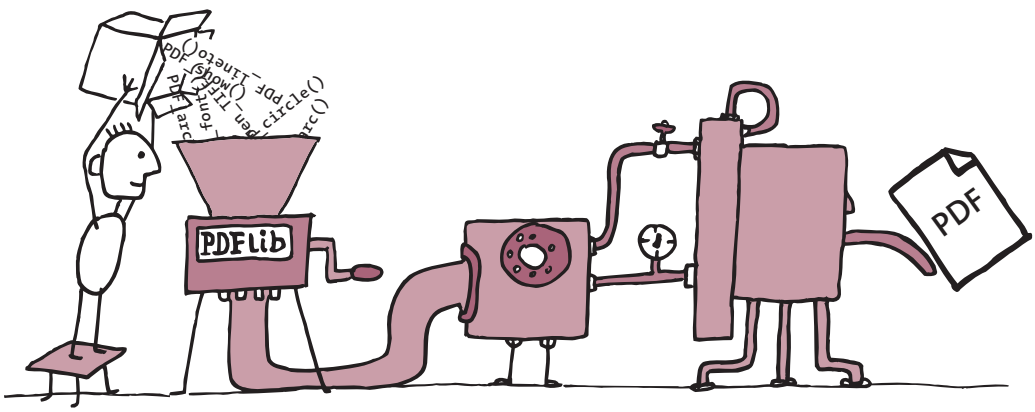
Thomas Merz

PDFlib Reference Manual

API Reference for PDFlib 0.6, a C library for dynamically generating PDF files



thomas merz
CONSULTING PUBLISHING

http://www.ifconnection.de/~tm

Contents

1 Introduction

What is PDFlib? PDFlib is a library of C routines which allow you to programmatically generate files in Adobe's Portable Document Format PDF. PDFlib acts as a backend processor to your own programs. While you (the programmer) are responsible for retrieving or maintaining the data to be processed, PDFlib takes over the task of generating the PDF code which graphically represents your data. While you must still format and arrange your text and graphical objects, PDFlib frees you from the internals and intricacies of PDF. Although being far from complete, PDFlib already offers many useful functions for creating text, graphics, images and hypertext elements in PDF files.

PDFlib features. The PDFlib API offers the following major features:

- PDF documents of arbitrary length and page formats
- text output in different fonts
- the ability to embed PostScript font descriptions
- common vector graphics primitives – lines, curves, arcs, rectangles, etc.
- read PostScript font metrics from AFM files
- process common graphics file formats, e.g. TIFF, GIF, JPEG
- generate hypertext elements such as bookmarks
- features supported in PDF but not accessible in Acrobat software, e.g., page transition effects like shades and mosaic.

All of these may be achieved by using a simple API without the application programmer being directly involved with PDF objects or operators.

What can I use PDFlib for? PDFlib's primary target is creating dynamic PDF on the World Wide Web. Similar to HTML pages dynamically generated with a CGI script on the Web server, you may use a PDFlib program for dynamically generating PDF reflecting user input or some other dynamic data, e.g. data retrieved from the Web server's database. The PDFlib approach offers several advantages as opposed to creating PDF from PostScript files with Acrobat Distiller:

- The PDFlib »driver« can be integrated directly in the application generating (or otherwise handling) the data, eliminating the convoluted creation path application–PostScript–Acrobat Distiller–PDF.
- As an implication of this straightforward process, PDFlib is by far the fastest PDF-generating method, making it perfectly suited for the Web.
- PDFs need not be created ahead of time and stored on the server, but can be generated if needed. This is a big win not only if you want to deal with dynamic data which do not exist prior to the Web interaction, but also if large amounts of data have to be handled which make it impractical to pre-generate all the necessary PDF.

However, PDFlib is not restricted to dynamic PDF on the Web. Equally important are all kinds of converters from X to PDF, where X represents any text or graphics file format. Again, this replaces the sequence X–PostScript–PDF with simply X–PDF, which offers many advantages for some common graphics file formats like GIF or JPEG. Using such a PDF converter, batch converting lots of text or graphics files is much easier than using the Adobe Acrobat suite of programs. Several converters of this kind are supplied with the library.

Which platforms are supported? PDFlib is a portable ANSI C library which may be used on any reasonable operating system platform. Although being developed and tested on Unix and Windows NT systems, the library does not depend on any Unix specific features and may well be used on other platforms. Actually, since the library doesn't need any user interface, porting to other platforms is simply a matter of arranging a suitable build process. The supplied library clients, however, use Unix-style command line options which may not be considered appropriate for other platforms.

