.

QP.SignFile("sign_me.pdf", "", "Digital Signature Test", "signed.pdf", "qpl_test.pfx",
"testing", "A good reason", "Here", "Quick PDF Library");
```

PDF decryption

PDF files can be decrypted. Depending on if a user or master password has been used you will need to include this while decrypting the PDF.

```
/* Decrypt a PDF file */

// Load a file from the samples folder

QP.LoadFromFile("secured.pdf");

// Before we call the Decrypt function, we need
// to set the password.

QP.SetPassword("locked_down");

// Call the Decrypt function

QP.Decrypt();

// Save the decrypted file to the output folder

QP.SaveToFile("decrypted.pdf");
```

PDF Splitting

Splitting a PDF is the process of taking pages from one PDF to create a new PDF which contains only those pages. Splitting is a term that is used in the PDF industry but what is really happening behind the scenes is that all of the objects, dictionaries and other information that make up a certain page in your source document is copied and used to create a new document.

Split PDF documents

Quick PDF Library lets you split PDF files into individual PDF files. The splitting functionality is versatile so you can split each page in a PDF into a new document or you can split a range of pages into a new document or you can split based on a number of different requirements.

Split each page of a PDF file into a new document

The ExtractFilePages function can be used to split each page of a PDF file into a new document. This function lets you extract ranges of pages from a PDF document on disk and places the extracted pages into a new PDF document. If you want to process files in memory, instead of directly on disk, then you should use the ExtractFileRanges function.

```
/* Split PDF by page */

// Load the sample file that we will split

QP.LoadFromFile("sample.pdf");

// Use the SelectedDocument function to get the
// document ID of the file that we just loaded.

int DocID = QP.SelectedDocument();

// Count the total number of pages in the
// selected document. We need the total
// number of pages before we can use the
// ExtractFilePages function.

int TotalPages = QP.PageCount();

// Remove the selected document from memory.

QP.RemoveDocument(DocID);

// Loop through each page in the document
// and use the ExtractFilePages function
// to copy and save each page in the
// document to a new document.

for (int n = 1; n < TotalPages; n++)
{
QP.ExtractFilePages("sample.pdf", "sample_split_" + Convert.ToString(n) + ".pdf",
Convert.ToString(n));
}

// That's it. You can also copy page ranges using
// this function, for example "10,15,18-20,25-35".
```

Split PDF document by page range

Quick PDF Library's ExtractFilePages and ExtractPageRanges functions let you extract a range of pages from a PDF.

A sample page range is "10,15,18-20,25-35". Commas separate individual pages while dashes indicate consecutive pages to extract. Invalid characters will be ignored. Reversed page ranges such as "5-1" will be accepted. Duplicate page numbers will be accepted but if a change is made to such a page the same changes will appear on the duplicate pages. The list of pages will not be sorted so the resulting document will have the pages in the specified order.

Extract a range of pages on disk using ExtractPageRanges

Use the ExtractFilePages function to extract a range of pages from a PDF on disk.

```
/* Extract a range of pages using direct access */

// The ExtractFilePages function makes extracting
// a range of pages very easy. Simply specify the file
// that pages should be extracted from and the file
// that the pages should be saved to -- as well as the
// range-list that you would like to extract.

QP.ExtractFilePages("example.pdf", "example_page_range.pdf", "2-5, 8, 10, 13-19");
```

Extract a range of pages in memory using ExtractPageRange

Use the ExtractPageRanges function to extract a range of pages from a PDF in memory.

```
/* Extract a range of pages in memory */

// Load the PDF

QP.LoadFromFile("example.pdf");

// Extract a range of pages

QP.ExtractPageRanges("2-5, 8, 10, 13-19");

// New doc with extract pages auto-selected

QP.SaveToFile("example_page_range.pdf");
```

The ExtractPages function can also be used to extract only consecutive pages.

PDF Merging

Two or more PDF files can be merged together. Sometimes this process is known is combining PDF files, joining PDF files or appending one PDF file to another.

Merge PDF documents

Quick PDF Library supports the merging of two or more PDF documents into one PDF document. The primary functions available for this task are MergeDocument, MergeFileList and MergeFileListFast. The function that you use for merging will depend on what your exact requirements are.

Merge two PDF documents together

If you want to merge two PDF files together then the MergeDocument function can be used. This function allows you to join one PDF document to another PDF document.

```
/* Combine two PDF files into one PDF */

// Setup the files for merging

string firstDoc = "sample1.pdf";
string secondDoc = "sample2.pdf";
string DestFileName = "merged_samples.pdf";

// Load the first PDF into memory

QP.LoadFromFile(firstDoc);
int MainDocID = QP.SelectedDocument();

// Load the second PDF into memory

QP.LoadFromFile(secondDoc);
int AppendID = QP.SelectedDocument();

// Select the PDF to which the other PDF
// should be appended to and then merge.
// After merging AppendID will be removed
// from memory and MainDocID will be selected

QP.SelectDocument(MainDocID);
QP.MergeDocument(AppendID);

// Remove document from memory

QP.SaveToFile(DestFileName);
```

During the merging process any form fields and annotations from the second document are preserved but outlines (bookmarks) are not.

Merge a list of PDF files together

If you want to merge a list (two or more) of PDF files together then the MergeFileList function is what you need. This function Merges all the files in a named file list and saves the resulting merged document to the specified file.

```
/* Merge a list of PDF files */

// Add a range of files to a list.

QP.AddToFileList("FilesToMerge", "sample1.pdf");
QP.AddToFileList("FilesToMerge", "sample2.pdf");
QP.AddToFileList("FilesToMerge", "sample3.pdf");

// Merge the list of files together

QP.MergeFileList("FilesToMerge", "merged_file_list.pdf");
```

Outlines (bookmarks), form fields and annotations from all the documents will be present in the merged document. The file list can be cleared using the ClearFileList function. Depending on the content and size of the documents that you are merging, the MergeFileListFast function can be used to obtain faster merging speeds.

Technical note

Read the below information only if you want to know more about PDF. This knowledge is not required for using the functions mentioned above. It is just for the curious.

It is important to note that merging two PDF files together is not the same as stapling two pieces of paper together. The only time you see a “page” in a PDF is when the PDF is rendered.

Internally each page in a PDF is represented by a page object. The page object is a dictionary which includes references to the page's content and other attributes. The individual page objects are tied together in a structure called the page tree. However, the structure of the page tree is not necessarily related to the logical structure or flow of the document.

When you merge two PDF files together you need to take two separate PDFs and merge all of the objects and resources together. It is not simply a case of physically appending one document to another.

So instead of stapling two sheets together, a more accurate comparison for the PDF merging process would be the recycling of paper where multiple different sheets of paper are broken down (and fibers are mixed) to create a new ream of paper.

Note: there are a few different terms which can be used when describing the task of merging PDF files together. For example, you could say combining PDF files together or appending one PDF file to another PDF file or joining PDF files together . All of these terms are essentially the same.

PDF Page Extraction

Quick PDF Library lets you extract individual pages or page ranges from a PDF, and create new documents with the pages extracted. The ExtractFilePages, ExtractPages and ExtractPageRanges functions will do the job for you (but you only need one of them).

Extract one page

The ExtractPages function can be used to easily extract one or more consecutive pages from a document (the other page extraction functions can technically be used for this as well).

```
/* Extract one page from a PDF */

// Load the PDF

QP.LoadFromFile("example.pdf");

// Extract one page. Specify start page and number of subsequent pages to extract.

QP.ExtractPages(1, 1);

// New doc with extract pages auto-selected

QP.SaveToFile("example_page1.pdf");
```

Extract a page range

The ExtractFilePages and ExtractPageRanges functions can be used to extract page ranges.

A sample page range is "10,15,18-20,25-35". Commas separate individual pages while dashes indicate consecutive pages to extract. Invalid characters will be ignored. Reversed page ranges such as "5-1" will be accepted. Duplicate page numbers will be accepted but if a change is made to such a page the same changes will appear on the duplicate pages. The list of pages will not be sorted so the resulting document will have the pages in the specified order.

Extract a range of pages on disk using ExtractFilePages

Use the ExtractFilePages function to extract a range of pages from a PDF on disk.