Scripting support. PDFlib not only offers an ANSI C programming interface but may also be used from within scripting languages. Currently Perl, Tcl, and Python are supported. You can call PDFlib routines (as described in this manual) from your Perl, Tcl, or Python scripts. This greatly simplifies the process of writing useful PDF-generating applications or scripts.

Requirements for using PDFlib. PDFlib tries to make possible PDF generation without wading through the 400 page PDF specification. While PDFlib tries to hide technical PDF details from the user, a general understanding of PDF is highly desirable. In order to make the best use of PDFlib, application programmers should be familiar with the graphics model of PostScript (and therefore PDF). However, a reasonably experienced application programmer who has dealt with any graphics API for screen display or printing of his application data shouldn't have much trouble adapting to the PDFlib API which will be described in this manual.

About this manual. This manual describes the API implemented in PDFlib. It does not describe the process of compiling the library on certain platforms. This is covered in the accompanying text files. The function interfaces described in this manual are believed to remain unchanged during future PDFlib development. There may well be other useful functions contained in the library which are not described here. Support for these, however, may be dropped in the future or – more likely – they may remain in the library but have their interfaces changed.

This manual doesn't even attempt to explain PDF features or internals. Please refer to the material at the end of the manual for further reference.

2 PDFlib Programming

This chapter is meant to give you a jump start to PDFlib programming. The supplied sample programs don't go into details but will provide a framework for writing your own applications. It is suggested that you try these samples and then take a look at the supplied demo clients which are part of the library distribution.

2.1 A Programming Classic in C

Being a well-known classic, the »Hello, world!« sample will be used for the first PDFlib program. It uses PDFlib to generate a one-page PDF file with the text »Hello, world!« on the page:

```
/* hello.c
 * PDFlib sample application
 * (c) Thomas Merz 1997-98
 */

#include <stdio.h>

#include "pdf.h"

#define FILENAME    "hello.pdf"
#define FONTNAME    "Helvetica-Bold"
#define FONTSIZE    24.0

void
main(int argc, char *argv[])
{
    FILE     *pdffile;                  /* PDF output file pointer        */
    PDF      *p;                        /* pointer to the PDF structure   */
    PDF_info *info;                     /* pointer to document info block */
    char *filename = FILENAME;

    if (argc > 1)
        filename = argv[1];

    if ((pdffile = fopen(filename, "w")) == NULL) {
        fprintf(stderr, "Error: cannot open PDF file %s.\n", filename);
    }

    info = PDF_get_info();                          /* get info block */
    info->Creator       = "hello.c";                /* and fill some  */
    info->Author        = "Thomas Merz";            /* elements       */
    info->Title         = "Hello, world!";

    p = PDF_open(pdffile, info);                    /* open new PDF file  */

    PDF_begin_page(p, a4.width, a4.height);         /* start a new page   */
```

```
    PDF_set_font(p, FONTNAME, FONTSIZE, winansi);
    PDF_set_text_pos(p, 50, 700);
    PDF_show(p, "Hello, world!");
    PDF_end_page(p);                                /* close page         */

    PDF_close(p);                                   /* close PDF document */
    exit(0);
}
```

That's it! Note that we not only produced some text output but also populated some of the PDF's document info fields with suitable values.

You will be able to gather the basic structure of a PDFlib program from this sample. In order to work with the library you need a PDF object, pardon me: a pointer to a PDF structure. This is an opaque data structure which is used extensively in the library and is needed for almost every API call. In order to create this structure, you'll need a conventional FILE pointer for the PDF output file and a pointer to a so-called info block. This holds general information about the PDF (visible in Acrobat's »Document Info« box) as well as several options for generating the file. You must allocate the info block with the `PDF_get_info()` routine (and fill the elements of this block) before opening the PDF with `PDF_open()`.

Each page to be created must be bracketed with `PDF_begin_page()` and `PDF_end_page()`. As you can see, each page can have its own dimensions. In this case we used one of the predefined page sizes to create an A4-sized page.

2.2 Scripting with Perl, Tcl, and Python

Now let's rewrite the »Hello, world!« sample in the three supported scripting languages. We will start with the Perl version. It's quite easy (admittedly it lacks any error handling) and looks similar to the C version above:

```
#!/usr/bin/perl
use pdflib;
package pdflib;

$fp = fopen("hello_perl.pdf", "w");
$ip = PDF_get_info();
$ip->{Creator} = "hello.pl";
$ip->{Author} = "Thomas Merz";
$ip->{Title} = "Hello world (Perl)";
$ip->{fontpath} = "../../fonts";

$p = PDF_open($fp, $ip);

PDF_begin_page($p, $a4->{width}, $a4->{height});
PDF_set_font($p, "Helvetica-Bold", 18.0, $winansi);
PDF_set_text_pos($p, 50, 700);
PDF_show($p, "Hello world!");
PDF_continue_text($p, "(says Perl)");
```

```
PDF_end_page($p);
PDF_close($p);
```

Although the Tcl syntax is somewhat different, the Tcl version isn't very complicated either:

```
#!/usr/local/bin/tclsh
# simple loading the shared-library:
load ./pdflib.so
# using pdflib as package:
# lappend is unnecessary if installed at some right place

lappend auto_path .
package require pdflib

# doesn't yet work
#namespace import pdflib::*

set fp [fopen hello_tcl.pdf w]

set ip [PDF_get_info]

PDF_info_Creator_set $ip "hello.tcl"
PDF_info_Author_set $ip "Thomas Merz"
PDF_info_Title_set $ip "Hello world (Tcl)"
PDF_info_fontpath_set $ip "../../fonts"

set p [PDF_open $fp $ip]

PDF_begin_page $p 595 842
PDF_set_font $p Helvetica-Bold 18.0 $winansi
PDF_set_text_pos $p 50 700
PDF_show $p "Hello world!"
PDF_continue_text $p "(says Tcl)"
PDF_end_page $p
PDF_close $p
```

Finally, let's try the Python version:

```
#!/usr/bin/python
from pdflib import *

fp = fopen("hello_python.pdf", "w")
ip = PDF_get_info()
PDF_info_Creator_set(ip, "hello.py")
PDF_info_Author_set(ip, "Thomas Merz")
PDF_info_Title_set(ip, "Hello world (Python)")
PDF_info_fontpath_set(ip, "../../fonts")

p = PDF_open(fp, ip)

PDF_begin_page(p, 595, 842)
PDF_set_font(p, "Helvetica-Bold", 18.0, winansi)
PDF_set_text_pos(p, 50, 700)
```

```
PDF_show(p, "Hello world!")
PDF_continue_text(p, "(says Python)")
PDF_end_page(p)
PDF_close(p)
```

Strictly speaking, the PDFlib scripting API isn't yet documented anywhere. However, given the similarity between the C API and the scripting API, any reasonably experienced programmer should be able to use this C API reference manual also for scripting by looking at the examples and using his own wits.

Note that currently there isn't any error or exception handling with respect to PDFlib scripting. This is obviously ridiculous and will be fixed soon.

2.3 General Programming Issues

Naming Conventions. PDFlib contains two kinds of C routines – the public ones constitute the API, the private ones are used internally in the library. Consequently, there are two header files: pdf.h contains all public definitions to be used by a PDFlib program, whereas p_intern.h describes the undocumented internal functions. The API functions are described in this manual, whereas the undocumented functions are not (in order to justify their name). Do not include p_intern.h in your own source file! If you feel inclined to do so, most certainly you use functions in an inappropriate way which may result in damaged PDF files!

Note that there are different naming conventions for public and private functions: All functions to be called by PDFlib clients have names like `PDF_*()`, whereas private functions are called `pdf_*()`. Again: Do not use `pdf_*()` functions!

PDFlib Data Structures. All structures and data types to be used by library clients are declared in pdf.h. Please refer to this include file if you encounter any unknown data types starting with PDF_. It is strongly advised to take a look at pdf.h before starting PDFlib programming

General PDFlib Program Structure. PDFlib applications must obey certain structural rules. It is very important to understand that PDFlib does not check all API calls for their structural correctness (although many client bugs are discovered and recorded in the final stage of PDF generation).

However, the rules are very easy to understand and to obey. In the API reference, the restrictions are noted. Writing applications according to these restrictions should be straightforward. For example, you don't have to think about opening a page first before closing it, etc. The main program structure of a PDFlib application is as shown in the »Hello, world!« sample program.