```
/* Extract a range of pages from a PDF on disk */

// The ExtractFilePages function makes extracting
// a range of pages very easy. Simply specify the file
// that pages should be extracted from and the file
// that the pages should be saved to -- as well as the
// range-list that you would like to extract.

QP.ExtractFilePages("example.pdf", "example_page_range.pdf", "2-5, 8, 10, 13-19");
```

Extract a range of pages in memory using ExtractPageRanges

Use the ExtractPageRanges function to extract a range of pages from a PDF in memory.

```
/* Extract a range of pages from a PDF in memory */
```

```
// Load the PDF

QP.LoadFromFile(“example.pdf”);

// Extract a range of pages

QP.ExtractPageRanges("2-5, 8, 10, 13-19");

// New doc with extract pages auto-selected

QP.SaveToFile(“example_page_range.pdf”);
```

The ExtractPages function can also be used to extract only consecutive pages.

PDF Forms

An interactive form -- sometimes referred to as an AcroForm -- is a collection of fields for gathering information interactively from the user. A PDF document may contain any number of fields appearing on any combination of pages, all of which make up a single, global interactive form spanning the entire document.

Quick PDF Library provides fully support for working with form fields in PDF files, from adding new form fields to retrieving data from form fields to updating existing form fields with new data.

Form Field Types

PDF forms support the following field types:

- *Button fields* represent interactive controls on the screen that the user can manipulate with the mouse. They include *pushbuttons*, *check boxes*, and *radio buttons*.
- *Text fields* are boxes or spaces in which the user can enter text from the keyboard.
- *Choice fields* contain several text items, at most one of which may be selected as the field value. They include *scrollable list boxes* and *combo boxes*.
- *Signature fields* represent digital signatures and optional data for authenticating the name of the signer and the document's contents.

The GetFormFieldType function can be used to determine which form field type your fields are.

Create a new simple PDF form

Quick PDF Library lets you create new PDF forms from scratch. In this example we'll create four new text form fields with varying characteristics.

```
/* Create a PDF form */

// Tell the library that all co-ordinates should
// begin from the top left corner of the page.

QP.SetOrigin(1);

// Add the first new form field

var iDf1 = QP.NewFormField("First Name", 1);
QP.SetFormFieldValue(iDf1, "Jane");
QP.SetNeedAppearances(1);
QP.SetFormFieldBounds(iDf1, 20, 20, 100, 20);
QP.SetFormFieldAlignment(iDf1, 2);

// Add the second new form field

var iDf2 = QP.NewFormField("Second Name", 1);
QP.SetFormFieldValue(iDf2, "Doe");
QP.SetNeedAppearances(1);
QP.SetFormFieldBounds(iDf2, 20, 50, 100, 20);

fID = QP.AddTrueTypeFont("Myriad Pro Cond", 1);
ffID = QP.AddFormFont(fID);
```

```
QP.SetFormFieldFont(iDf2, ffID);

// Add the third new form field

var iDf3 = QP.NewFormField("Age", 1);
QP.SetFormFieldValue(iDf3, "31");
QP.SetNeedAppearances(1);
QP.SetFormFieldBounds(iDf3, 20, 80, 100, 20);

// Add the fourth new form field

var iDf3 = QP.NewFormField("Nationality", 1);
QP.SetFormFieldValue(iDf3, "Australian");
QP.SetNeedAppearances(1);
QP.SetFormFieldBounds(iDf3, 20, 110, 100, 20);

// Save the new PDF form to the hard disk

QP.SaveToFile("new_pdf_form.pdf");
```

Fill PDF form

Filling in a PDF form can be done with just a few function calls. The data that you use to fill in the form can come from anywhere -- a database, captured user input, a hardcoded variable, etc. The important functions here are FormFieldCount, GetFormFieldType and SetFormFieldValue.

```
/* Fill a PDF form */

// Load a PDF form from the samples folder

QP.LoadFromFile("example_form.pdf");

// Count the number of form fields in the
// loaded document

FieldCountAcroForms = QP.FormFieldCount();

// Count the number of pages in the document

TotalPages = QP.PageCount();

// Loop through each page

for(p = 1; p  <= TotalPages; p++)
{
  // Select page number

  QP.SelectPage(p);

  // Loop through each form field on the selected page

  for(i = 1; i <= FieldCountAcroForms; i++)
  {
    // Determine form field type

    if (QP.GetFormFieldType(i) == 1)
    {
       // If form field type is text then add dummy text

        QP.SetFormFieldValue(i, "Dummy Text");
    }
```

```
  }
}

// Save the updated form

QP.SaveToFile("example_form_updated.pdf");
```

Delete all form fields from a PDF

Instead of adding or updating form fields you may wish to delete all of them. This is possible using the DeleteFormField function.

```
/* Delete PDF form fields */

// Load a PDF that contains some form fields

QP.LoadFromFile("pdf_form.pdf");

// Count the total number of form fields in the file

TotalFormFields = QP.FormFieldCount();

// Loop through each of the form fields and
// delete them using the DeleteFormField function

while (TotalFormFields > 0)
{
  QP.DeleteFormField(TotalFormFields);
  TotalFormfields = TotalFormFields--;
}

// Save the updated file to the output folder

QP.SaveToFile("no_pdf_form_fields.pdf");
```

Flatten form fields in a PDF

Sometimes you want to take the data in form fields and add it to the PDF as real text objects that are still displayed even when the PDF is printed or rendered in applications which do not support form fields. This process is called flattening. Basically you're flattening the interactive form, so that it is no longer interactive and all that is left is the text which was in the form fields. The main function for doing this is the FlattenFormField function.

```
/* Flatten PDF form fields */

// Load a PDF that contains some form fields

QP.LoadFromFile("pdf_form.pdf");

// Count the total number of form fields in the file

TotalFormFields = QP.FormFieldCount();

// Loop through each of the form fields and
// delete them using the DeleteFormField function

while (TotalFormFields > 0)
{
  QP.FlattenFormField(TotalFormFields);
```

```
  TotalFormfields = TotalFormFields--;
}

// Save the updated file to the output folder

QP.SaveToFile("flattened_pdf_form.pdf");
```

Get data from form fields

Quick PDF Library allows you to extract data from form fields using the GetFormFieldValue function.

```
/* Get data from PDF form fields */

// Load a PDF that contains some form fields

QP.LoadFromFile("pdf_form.pdf");

// Count the total number of form fields in the file

TotalFormFields = QP.FormFieldCount();

// Loop through each of the form fields and
// get value and display it in message box

while (TotalFormFields > 0)
{
  fieldValue = QP.GetFormFieldValue(TotalFormFields);
  MsgBox(fieldValue);
  TotalFormfields = TotalFormFields--;
}

// Save the updated file

QP.SaveToFile("flattened_pdf_form.pdf");
```

Duplicate form fields

Quick PDF Library allows you to add duplicate form fields to PDF files. This means that two or more form fields share the same name and if one of the form fields is updated then the other form fields are also updated.

```
/* Duplicate form fields */

// Create first form field

QP.NewFormField('Name', 1);
QP.SetFormFieldBounds(1, 100, 500, 200, 100);
QP.SetFormFieldBorderColor(1, 1, 0, 0);

// Create second form field and use same
// name as the first

QP.NewFormField('Name', 1);
QP.SetFormFieldBounds(2, 100, 400, 200, 100);
QP.SetFormFieldBorderColor(2, 0, 1, 0);
QP.SaveToFile('dupfields.pdf');
```

If you use the GetFormFieldValueByTitle or SetFormFieldValueByTitle functions then these functions will only work with the first occurrence of the field that has that title, it will not work with the duplicate field. So it is better to use the index with the GetFormFieldValue and SetFormFieldValue functions, instead of the title.

AcroForm vs XFA

There are two types of forms technologies used in PDF files: AcroForm and XFA. The AcroForm technology was added to the PDF specification in 1996 and standardized as part of the ISO standard for PDF in 2008. The use of XFA technology was added to the PDF specification in 2005 as a normative reference, meaning that it is not officially part of the PDF specification, just referenced by it. XFA is a proprietary Adobe technology and has not been standardized by the ISO, though its license permits other companies to use XFA technology.

Quick PDF Library provides extensive support for AcroForm form fields and limited support for XFA form fields. We recommend to our customers that they use the AcroForm technology because it is more supported by many PDF viewers and PDF editors, while XFA technology is primarily supported by Adobe products.

PDF JavaScript

Quick PDF Library lets you add JavaScript to the whole document, a specific page or other elements such as form fields, outlines and annotations.

Add global JavaScript to a PDF

It is possible to add JavaScript to the document which can be triggered by outlines, form fields, links and other elements. In this example we add a form field button which when clicked triggers a JavaScript message box.