Error Handling and Programming Restrictions. In generating PDF documents with PDFlib, several classes of events may raise errors and warnings:

- external events (e.g., no disk space)
- not adhering to programming restrictions (e.g., closing a document before opening it)
- wrong parameters to API functions (e.g., trying to draw a circle with a negative radius)
- bad data (e.g., trying to place a corrupt JPEG file)

If the library catches an error, a central error handler is called in order to deal with the situation. There are several classes of errors: informational messages, warnings, fatal errors, and PDFlib internal errors.

The default error handler issues an appropriate message on *stderr* in all cases and exits in case of a fatal or internal error. The PDF output file is left in an inconsistent state! Since this may not be adequate for a library routine, the default error handler can be replaced with a user-supplied one. A user-defined error handler may, for example, present the error messages in a GUI dialog box and may take other measures instead of aborting. More details on installing a custom error handler can be found in Section 3.3, »Error Handling«.

Serious projects are strongly advised to supply their own error handler to PDFlib!

2.4 Coordinate System and Vector Graphics

2.4.1 The Coordinate System

PDF's default coordinate system is used within PDFlib, although this can be changed by rotating, scaling, or translating the coordinate system. The default coordinate system has the origin in the lower left corner of the page and uses the DTP point as its unit:

```
1 pt = 1 inch / 72 = 25.4 mm / 72 = 0.3528 mm
```

2.4.2 The Graphics States

There are several graphics state parameters in PDF which together make up the Graphics State. See [1] for an explanation of these parameters. The following tables specify which of the parameters are handled in PDFlib. »NYI« specifies that the corresponding graphics state parameter is not yet implemented in PDFlib.

Special Graphics State

Parameter	PDFlib implementation	Default
clipping path	OK	page size
transform. matrix	OK	identity matrix
current point	OK	none

General Graphics State

Parameter	PDFlib implementation	Default
flatness	OK	0
line cap	OK	0
line dash	OK	solid line
line join	OK	0
line width	OK	1
miter limit	OK	10
device-dependent parameters	not supported	

Color and Color Space

Parameter	PDFlib implementation	Default
fill color	gray values or RGB triples	black
stroke color	gray values or RGB triples	black
fill color space	DeviceGray or DeviceRGB	black
stroke color space	DeviceGray or DeviceRGB	black
rendering intent	NYI	black

Text State

Parameter	PDFlib implementation	Default
character spacing	OK	0
word spacing	OK	0
horizontal scaling	OK	100%
leading	OK	0
text font	OK	none
text font size	OK	none
text matrix	OK	identity matrix
rendering mode	OK	solid fill
text rise	OK	0

2.5 Text Output and Fonts

2.5.1 Font Embedding and AFM Files

PDFlib is capable of embedding font descriptions into the generated PDF output.

Restrictions:

- Currently only PostScript fonts can be embedded
- Font subsetting is not yet implemented
- Only fonts in ASCII format (PFA) are supported
- Font files must use Unix-style line ends

Alternatively, a font descriptor can be embedded instead of the font outline data. In both cases (font outline embedding and font descriptor embedding) an AFM file containing metrics information for the used font must be supplied. In case of font embedding, a font file must also be available.

PDF and Acrobat viewers support a core set of 14 fonts which need not be embedded in any PDF file. AFM files for these fonts are included in the PDFlib distribution in order to allow metrics calculations for formatting purposes. The core fonts are the following:

```
Courier, Courier-Bold, Courier-Oblique, Courier-BoldOblique,
Helvetica, Helvetica-Bold, Helvetica-Oblique, Helvetica-BoldOblique,
Times-Roman, Times-Bold, Times-Italic, Times-BoldItalic,
Symbol, ZapfDingbats
```

In order to let PDFlib find the font and AFM files, some rules must be obeyed: The font file name must consist of the PostScript font name (i.e., the value of the `/FontName` key), the AFM file name must consist of the PostScript font name plus the suffix ».afm«.

PDFlib first looks for the font and metrics files in the current directory and then in a special font directory which can be supplied in the `PDF_info` block when opening a new PDF. Default value for the font path is »./fonts«, but this may be changed in pdf.h using a macro definition.

2.5.2 Character Sets and Encoding

PDF supports several encoding methods for text fonts. PDFlib includes provisions for supporting diverse encoding vectors in the generated PDF output. However, encoding vector support is not yet completed. For this reason, text strings used in PDFlib calls are always interpreted according to Windows' ANSI encoding vector.

Since Windows ANSI encoding is for the most part identical to ISO Latin 1 encoding, on Windows and Unix systems one may assume the operating systems' encoding vector as far as PDFlib strings are concerned.

Restriction:

- Multiple encoding vectors are not yet supported.
- Using character code 0 is not supported (since PDFlib uses C-style strings internally).

2.6 Raster Images

Embedding raster images in the generated PDF is an important feature of PDFlib. Although support for raster image file formats is far from complete, PDFlib already deals with some common graphics file formats:

- JPEG images: PDF supports only the »baseline« flavor of JPEG compression. However, JPEG baseline images account for the vast majority of JPEG files. Note, however, that progressive JPEGs and some other non-baseline flavors of the JPEG image file format are not supported in PDF, and therefore in PDFlib.
- GIF images: PDFlib contains internal GIF handling code. Regular or interlaced GIFs may be used.
- TIFF images: Sam Leffler's TIFFlib can be plugged into PDFlib in order to support zillions of TIFF compression and encoding flavors. Although there has much to be done concering the TIFFlib integration, many TIFF flavors can already be used.

Restrictions:

- Currently raster image output in PDF files is uncompressed.
- TIFF handling is very slow.

Embedding raster images with PDFlib is a multi-step process which is easy to accomplish. First, the image file has to be opened with a PDF-lib function which does a brief analysis of the image parameters. The parameters are supplied in the returned `image` structure. This structure can be used in a call to `PDF_place_image()`, along with positioning and scaling parameters:

```
if ((image = PDF_open_JPEG(p, JPEGFILENAME)) == NULL) {
        fprintf(stderr,"Error: Couldn't analyze image %s - skipped.\n",
                JPEGFILENAME);
        continue;
}

scale = 1.0;

PDF_begin_page(p, image->width * scale, image->height * scale);

PDF_place_image(p, image, 0.0, 0.0, scale);
PDF_close_image(p, image);

PDF_end_page(p);
```

Alternatively, PDFlib can be used to embed »inline images« in PDF files with a call to the PDF_place_inline_image() routine. [1] recommends using this feature only for images with less than 4KB in size.

2.7 Color Spaces

2.7.1 Color Spaces for Text and Graphics

PDFlib supports two color spaces for specifying text and graphics colors: DeviceGray and DeviceRGB. You can specify the gray level of text or path objects by supplying a value between 0 and 1 (0 means black, 1 means white).

RGB colors are specified using three byte values with the corresponding red, green, and blue components.

Restriction:

- Other color spaces for text and graphics are not supported.

2.7.2 Color Spaces for Raster Images

Concerning raster images, more color spaces are supported than for text or graphics. However, it isn't necessary to directly specify the color space for an embedded image since PDFlib automatically detects the kind of color space needed for a particular image.

2.8 Binary Output and Compression

2.8.1 Binary and ASCII Output

PDFlib is capable of generating ASCII and binary output files. Obviously, ASCII files may end up larger than their binary counterparts, but they are useful for debugging or otherwise examining PDF files. ASCII/binary output can be selected in the PDF_info block. By default, PDFlib generates binary files.

2.8.2 Compression

Currently, all PDFlib output is uncompressed. It is anticipated that the public-domain ZLIB compression library will be plugged into PDFlib, supplying a highly effective and PDF-compatible compression method which may be applied to text, graphics, and raster images.

3 PDFlib API Reference

3.1 Data Structures

The PDFlib programmer is strongly advised to take a closer look at the interface file pdf.h which defines the relevant data types and functions exported from the library. Although this section doesn't attempt to cover all these, the most important data types are briefly sketched.

struct PDF_info
This structure holds general information about a PDF file, such as title, creator, subject, etc. A PDF_info block is necessary for creating a new PDF document with PDFlib. The programmer specifies the relevant information and uses it in a call to the PDF_open() routine.

struct PDF
This is a handle used to refer to a PDF document which is to be generated by PDFlib. The library supplies such a handle with the PDF_open() routine. The handle has to be used throughout all operations on that particular document. The contents of the structure are considered private to the library, only pointers to the PDF structure are used at the API level.

struct PDF_image
This is an opaque data structure used to refer to images. The contents of PDF_image are considered private, only pointers are used at the API level.

3.2 General Functions

PDF_info *PDF_get_info(void)
Return a pointer to a PDF_info block. The elements of this block may optionally be populated with user data and must be used for creating a new PDF file with PDF_open().