```
/* Add JavaScript to a form field button in a new PDF */

// Use the AddGlobalJavaScript function to add
// JavaScript to a document.

QP.AddGlobalJavaScript("QPL", 'function Global_Sample(){app.alert("Quick PDF Library
rocks",3,0,"Something you should know...");}');

// Set the paper size

QP.SetPageSize("A4");

// Set the origin to the top-left corner

QP.SetOrigin(1);

// Set the measurement units to millimetres

QP.SetMeasurementUnits(1);

// Add the heading font

QP.AddStandardFont(5); // Helvetica bold
QP.SetTextSize(10);

ButtonText = "Big JavaScript Button:";
ButtonWidth = QP.GetTextWidth(ButtonText);
ButtonLocation = (QP.PageWidth()/2) - (ButtonWidth/2);
QP.DrawText(ButtonLocation, 68, ButtonText);

// Add the font to use for the form fields

FontID = QP.AddStandardFont(0); // Courier

QP.AddFormFont(FontID);

FieldIndex = QP.NewFormField("JavaScript Button:", 2);
QP.SetNeedAppearances(0);
QP.SetFormFieldBounds(FieldIndex, ButtonLocation, 70, ButtonWidth, 10);
QP.SetFormFieldFont(FieldIndex, QP.GetFormFontCount());
QP.SetFormFieldTextSize(FieldIndex, 12);
QP.SetFormFieldBorderColor(FieldIndex, 0.5, 0, 0);
QP.SetFormFieldBorderStyle(FieldIndex, 1, 0, 0, 0);
QP.SetFormFieldBackgroundColor(FieldIndex, 0.3, 0.3, 0.5);
QP.SetFormFieldValue(FieldIndex, "");
QP.SetFormFieldHighlightMode(FieldIndex, 3);
QP.FormFieldJavaScriptAction(FieldIndex, "U", "Global_Sample()");
```

```
// Save the file

QP.SaveToFile("Global JavaScript.pdf");
```

Here we have linked the global JavaScript to a form field but it could also be linked to an annotation hotspot (link) or an outline using the AddLinkToJavaScript function or the SetOutlineJavaScript function. You can also use these global JavaScript functions from page or document actions.

Page actions and JavaScript

Using the PageJavaScriptAction function you can trigger JavaScript on page open or page close events.

Document actions and JavaScript

Using the DocJavaScriptAction function you can trigger JavaScript on a document close, print and save event.

Retrieve JavaScript

Using these functions it is possible to retrieve JavaScript from the PDF:

- GetDocJavaScript -- returns JavaScript for document actions
- GetGlobalJavaScript -- returns global JavaScript packages
- GetOpenActionJavaScript -- returns JavaScript for document open action
- GetOutlineJavaScript -- returns JavaScript for outline actions
- GetPageJavaScript -- returns pave-level JavaScript open and close actions

PDF Markup Annotations

Annotations that are used primarily to markup a PDF are called markup annotations. Quick PDF Library provides support for sticky notes and retrieving information about markup annotations.

Add a sticky note

Using the AddNoteAnnotation function you can add a sticky note to a page in a PDF.

```
/* Add a sticky note to a PDF */

// Set the origin for the co-ordinates to be the
// top left corner of the page.

QP.SetOrigin(1);

// Adding a sticky note to a page is simple with the
// help of the AddNoteAnnotation function.

QP.AddNoteAnnotation(10, 10, 1, 200, 100, 335, 135, "My Fair Lady (1964)", "Many critics
found Wilfrid Hyde-White to be rather bland as Colonel Pickering...", 0.1, 0.2, 0.4, 1);

// When the QPL object is initiated a blank document
// is created and selected in memory by default. So
// all we need to do now is save the document to
// the local hard disk to see the changes that we've made.

QP.SaveToFile("sticky_note.pdf");
```

Get annotation properties

Using the GetAnnotDblProperty, GetAnnotIntProperty, GetAnnotStrProperty and AnnotationCount functions you can retrieve lots of information about annotations.

PDF Links

Links in PDF files are actually technically annotations. They are known as link annotations. They are not part of the content of a PDF but rather an interactive element that sits on top and can be added, moved or deleted with ease.

It is easy to confuse links in PDF files with how links work in HTML. In HTML the link and the text or image that is being linked are directly connected. They are part of the same element. But in PDF this is not the case. The link and the object being linked are not directly related. In the PDF specification the text on the page is part of the page description commands while the link itself is just an invisible area over the text stored in an "annotation" object.

At times the disconnect between the link annotation and the object that you're trying to link can make positioning the link over the object difficult. The co-ordinate points must be determined for each object and the annotation placed on top of the object that will be linked.

Add link to the web

Use the AddLinkToWeb function to add links to web pages.

```
/* Add a link to the web*/

// Set the origin for the co-ordinates to be the
// top left corner of the page.

QP.SetOrigin(1);

// Adding a link to the web is easy
// with the AddLinkToWeb function

QP.AddLinkToWeb(200, 100, 60, 20, "http://www.quickpdflibrary.com", 1);

// Hyperlinks and text are two separate
// elements in a PDF, so we'll draw some
// text now so that you know where the
// hyperlink is located on the page.

QP.DrawText(205, 114, "Click me!");

// When the QPL object is initiated a blank document
// is created and selected in memory by default. So
// all we need to do now is save the document to
// the local hard disk to see the changes that we've made.

QP.SaveToFile("link_to_web.pdf");
```

Add link to another page in same PDF

Use the AddLinkToPage function to add links from one page to another in the same PDF.

```
/* Add a link to a specific page in an external PDF */

// Set the origin for the co-ordinates to be the
// top left corner of the page.
```

```
QP.SetOrigin(1);

// Create page 2 in the default document.
// Page 2 is automatically selected after
// being created.

QP.NewPages(1);
QP.SetTextSize("24");
QP.DrawText(250, 114, "Page 2");

// Select page 1 again so that
// we can add our link to page 2.

QP.SelectPage(1);

// Adding a link to a page is easy using
// the AddLinkToPage function.

QP.AddLinkToPage(200, 100, 60, 20, 2, 0, 1);

// Hyperlinks and text are two separate
// elements in a PDF, so we'll draw some
// text now so that you know where the
// hyperlink is located on the page.

QP.DrawText(205, 114, "Click me!");

// Save the new PDF to disk

QP.SaveToFile("link_to_page.pdf");
```

Add link to another document

Use the AddLinkToFile function to add a link to a specific page and position in another PDF.

```
/* Add a link to a specific page in an external PDF */

// Set the origin for the co-ordinates to be the
// top left corner of the page.

QP.SetOrigin(1);

// Adding a link to a page in an external PDF
// is easy using the AddLinkToFile function.

QP.AddLinkToFile(200, 100, 60, 20, "example.pdf", 22, 200, 1, 1);

// Hyperlinks and text are two separate
// elements in a PDF, so we'll draw some
// text now so that you know where the
// hyperlink is located on the page.

QP.DrawText(205, 114, "Click me!");

// When the QPL object is initiated a blank document
// is created and selected in memory by default. So
// all we need to do now is save the document to
// the local hard disk to see the changes that we've made.

QP.SaveToFile("link_to_another_doc.pdf");
```

Add link to embedded file

Use the AddLinkToEmbeddedFile function to add a link to file embedded within the PDF.

```
/* Add a link to a file embedded within a PDF*/

// Embed a file into the current document

EmbeddedFileID = QP.AddEmbeddedFile("debenu final tm.pdf", "application/pdf");

// Set the origin for the co-ordinates to be the
// top left corner of the page.

QP.SetOrigin(1);

// Adding a link to an embedded file is
// easy using the AddLinkToEmbeddedFile function.

QP.AddLinkToEmbeddedFile(200, 100, 80, 20, EmbeddedFileID , "My embedded file", 1);

// Hyperlinks and text are two separate
// elements in a PDF, so we'll draw some
// text now so that you know where the
// hyperlink is located on the page.

QP.DrawText(205, 114, "Double-Click Me!");

// When the QPL object is initiated a blank document
// is created and selected in memory by default. So
// all we need to do now is save the document to
// the local hard disk to see the changes that we've made.

QP.SaveToFile("link_to_embedded_file.pdf");
```

Add link to link to JavaScript

Use the AddLinkToJavaScript function to add a link to a JavaScript action.

```
/* Add a link that executes a snippet of JavaScript when clicked on */

// Load a PDF with JavaScript

QP.LoadFromFile("doc_with_javascript.pdf");

// Use the AddLinkToJavaScript function to add a link
// that executes some JavaScript when clicked on

QP.AddLinkToJavaScript(220, 570, 190, 210, 'app.alert("Quick PDF Library
rocks",3,0,"Something you should know...");',1);

// Set the opening page and zoom factor
// for our sample file.

QP.SetOpenActionDestination(1, -1);

// Save the file to disk in the output folder.

QP.SaveToFile("link_to_javascript.pdf");
```

Add link to a destination

Use the AddLinkToDestination function to add a link to a destination in the same document. The target page, position and zoom level are specified by a destination object which can be created with the NewDestination function.

```
/* Add a link a link to a destination in the same document*/

// Set page origin to top left

QP.SetOrigin(1);

// Add a 2nd page to the default 1 page doc

QP.NewPages(1);

// Add a new destination on page 2, draw some text too

DestID = QP.NewDestination(2, 0, 7, 0, 200, 0, 0);

QP.DrawText(300, 300, "Page 2 Destination");

// Select page 1 again

QP.SelectPage(1);

// Use the AddLinkToDestination function to add
// a link to the destination that we've previously created

QP.AddLinkToDestination(200, 100, 60, 20, DestID, 1);

// Hyperlinks and text are two separate
// elements in a PDF, so we'll draw some
// text now so that you know where the
// hyperlink is located on the page.

QP.DrawText(205, 114, "Click me!");

// Save the file to disk.

QP.SaveToFile("link_to_destination.pdf");
```

Add link to named destination

Use the AddLinkToDestination and NewNamedDestination functions to add a link to a named destination in the same document. The target page, position and zoom level are specified by a destination object which can be created with the NewDestination function.

```
/* Add a link a link to a named destination in the same document */

// Set page origin to top left and add
// a 2nd page to the default 1 page doc

QP.SetOrigin(1);
QP.NewPages(1);

// Add a new destination on page 2

DestID = QP.NewDestination(2, 0, 7, 0, 200, 0, 0);

// Add a named destination reference to the
// destination object that we've just created
```

```
QP.NewNamedDestination("MyNamedDestination", DestID);

// Draw some text so we know where we are

QP.DrawText(300, 300, "Page 2 Named Destination");

// Select page 1 again

QP.SelectPage(1);

// Use the AddLinkToDestination function to add
// a link to the destination that we've previously created

QP.AddLinkToDestination(200, 100, 60, 20, DestID, 1);

// Hyperlinks and text are two separate
// elements in a PDF, so we'll draw some
// text now so that you know where the
// hyperlink is located on the page.

QP.DrawText(205, 114, "Click me!");

// Save the file to disk

QP.SaveToFile("link_to_destination.pdf");
```

PDF Bookmarks (Outlines)

Quick PDF Library provides extensive support for adding, editing and removing bookmarks from PDF files. In the PDF specification bookmarks are technically referred to as outlines and this is the terminology that we have used when naming our bookmark related functions but in this guide we have referred to them as bookmarks.

Find and list all bookmarks

Use the OutlineCount and OutlineTitle functions to find and list all bookmarks found in a PDF.

```
/* Find all bookmarks in a document and display them in a message box */

// Load the sample file from the input folder

QP.LoadFromFile("bookmarks.pdf");
iMainDocID = QP.GetDocumentID(QP.DocumentCount());

// Count the number of pages

iFilePages = QP.PageCount();

// Count the number of bookmarks

iNumOutlines = QP.OutlineCount();

// Declare the variable that we'll use for storing the bookmark titles

DocName = "";

// Cycle through each bookmark, saving the chunk as a separate file
```

```
for (n = 1; n < iNumOutlines; n++)
{
  // Select the main document
  QP.SelectDocument(iMainDocID);

  // Select the current bookmark
  iOutlineID = QP.GetOutlineID(n);

  // Get the bookmark title
  DocName = DocName + QP.OutlineTitle(iOutlineID) + "\n";
}

// Display a message box with all the bookmarks listed

MsgBox(DocName);
```

PDF Fonts

Quick PDF Library provides support for a wide range of different fonts, enabling you to embed and subset fonts. Support for Unicode characters is also provided.

Add a TrueType font

Use the AddTrueTypeFont function to embed a TrueType font in a PDF.

```
/* Embed a TrueType font within a PDF */

// Use the AddTrueTypeFont function to add a font to
// the default blank document and get the return
// value which is the font ID.

fontID1 = QP.AddTrueTypeFont("Arial Rounded MT Bold", 1);

// Select the font using its font ID

QP.SelectFont(fontID1);

// Draw some text onto the document to see if
// everything is working OK.

QP.DrawText(100, 700, "Arial Rounded MT Bold");

// Repeat exercise to see what a couple of other
// fonts will look like as well.

fontID2 = QP.AddTrueTypeFont("Times New Roman", 1);
QP.SelectFont(fontID2);
QP.DrawText(100, 650, "Times New Roman");

fontID3 = QP.AddTrueTypeFont("Century Gothic", 1);
QP.SelectFont(fontID3);
QP.DrawText(100, 600, "Century Gothic");

// Save the new document to the output folder.

QP.SaveToFile("embedded_fonts.pdf");
```

Add a standard font

Use the AddStandardFont function to embed a standard font to a PDF.

```
/* Add a Windows standard font to a PDF */

// Use the AddStandardFont function to add a font to
// the default blank document and get the return
// value which is the font ID.

fontID1 = QP.AddStandardFont(0);

// Select the font using its font ID

QP.SelectFont(fontID1);

// Draw some text onto the document to see if
// everything is working OK.
```

```
QP.DrawText(100, 700, "Courier");

// Repeat exercise to see what a couple of other
// fonts will look like as well.

fontID2 = QP.AddStandardFont(1);
QP.SelectFont(fontID2);
QP.DrawText(100, 650, "CourierBold");

fontID3 = QP.AddStandardFont(2);
QP.SelectFont(fontID3);
QP.DrawText(100, 600, "CourierBoldOblique");

fontID4 = QP.AddStandardFont(3);
QP.SelectFont(fontID4);
QP.DrawText(100, 550, "Helvetica");

fontID5 = QP.AddStandardFont(4);
QP.SelectFont(fontID5);
QP.DrawText(100, 500, "HelveticaBold");

// Save the new document to the output folder.

QP.SaveToFile("embedded_standard_fonts.pdf");
```

Add a subsetted font

Use the AddSubsettedFont function to embed a subset of a font in a PDF. This means that only the font information for the specified characters will be embedded, reducing the size of the document.