PDF *PDF_open(FILE *fp, PDF_info *info)
Use the supplied file handle fp to create a new PDF file, using the descriptive elements given in the supplied info block.

void PDF_close(PDF *p)
Must be used when the PDF c onstruction is finished and shall be closed. All internal data structures used for this particular PDF in the library, including the info block, are deallocated.

void PDF_begin_page(PDF *p, float height, float width)
Start a new page in a PDF file. The height and width parameters are the dimensions of the new page in points. Note that there are convenience data structures for most common page formats (see Section 3.9, »Convenience Stuff«).

Restriction:

- Although PDF and PDFlib don't impose any restriction on the usable page size, Acrobat Reader and Exchange suffer from an architectural limit concerning the page size. As of Acrobat 3.01, the page size can range from 1 to 45 inches (72-3240 points). This means that A0 size drawings cannot be used with Acrobat (although they can be generated with PDFlib). Note that other PDF interpreters (such as Ghostscript) may well be able to deal with larger document formats.

void PDF_end_page(PDF *p)
Must be used to finish a page description.

void *PDF_malloc(size_t size, char *caller)
void PDF_free(void *mem)
These functions are used to allocate and free blocks of memory through PDFlib. They resemble the well-known malloc() and free() library calls and may provide additional functionality in the future. The functions allow tracing memory allocation within PDFlib applications. A client need not necessarily use these PDFlib functions but may directly use the system allocation routines instead.

3.3 Error Handling

The PDF_info structure contains a function pointer which refers to a PDFlib error handler with the following signature:

void (*error_handler)(int level, const char* fmt, va_list ap)
PDFlib calls this function pointer with different error levels (information, warning, fatal, internal problem). While informational and warning messages can be ignored, PDFlib is unable to continue if fatal or internal problems occur.

When requesting a new PDF_info block via PDF_get_info, a pointer to PDFlib's default error handler is included in the info block. The default error handler prints a message describing the error on *stdout*. For fatal and internal problems, the error handling also calls exit(). Think of it twice: a library routine exits your program! Since this is probably not what a serious application programmer wants, you are strongly advised to install your own error handler in PDFlib. A custom error handler might, for example,

pop up a dialog box describing the problem and/or handle the cause of the problem in some way appropriate for the application.

3.4 Text Functions

void PDF_show(PDF *p, char *text)
Print text in the current font and font size at the current text position.

void PDF_show_xy(PDF *p, char *text, float x, float y)
Print text in the current font at position (x, y).

void PDF_set_font(PDF *p, char *fontname, float fontsize, PDF_encoding enc)
Set the current font name, font size, and encoding vector. The following names of encoding vectors are supported: `builtin`, `winansi`, and `macroman`.

void PDF_set_leading(PDF *p, float l)
Set the leading (distance between text baselines).

void PDF_set_text_rendering(PDF *p, byte mode)
Set the text rendering mode to one of the following values:

0	*fill text*
1	*stroke text*
2	*fill and stroke text*
3	*invisible text*
4	*fill text and add it to the clipping path*
5	*stroke text and add it to the clipping path*
6	*fill and stroke text and add it to the clipping path*
7	*add text to the clipping path*

void PDF_set_horiz_scaling(PDF *p, float scale)
Set the horizontal text scaling to to a value of `scale` percent.

void PDF_set_text_rise(PDF *p, float rise)
Set the text rise parameter to a value of `rise` units.

void PDF_set_text_matrix(PDF *p, PDF_matrix m)
Set the text matrix which describes a transformation to be applied to the current text font, e.g. for skewing the text.

void PDF_set_text_pos(PDF *p, float x, float y)
Set the current text position to (x, y).

`void PDF_set_char_spacing(PDF *p, float spacing)`
Set the character spacing value, i.e., the horizontal shift of the current point after placing the individual characters in a string. The spacing value is given in text space units. It is reset to zero at the start of a new page. Other than that, the user has to reset the spacing value if so desired.

`void PDF_set_word_spacing(PDF *p, float spacing)`
Set the word spacing value, i.e., the horizontal shift of the current point after individual words in a text line. The spacing value is given in text space units. It is reset to zero at the start of a new page. Other than that, the user has to reset the spacing value if so desired.

`void PDF_continue_text(PDF *p, char *text)`
Move to the next line (determined by the leading parameter) and print text.

`float PDF_stringwidth(PDF *p, byte *text)`
Return the width of the text in the current font in the current coordinate system.