```
/* Add subsetted text to a PDF */

// Our string of Unicode text goes here

drawstr = "Hello World";

// Add a subset font for the text string

QP.AddSubSettedFont("Verdana", 1, drawstr);

// Remap the string to ensure that the correct character
// codes are used.

substr = QP.GetSubsetstring(drawstr);

// Draw the Unicode text onto the page

QP.DrawText(100, 600, substr);

// Save the new file to disk

QP.SaveToFile("subsetted_text.pdf");
```

Add a subsetted font with Unicode text

Unicode text can be added to PDFs using the AddSubsettedFont function.

```
/* Add subsetted Unicode text to a PDF */
```

```
// Our string of Unicode text goes here

drawstr = "你好";

// Add a subset font for the text string

QP.AddSubSettedFont("Verdana", 7, drawstr);

// Remap the string to ensure that the correct character
// codes are used.

substr = QP.GetSubsetstring(drawstr);

// Draw the Unicode text onto the page

QP.DrawText(100, 600, substr);

// Save the new file to disk

QP.SaveToFile("unicode_text.pdf");
```

Add a Type 1 font

Use the AddType1Font function to embed PostScript Type 1 fonts in a PDF.

```
/* Add a Type 1 font to a PDF */

// Load and add the Type 1 font using the
// AddType1Font function. Read function
// description for full requirements.

FontID = QP.AddType1Font("Allandale.PFM");

// Select the font that we've just added

QP.SelectFont(FontID);

// Set page origin to top left

QP.SetOrigin(1);

// Draw some sample text

QP.DrawText(100, 100, "Hello World");

// Save the new file to disk

QP.SaveToFile("embedded_type1_font.pdf");
```

Add a CJK font

Use the AddCJKFont function to add a Chinese, Japanese or Korean font to a PDF. Read the function description for full information.

```
/* Add a CJK font to a PDF */

// Add a CJK font and select it

FontID = QP.AddCJKFont(1);
QP.SelectFont(FontID);
```

```
// Specify text to be drawn

InputText = "中文文本"; // Chinese text

// Convert string of text to Unicode

UnicodeInputText = QP.ToPDFUnicode(InputText);

// Draw the string of text onto the PDF

QP.DrawText(100, 600, UnicodeInputText);

// Save the new file to disk

QP.SaveToFile("cjk_font.pdf");
```

Check PDF for font data

The HasFontResources function can be used to check a PDF for any font resources. If the PDF does not have any font resources then it can be assumed to be an image only PDF.

```
/* Check PDF for font resources */

// Load the PDF to test

QP.LoadFromFile("example.pdf");

// Check PDF for font resources. If this function returns 0
// then no font resources exist, but if it returns a non-zero value
// then font resources do exist.

HasFontResourcesResult = QP.HasFontResources();

if (HasFontResourcesResult == 0)
{
  MsgBox("No font resources found. Image only PDF.");
}
else
{
  MsgBox(HasFontResourcesResult + " font resource(s) found.");
}
```

Save embedded TrueType font data to file

Use the SaveFontToFile function to save embedded TrueType font data to file.

```
/* Save TrueType font data to file*/

// Load the PDF to extract TrueType font data from

QP.LoadFromFile("bookmarks.pdf");

// Locate all fonts in the PDF

FindFontsResult = QP.FindFonts();

// Loop through each font and save
// to file if the font is TrueType and embedded

for (i = 1; i < FindFontsResult; i++)
{
```

```
        FontID = QP.GetFontID(i);
        QP.SelectFont(FontID);
        if (QP.FontType() == 4)
        {
            QP.SaveFontToFile(QP.FontName() + ".ttf");
        }
    }
```

Some additional details:

1. First you need to load the PDF using the LoadFromFile function or one of the other LoadFrom* functions.
2. Then you need to count all of the fonts in the document using the FindFonts function. This function will return the total number of fonts in the document. Use this as an index to loop through each font in the document -- starting from 1 to the total number of fonts.
3. While looping through the index of the fonts, retrieve the fond ID and use this ID to select the font using the SelectFont function.
4. Only embedded TrueType fonts are supported by the SaveFontToFile function, so you need to use the FontType function to filter our all fonts that are not embedded TrueType fonts.
5. Finally, now you can save the embedded TrueType fonts to disk using the SaveFontToFile function as they are discovered in the loop

PDF Text

Quick PDF Library lets you draw text onto PDF files in a variety of different ways.

Draw text

Quick PDF Library lets you easily add simple text strings to PDF.

```
/* Draw a variety of different text on a new document */

// Set the origin for the drawing co-ordinates. In this case
// we'll draw the co-ordinates from the top left corner of the page.

QP.SetOrigin(1);

// Draw normal text

QP.DrawText(25, 25, "This text was drawn using the DrawText function.");

// Save the new file to the output folder

QP.SaveToFile("simple_text.pdf");
```

Draw styled text

Quick PDF Library gives you powerful control over styling your text with different fonts, text size and color and also the positing of the text on the page.

```
/* Draw a variety of different text on a new document */

// Set the origin for the drawing co-ordinates. In this case
// we'll draw the co-ordinates from the top left corner of the page.

QP.SetOrigin(1);

// Draw normal text

QP.DrawText(25, 25, "This text was drawn using the DrawText function.");

// Draw an arc of text

QP.DrawTextArc(150, 150, 100, 280, "This text was drawn using the DrawTextArc function.",
0);

// Draw text wrapped to a specified width

QP.DrawWrappedText(400, 50, 200, "This text was drawn using the DrawWrappedText function.
As you can see, the text automatically wraps when it exceeds the specified width.");

// Set the alignment of the text that we'll draw next

QP.SetTextAlign(2);

// Draw text in a text box. Specify width and height
// of the text box.

QP.DrawTextBox(350, 150, 200, 200, "This text was drawn using the DrawTextBox function.
Similar to the DrawText function except that the alignment can be specified.", 1);
```

```
// Change the alignment

QP.SetTextAlign(0);

// Draw rotated text

QP.DrawRotatedTextBox(300, 200, 200, 200, 90, "This text was drawn using the
DrawRotatedTextBox function.", 1);

// Draw text where each character has a space between it

QP.DrawSpacedText(15, 300, 10, "This text was drawn using the DrawSpacedText function.");

// Draw some more text

QP.DrawText(25, 25, "This text was drawn using the DrawText function.");

// Save the new file to the output folder

QP.SaveToFile("Text.pdf");
```

Add HTML text

Quick PDF Library provides some HTML text functions which give you greater control over the layout and styling of your text. See Appendix A of the Function Reference for a full list of the HTML tags that are available for use with Quick PDF Library.

```
/* Use HTML text to make styling and laying out text easier */

// Set the origin for the drawing co-ordinates. In this case
// we'll draw the co-ordinates from the top left corner of the page.

QP.SetOrigin(1);

// Draw a bullet list using DrawHTMLText

QP.DrawHTMLText(100, 100, 200, "<ul><li>Item 1</li><li>Item 2</li><li>Item 3</li><li>Item
4</li><li>Item 5</li></ul>");

// Draw a paragraph of text in a text box
// with some font and italic text

QP.DrawHTMLTextBox(200, 300, 200, 200, "<p>This is a text box. I can make some of it
<b>bold</b> and some <i>italic</i> using HTML tags.</p>");

// Save the new file to the output folder

QP.SaveToFile("html_text.pdf");
```

PDF Text Extraction

Extracting text from a PDF can at times be a difficult task. If a PDF has been scanned from paper and no OCR process has been performed then the page in a PDF might display what looks like text but in the objects in the PDF no text actually exists, just an image. You can use the HasFontResources function to check if a PDF has any text. Extracting text from a PDF that does indeed contain text objects is a relatively straight forward process using one of the text extraction functions.

Extract text

Simple text extraction where the text of a PDF is output in a human readable format into plain text is very simple using either the GetPageText, ExtractFilePageText or DAExtractPageText functions.

```
/* Extract text from a PDF */

// Load the test file and iterate through each page in the
// PDF and append it to a text file.

strInputFilePath = "apply_ fingerprint.pdf";

QP.LoadFromFile(strInputFilePath);

iNumPages = QP.PageCount(); // Calculate the number of pages

strText = "";
nPage = 0;

for(nPage = 1; nPage<=iNumPages; nPage++)
{
  strText = strText + QP.ExtractFilePageText(strInputFilePath, "", nPage, 0);
}

// Write all the data to a file
s = oFSO.CreateTextFile("extracted_text.txt", 1);
s.Writeline(strText);
s.Close();
```

Extract text advanced

Advanced text extraction where the text, along with co-ordinates, font information is returned as a CSV string is possible using the GetPageText, ExtractFilePageText or DAExtractPageText functions. Using options 3 or 4 of the ExtractOptions parameter you can have text returned as individual words or chunks.

```
/* Extract advanced text from a PDF */

// Load the file and iterate through each page in the
// PDF and append it to a text file.

strInputFilePath = "apply_ fingerprint.pdf";

QP.LoadFromFile(strInputFilePath);

iNumPages = QP.PageCount(); // Calculate the number of pages
```

```
strText = "";
nPage = 0;

for(nPage = 1; nPage<=iNumPages; nPage++)
{
  strText = strText + QP.ExtractFilePageText(strInputFilePath, "", nPage, 4);
}

// Write all the data to a file

s = oFSO.CreateTextFile("extracted_text.txt", 1);
s.Writeline(strText);
s.Close();
```

PDF Images

Quick PDF Library provides extensive support for adding images to PDF files, extracting images from PDF files, replacing images in PDF files and converting images to PDF and PDFs to images.

Add image to PDF

Use the AddImageFromFile and DrawImage functions to add images to PDF files. Supported image types: BMP, TIFF, JPEG, PNG, GIF, WMF and EMF.

```
/* Add an image to an existing PDF document */

// Load a file from disk. We'll place the image
// onto this file.

QP.LoadFromFile("example.pdf");

// Load your image into memory

QP.AddImageFromFile("example.bmp", 0);

// Get width and height of the image

lWidth = QP.ImageWidth();
lHeight = QP.ImageHeight();

// Draw the image onto the page using the specified width/height

QP.DrawImage(250, 450, lWidth, lHeight);

// Save the updated file to the output folder

QP.SaveToFile("pdf_with_image.pdf");
```

Extract image from PDF to file

Use the SaveImageToFile function to extract embedded images from PDF files to disk as JPG, BMP or TIFF images.

```
/* Extract an image from a PDF to file */

// Load the PDF

QP.LoadFromFile("example.pdf")

// Find all images in the PDF

ImagesFound = QP.FindImages()

// Get the image ID for the first image
// found in the PDF

ImageID = QP.GetImageID(1)

// Select the image using its ID

QP.SelectImage(ImageID)
```

```
// Determine the embedded images
// file type and then save to disk

ImageTypeFound = QP.ImageType();

if (ImageTypeFound == 0)
{
MsgBox("No image is selected");
}
else if (ImageTypeFound == 1)
{
QP.SaveImageToFile("embedded_image.jpg");
}
else if (ImageTypeFound == 2)
{
QP.SaveImageToFile("embedded_image.bmp");
}
else if (ImageTypeFound == 3)
{
QP.SaveImageToFile("embedded_image.tiff");
}
```

Replace an image

Use the ReplaceImage function to replace an old image in a PDF with a new image.

```
/* Replace an image in a PDF */

// Load the PDF

QP.LoadFromFile("example.pdf")

// Find all images in the PDF

ImagesFound = QP.FindImages()

// Get the image ID for the first image
// found in the PDF

OriginalImageID = QP.GetImageID(1)

// Add new image to the PDF

NewImageID = QP.AddImageFromFile("example.bmp", 0);

// Replace the old image with the new image

QP.ReplaceImage(OriginalImageID, NewImageID);

// Save the updated PDF

QP.SaveToFile("image_replaced.pdf");
```

Convert EMF to PDF

Use the ImportEMFFromFile function to convert EMF files to PDF.

```
/* EMF to PDF conversion */

// Load your image into memory
```

```
eImageID = QP.ImportEMFFromFile"tiger.emf", 0, 0);

// Select the imported image

QP.SelectImage(eImageID);

// Get width, height of the image

lWidth = QP.ImageWidth();
lHeight = QP.ImageHeight();

// Reformat the size of the page in the selected document

QP.SetPageDimensions(lWidth, lHeight);

// Draw the image onto the page using the specified width/height

QP.DrawImage(0, lHeight, lWidth, lHeight);

// Store the updated PDF where you like

QP.SaveToFile("tiger_emf.pdf");
```

Convert Image to PDF

Use the AddImageFromFile, SetPageDimensions and DrawImage functions to convert an image to PDF. Supported image types: BMP, TIFF, JPEG, PNG, GIF, WMF and EMF.

```
/* Convert an image to a PDF */

// Load a sample image into memory

QP.AddImageFromFile("sample123.jpg", 0);

// Get the width and height of the image

lWidth = QP.ImageWidth();
lHeight = QP.ImageHeight();

// Reformat the size of the page in the selected document

QP.SetPageDimensions(lWidth, lHeight);

// Draw the image onto the page using the specified width/height

QP.DrawImage(0, lHeight, lWidth, lHeight);

// Store the updated PDF where you like

QP.SaveToFile("sample123.pdf");
```

Convert PDF to Image

Converting a PDF to an image is easy using the RenderDocumentToFile function or any of the other functions that start with Render*.

```
/* PDF to image conversion */

// Load the 'debenu final tm.pdf' sample file
```

```
QP.LoadFromFile("example.pdf");

// Calculate the number of pages

iNumPages = QP.PageCount();

// Render each page of the document to a separate file.
// To view the images open the output folder.

QP.RenderDocumentToFile(72, 1, iNumPages, 0,"example.bmp");
```

PDF Color

Quick PDF Library provides you with extensive support for controlling the color of text and vector graphics that you add to PDF files.

```
/* Set the color for a variety of different objects -- text, vector graphics, etc */

// Specify the page size

QP.SetPageSize('A4');

// Specify the corner of the page where co-ordinates should start.
// In this example we'll specify the top left corner.

QP.SetOrigin(1);

// Set the line color and width

QP.SetLineColor(255, 0, 255);
QP.SetLineWidth(0.5);

// Set fill color and then draw a circle

QP.SetFillColor(0, 0, 255);
QP.DrawCircle(250, 600, 100, 2);

// Draw some text and specify various
// different colors for the text.

QP.DrawText(25, 25, "This text was drawn using the DrawText function.");

QP.SetTextColor(.4, .5, 0);
QP.DrawTextArc(150, 150, 100, 280, "This text was drawn using the DrawTextArc function.",
0);

QP.SetTextColor(0, .3, 0);
QP.DrawWrappedText(400, 50, 200, "This text was drawn using the DrawWrappedText function.
As you can see, the text automatically wraps when it exceeds the specified width.");

QP.SetTextAlign(2);
QP.SetTextColor(0, .1, .4);
QP.DrawTextBox(350, 150, 200, 200, "This text was drawn using the DrawTextBox function.
Similar to the DrawText function except that the alignment can be specified.", 1);

QP.SetTextAlign(0);
QP.DrawRotatedTextBox(300, 200, 200, 200, 90, "This text was drawn using the
DrawRotatedTextBox function.", 1);
QP.SetTextColor(.6, .11, .44);

QP.DrawSpacedText(15, 300, 10, "This text was drawn using the DrawSpacedText function.");
QP.DrawText(25, 25, "This text was drawn using the DrawText function.");

// Save the new document to disk

QP.SaveToFile("Color.pdf");
```

PDF Vector Graphics

Quick PDF Library provides extensive support for vector graphics enabling you to draw various different shapes onto your PDF files. In this example we draw multiple boxes and circles with a variety of different colors.

```
/* Draw some shapes on a new PDF */

// Set the page size for the new PDF

QP.SetPageSize('A4');

// Set the origin for the drawing co-ordinates
// to be the top left corner of the page.

QP.SetOrigin(1);

// Set the measurement unit that we'll use.
// In this example we'll use millimetres.