Restrictions:
- A font must be selected before calling this function.
- The font must be one of the 14 core fonts, or a font metrics file (AFM) must be available for the current font (see Section 2.5.1, »Font Embedding and AFM Files«).

3.5 Graphics Functions

3.5.1 Graphics State and Coordinate System

All graphics state parameters are restored to their default values at the beginning of a new page. The default values are documented in Section 2.4.2, »The Graphics States«.

Note that functions related to the text state are listed in Section 3.4, »Text Functions«.

Restriction:
- Don't use graphics state functions within a path description.

`void PDF_save(PDF *p)`
Save the current graphics state.

`void PDF_restore(PDF *p)`
Restore the most recently saved graphics state.

void PDF_translate(PDF *p, float tx, float ty)
Translate the origin of the coordinate system to (tx, ty).

void PDF_scale(PDF *p, float sx, float sy)
Scale the coordinate system by sx and sy.

void PDF_rotate(PDF *p, float phi)
Rotate the coordinate system by phi degrees.

void PDF_setflat(PDF *p, float flat)
Set the flatness to a value between 0 and 100 inclusive.

void PDF_setlinejoin(PDF *p, byte join)
Set the line join parameter to a value between 0 and 2 inclusive.

void PDF_setlinecap(PDF *p, byte cap)
Set the line linecap parameter to a value between 0 and 2 inclusive.

void PDF_setmiterlimit(PDF *p, float miter)
Set the miter limit to a value greater or equal to 1.

void PDF_setlinewidth(PDF *p, float width)
Set the current linewidth to width.

void PDF_setdash(PDF *p, float d1, float d2)
Set the current dash pattern to d1 white and d2 black units. In order to produce a solid line, choose d1 == d2 == 0.

void PDF_setpolydash(PDF *p, float darray, int length)
Set a more complicated dash pattern. The array of the given length contains alternating values for black and white dash lengths. In order to produce a solid line, choose length == 0 or length == 1.

3.5.2 Basic Drawing

void PDF_moveto(PDF *p, float x, float y)
Set the current point to (x, y).

void PDF_lineto(PDF *p, float x, float y)
Draw a line from the current point to (x, y).

void PDF_curveto(PDF *p,
float x1, float y1, float x2, float y2, float x3, float y3)
Draw a Bézier curve from the current point to (x3, y3), using (x1, y1) and (x2, y2) as control points.

void PDF_circle(PDF *p, float x, float y, float r)
Draw a circle with center (x, y) and radius r.

void PDF_arc(PDF *p, float x, float y, float r, float alpha1, float alpha2)
Draw a circular arc with center (x, y), radius r, extending from alpha1 to alpha2 degrees.

void PDF_rect(PDF *p, float x, float y, float width, float height)
Draw a rectangle with lower left corner (x, y) and the supplied width and height.

void PDF_closepath(PDF *p)
Close the current path, i.e. draw a line from the current point to the starting point of the path.

3.5.3 Using the Path

void PDF_stroke(PDF *p)
Stroke (draw) the current path with the current line width and the current stroke color. This operation clears the path.

void PDF_closepath_stroke(PDF *p)
Close the current path and stroke it with the current line width and the current stroke color. This operation clears the path.

void PDF_fill(PDF *p)
Fill the interior of the current path with the current fill color.

void PDF_fill_stroke(PDF *p)
Fill and stroke the path with the current fill and stroke color, respectively.

void PDF_closepath_fill_stroke(PDF *p)
Close the path, fill, and stroke it.

void PDF_endpath(PDF *p)
End the current path.

void PDF_clip(PDF *p)
Use the current path as the clipping path.

3.6 Color Functions

Restriction:

- Don't use color functions within a path description.

void PDF_setgray_fill(PDF *p, float g)
Set the current fill color to a gray value with 0 <= g <= 1.

void PDF_setgray_stroke(PDF *p, float g)
Set the current stroke color to a gray value with 0 <= g <= 1.

void PDF_setgray(PDF *p, float g)
Set the current fill and stroke color to a gray value with 0 <= g <= 1.

void PDF_setrgbcolor_fill(PDF *p, float red, float green, float blue)
Set the current fill color to the supplied RGB values.

void PDF_setrgbcolor_stroke(PDF *p, float red, float green, float blue)
Set the current stroke color to the supplied RGB values.

void PDF_setrgbcolor(PDF *p, float red, float green, float blue)
Set the current fill and stroke color to the supplied RGB values.