QP.SetMeasurementUnits(1);

// Set the color and the width for the line

QP.SetLineColor(255, 0, 0);
QP.SetLineWidth(0.5);

// Create a variety of boxes and circles
// using a loop.

for (x = 0; x <= 9; x++)
{
  for (y = 0; y <= 10; y++)
  {
    QP.DrawBox(5 + x * 20, 5 + y * 25, 20, 25, 0);
    QP.SetFillColor(0, 192, 0);
    QP.DrawCircle(15 + x * 20, 17.5 + y * 25, 8, 2);
  }
}

// Save the new file to the output folder

QP.SaveToFile("Shapes.pdf");
```

PDF Optional Content Groups (aka Acrobat Layers)

*Please note: optional content groups are what Acrobat and Adobe Reader refer to as layers. Optional content groups is the term used in the PDF specification. See the **API Overview** section for a more detailed analysis.*

Quick PDF Library provides extensive support for working with optional content groups in a variety of different ways, from creating new OCGs to editing or deleting existing OCGs.

Create OCGs

Use the NewOptionalContentGroup function to create optional content groups and add content to them.

```
/* These script shows you how to create multiple layers in a document. Layers are
technically called Optional Content Groups (OCGs). */

// Create four new optional content groups

OCG1 = QP.NewOptionalContentGroup("Layer 1");
OCG2 = QP.NewOptionalContentGroup("Layer 2");
OCG3 = QP.NewOptionalContentGroup("Layer 3");
OCG4 = QP.NewOptionalContentGroup("Layer 4");

// Select the page that you want the layers to be
// associated with.

QP.SelectPage(1);

// Specify top left corner for starting point
// of all drawing functions.

QP.SetOrigin(1);

// Add layer 1

QP.NewLayer();
QP.SelectLayer(1);
QP.DrawText(100, 100, "Layer 1");
QP.SetLayerOptional(OCG1);
QP.SetOptionalContentGroupVisible(OCG1, 1);

// Add layer 2

QP.NewLayer();
QP.SelectLayer(2);
QP.DrawText(200, 100, "Layer 2");
QP.SetLayerOptional(OCG2);
QP.SetOptionalContentGroupVisible(OCG2, 1);

// Add layer 3

QP.NewLayer();
QP.SelectLayer(3);
QP.DrawText(300, 100, "Layer 3");
QP.SetLayerOptional(OCG3);
QP.SetOptionalContentGroupVisible(OCG3, 1);

// Add layer 4

QP.NewLayer();
QP.SelectLayer(4);
QP.DrawText(400, 100, "Layer 4");
QP.SetLayerOptional(OCG4);
QP.SetOptionalContentGroupVisible(OCG4, 1);

// Save file to disk with new layers

QP.SaveToFile("option_content_groups.pdf");
```

Control visibility and printability of OCGs

Use the SetOptionalContentGroupVisible and SetOptionalContentGroupPrintable functions to control the visibility and printability settings of OCGs.

```
/* Create OCGs with differing visibility and printability settings */

// Create four new optional content groups

OCG1 = QP.NewOptionalContentGroup("Layer 1");
OCG2 = QP.NewOptionalContentGroup("Layer 2");
OCG3 = QP.NewOptionalContentGroup("Layer 3");
OCG4 = QP.NewOptionalContentGroup("Layer 4");

// Select the page that you want the layers to be
// associated with.

QP.SelectPage(1);

// Specify top left corner for starting point
// of all drawing functions.

QP.SetOrigin(1);

// Add OCG  1

QP.NewLayer();
QP.SelectLayer(1);
QP.DrawText(100, 100, "OCG  1");
QP.SetLayerOptional(OCG1);
QP.SetOptionalContentGroupVisible(OCG1, 1); // Set this OCG to be visible

// Add OCG  2

QP.NewLayer();
QP.SelectLayer(2);
QP.DrawText(200, 100, "OCG 2");
QP.SetLayerOptional(OCG2);
QP.SetOptionalContentGroupVisible(OCG2, 0); // Set this OCG to not be visible

// Add OCG  3

QP.NewLayer();
QP.SelectLayer(3);
QP.DrawText(300, 100, "OCG 3");
QP.SetLayerOptional(OCG3);
QP.SetOptionalContentGroupVisible(OCG3, 1); // Set this OCG to be visible
QP.SetOptionalContentGroupPrintable(OCG3, 1);  // Set this OCG to be printable

// Add OCG  4

QP.NewLayer();
QP.SelectLayer(4);
QP.DrawText(400, 100, "OCG 4");
QP.SetLayerOptional(OCG4);
QP.SetOptionalContentGroupVisible(OCG4, 1); // Set this OCG to be visible
QP.SetOptionalContentGroupPrintable(OCG4, 0);  // Set this OCG to be not printable

// Save file to disk with new OCGs

QP.SaveToFile("ocgs.pdf");
```

Remove OCGs

Use the OptionalContentGroupCount, GetOptionalContentGroupID and DeleteOptionalContentGroup functions to delete optional content groups from your PDFs.

```
/* Delete all optional content groups */

// Load the PDF that contains the OCGs

QP.LoadFromFile("ocgs.pdf");

// Count OCGs

OCGCount = QP.OptionalContentGroupCount();

// Loop through each OCG and delete it

for(i = 1; i <= OCGCount; i++)
{
  OCGID = QP.GetOptionalContentGroupID(i);
  QP.DeleteOptionalContentGroup(OCGID);
}

// Save file to disk with new layers

QP.SaveToFile("no_ocgs.pdf");
```

PDF Content Streams

Content streams are the primary means for describing the appearance of pages and other graphical elements. In Quick PDF Library content streams are called layers.

Combine content streams

Use the CombineLayers function to combine all content streams for the selected page into one content stream.

```
/* Combine all layers for each page in a PDF */

// Load PDF to process

QP.LoadFromFile("example.pdf");

// Count pages

int xPageCount = QP.PageCount();

// Go through each page and combine layers

for (int i = 1; i <= xPageCount; i++)
{
    QP.SelectPage(i);
    QP.CombineLayers();
}

// Save the updated file

QP.SaveToFile("example_updated.pdf");
```

Remove shared content streams

Use the RemoveSharedLayers function to ensure that none of the pages in the selected document share any content streams.

```
/* Remove shared content streams */

// Load the PDF
QP.LoadFromFile(“example.pdf”);

// Remove shared content streams

QP.RemoveSharedLayers();

// Save the updated PDF

QP.SaveToFile(“example_updated.pdf”);
```

Encapsulate layers

Use the EncapsulateLayer function to surround the current layer with "save graphics state" and "restore graphics state" operators.

```
/* Encapsulate content streams in a PDF */

// Load the PDF

QP.LoadFromFile("example.pdf");

// Count pages

xPageCount = QP.PageCount();

// Go through each page and encapsulate layers

for (int i = 1; i <= xPageCount; i++)
{
    QP.SelectPage(i);
    int xLayerCount = QP.LayerCount();

    for (int x = 1; x <= xLayerCount; x++)
    {
        QP.SelectLayer(x);
        QP.EncapsulateLayer();
    }
}

// Save the updated file

QP.SaveToFile("example_updated.pdf");
```

PDF Attachments

PDF files can contain fully embedded files. These embedded files are accessible from the Attachments menu in conforming PDF readers. There is no limitation on the types of files which can be embedded, although recent versions of Adobe Acrobat have not permitted executable depending on the security settings specified in the preferences.

Embed a file in a PDF

Use the EmbedFile function to embed a file within a PDF.

```
/* Add a file attachment to a PDF */

// Load the PDF

QP.LoadFromFile("example.pdf");

// Embed a Word document (can be any media type) in
// selected document.

QP.EmbedFile("Link to embedded file...", "example.docx", "application/msword");

// Save the updated file to disk

QP.SaveToFile(example_updated.pdf");
```

Count embedded files in a PDF

Use the EmbeddedFileCount function to count the number of embedded files in a PDF.

```
/* Count embedded files in a PDF */

// Load the PDF

QP.LoadFromFile("example.pdf");

// Count the embedded files

embeddedFiles = QP.EmbeddedFileCount();

// Display number of embedded files

MsgBox("Number of Embedded Files: " + embeddedFiles);
```

Extract embedded file from PDF

Use the GetEmbeddedFileContentToFile function to extract the content of an embedded file from a PDF.

```
/* Extract embedded file from a PDF*/

// Load PDF with embedded file

QP.LoadFromFile("pdf_with_embedded_file.pdf");

// Extract embedded file content to file on disk

QP.GetEmbeddedFileContentToFile(1, "JavaScript.docx");
```

PDF Barcodes

Quick PDF Library provides support for Code39 (or Code 3 of 9), EAN-13, Code128, PostNet and Interleaved 2 of 5 barcodes.

Draw a barcode

Use the DrawBarcode function to add barcodes to your PDF.

```
/* Draw a variety of different barcodes on a new page in a PDF */

// Set the origin for the co-ordinates to be
// the top left corner of the page.

QP.SetOrigin(1);

// Draw three different barcodes

QP.DrawBarCode(25, 25, 150, 100, "MyBarcode256", 1, 0);
QP.DrawBarCode(225, 50, 100, 600, "MyBarcode257/RC", 1, 0);
QP.DrawBarCode(350, 50, 200, 150, "MyBarcode258", 3, 0);

// Save the new file

QP.SaveToFile("Barcodes.pdf");
```

Draw text under a barcode

Use the GetBarcodeWidth, DrawBarcode and Drawtext functions to add a barcode and draw text underneath it.

```
/* Draw text under a barcode */

// Add the Helvetica standard font

QP.AddStandardFont(4);

// Get the width of the barcode with a bar width of 1 unit

BarcodeWidth = QP.GetBarcodeWidth(1, "12345", 1);

// Draw the barcode

QP.DrawBarcode(100, 500, BarcodeWidth, 100, "12345", 1, 0);

// Center the text

QP.SetTextAlign(1);

// Draw the text at 10pt

QP.SetTextSize(10);
QP.DrawText(100 + BarcodeWidth / 2, 500 - 200 - 5 - QP.GetTextHeight(), "12345");

// Save the updated PDF

QP.SaveToFile("text_under_barcode.pdf");
```

PDF Metadata

Quick PDF Library lets you control metadata in PDF files as the Title, Author, Subject and Keyword properties, as well as custom metadata.

Set document properties

Use the SetInformation function to add Title, Author, Subject and Keyword properties to a PDF.

```
/* Add standard document information (metadata) to a document. */

// Custom information can be added using the

QP.SetInformation(0, "1.9");
QP.SetInformation(1, "John Smith");
QP.SetInformation(2, "The Life and Times of John Smith");
QP.SetInformation(3, "A very special book");
QP.SetInformation(4, "book, paperback, ebook");
QP.SetInformation(5, "Mother Earth");
QP.SetInformation(6, "Humanity");

// Draw some explanatory text onto the blank document.

QP.DrawText(100, 700, "Select Ctrl+D and then click on the Description tab to see the
document information.");

// Save the new file

QP.SaveToFile("document_information.pdf");
```

Get document properties

Use the GetInformation function to get document properties from PDF files.

```
/* Get and display the document properties (metadata) for a document */

// Load a PDF

QP.LoadFromFile("document properties.pdf");

// Extract information from the document

DocInformation = "";
DocInformation = DocInformation + "PDF Version: " + QP.GetInformation(0) + "\n";
DocInformation = DocInformation + "Author: " + QP.GetInformation(1) + "\n";
DocInformation = DocInformation + "Title: " + QP.GetInformation(2) + "\n";
DocInformation = DocInformation + "Subject: " + QP.GetInformation(3) + "\n";
DocInformation = DocInformation + "Keywords: " + QP.GetInformation(4) + "\n";
DocInformation = DocInformation + "Creator: " + QP.GetInformation(5) + "\n";
DocInformation = DocInformation + "Producer: " + QP.GetInformation(6) + "\n";
DocInformation = DocInformation + "Creation date: " + QP.GetInformation(7) + "\n";
DocInformation = DocInformation + "Modification date: " + QP.GetInformation(8) + "\n";

// Display the information to the user

MsgBox(DocInformation);
```

Set custom metadata

Use the SetCustomInformation function to add custom metadata to PDF files.

```
/* Add custom information (metadata) to documents. */

// Custom information can be added using the
// SetCustomInformation function. Create your
// own Key and Value and call the function as
// many times as you require.

QP.SetCustomInformation("FirstName", "John");
QP.SetCustomInformation("LastName", "Smith");
QP.SetCustomInformation("Coolness Level", "Very Cool");

// Draw some explanatory text onto the blank document.

QP.DrawText(100, 700, "Select Ctrl+D and then click on the Custom tab to see the custom
information.");

// Save the new file

QP.SaveToFile("custom_information.pdf");
```

Get custom metadata

Use the GetCustomInformation to retrieve custom metadata from PDF files.

```
/* Get custom information (metadata) from PDFs. */

QP.LoadFromFile("custom_information.pdf");

// Custom information/metadata can be retrieved
// from PDF files using the GetCustomInformation function.
// Custom metadata is stored in name/value pairs,
// so in order to retrieve metadata you must know the
// name of the entry that you want to retrieve.

myCustomInfo1 = QP.GetCustomInformation("FirstName");
myCustomInfo2 = QP.GetCustomInformation("LastName");
myCustomInfo3 = QP.GetCustomInformation("Coolness Level");

// Display the custom metadata that we've just retrieved

MsgBox(myCustomInfo1);
MsgBox(myCustomInfo2);
MsgBox(myCustomInfo3);
```

Misc

This section contains a variety of useful information that doesn't fall into any other category.

PDF Resources

There are many PDF resources available on the web, but we've compiled a few of the really good ones for you below to give you a head start. It isn't necessary to know all about the technical details of PDF when you work with our library, so this is just for the curious.

- Planet PDF -- PDF news, tips and in-depth articles

- Planet PDF Forum -- PDF discussion forum
- Planet PDF Q&A -- PDF questions and PDF answers
- PDF Tutorials by Jim King -- easy to follow technical presentations on PDF
- Inside PDF -- a blog from Jim King, a Senior Scientist at Adobe Systems
- PDF basics -- articles on the basic building blocks of PDF
- PDF Specification -- PDF Reference and Adobe Extensions to the PDF Specification

There are many more PDF resources available on the web, but these are some of the best. There are also some more specific resources dealing with PDF available at Planet PDF.

- Introduction to Acrobat & PDF
- Accessible PDF
- PDF Color
- PDF Preflight
- Introduction to Acrobat Development
- Introduction to Acrobat JavaScript

Useful 3rd Party Applications

These are just some useful applications that we use in our day to day testing.

- AsTiffTagViewer -- a free TIFF tag (code, data type, count, value) viewer application
- Quick PDF Tools - application for manipulating PDFs, built using Quick PDF Library
- ARTS PDF Workshop -
- tiffinfo.exe (uses libTIFF) - analyze TIFF images, discussed on our blog
- JPEGsnoop -- JPEG file decoding utility
- Nitro PDF Reader -- free PDF viewer with some form filling and other advanced features
- Foxit PDF Reader - free lightweight PDF viewer
- Adobe Acrobat -- Acrobat is a very useful tool if you work with PDF every day
- PDF CanOpener -- this is an Acrobat plug-in for COS level manipulation of PDFs
- GPL Ghostscript -- software based on interpreter for PostScript and PDF
- GSview -- graphical interface for Ghostscript, display PDF and PostScript
- Primo PDF -- open source PDF printer driver
- EMF Explorer -- EMF/WMF previewing, conversion and printing
- Enfocus Browser -- browse internal objects and dictionaries of PDF files
- Notepad++ -- plain text editor useful for opening PDF files in plain text
- AMP Font Viewer -- useful font manager for keeping track of installed fonts

You will notice a couple of different PDF viewers on this list, we always find it is a good idea to test your PDF files in a variety of different PDF viewers because you never know what PDF viewer your client is going to be using.