3.7 Image Functions

PDF_image *PDF_open_JPEG(PDF *p, char *filename)
PDF_image *PDF_open_TIFF(PDF *p, char *filename)
PDF_image *PDF_open_GIF(PDF *p, char *filename)
Open and analyze a raster graphics file in one of the supported file formats. The returned structure, if not NULL, contains crucial image parameters such as width and height in pixels and the number of colors used. The pointer to this structure must be used to embed the image data in a PDF file.

Restrictions:
- Not all file formats may be supported in a particular PDFlib implementation due to licensing or technical issues.
- Not all flavors of a supported file format may actually work.

void PDF_close_image(PDF *p, PDF_image *image)
void PDF_close_JPEG(PDF *p, PDF_image *image)
void PDF_close_TIFF(PDF *p, PDF_image *image)
void PDF_close_GIF(PDF *p, PDF_image *image)
Close the supplied image file and free the image structure.

void PDF_place_image(
PDF *p, PDF_image *image, float x, float y, float scale)
Place the supplied image (which must have been retrieved with one of the PDF_open_*() functions) on the current page. The lower left corner of the

image is placed at (x, y) on the current page and the image is scaled by the supplied scaling factor.

void PDF_place_inline_image(
PDF *p, PDF_image *image, float x, float y, float scale)
Same as PDF_place_image(), but the image data is placed inline in the PDF file. This should only be used in rare cases.

void PDF_put_image(PDF *p, PDF_image *image)
Put the image data in the PDF without »executing« it, i.e. the image data is only parked in the file for later reference.

void PDF_execute_image(
PDF *p, PDF_image *image, float x, float y, float scale)
»Execute« image data (place at (x, y) and scale) it. The image data must previously have been put into the PDF file with PDF_put_image().

The put/execute technique is useful for re-using image data on several places in a PDF file, e.g. a logo appearing on each page.

The image structure may only be closed after the last call to PDF_execute_image().

void PDF_data_source_from_buf(
PDF *p, PDF_data_source *src, byte *buffer, long len)
Utility routine for converting a memory buffer to a PDFlib data source. This may only be useful for internal and testing purposes.

3.8 Hypertext Functions

In this section, the term »hypertext« is used to denote features which do not directly affect the printed layout, such as bookmarks and page transitions.

void PDF_add_outline(PDF *p, char *text)
Add a PDF bookmark with the supplied text that points to the current page. The text must be encoded with PDFDocEncoding.

Restriction:

- Currently bookmarks cannot be nested.

void PDF_set_transition(PDF *p, PDF_transition t)
Set the page transition effect for the current page. The following transition types are supported:

void PDF_set_duration(PDF *p, float time)
Set the page display duration (in seconds) for self-animated PDFs.

trans_split	*Two lines sweeping across the screen reveal the page.*
trans_blinds	*Multiple lines sweeping across the screen reveal the page.*
trans_box	*A box reveals the page.*
trans_wipe	*A single line sweeping across the screen reveals the page.*
trans_dissolve	*The old page dissolves to reveal the page.*
trans_glitter	*The dissolve effect moves from one screen edge to another.*
trans_replace	*The old page is simply replaced by the new page (default).*

3.9 Convenience Stuff

PDF_pagesize a0, a1, a2, a3, a4, a5, a6, b5, letter, legal, ledger, p11x17;
These structures hold page and width values for the most common page formats which may be used in calls to PDF_begin_page().

3.10 Optimization Techniques

Re-using image data. Although the respective library functions are documented in Section 3.7, »Image Functions«, it might be worth pointing out that PDFlib supports an important PDF optimization technique for using repeated raster images.

Consider a layout with a constant logo or background on several pages. In this situation it is possible to include the image data only once in the PDF and generate only a reference on each of the pages where the graphic is used. Simply open the image file, include the image data with PDF_put_image() in the PDF, and call PDF_execute_image() each time you want to place the logo or background on a particular page. Depending on the image's size and the number of occurences, this technique may prove as a big space saver.

3.11 General Restrictions

Functional restrictions concerning PDFlib output:

- PDFlib-generated files are not optimized (in the sense of linearization necessary for page-at-a-time download).
- PDFlib is currently unable to produce streaming output.
- PDFlib is currently not thread-safe, and therefore shouldn't be used in multi-threading applications.

4 Supplied Library Clients

This section briefly describes the sample library clients supplied with the PDFlib distribution. These clients serve two purposes: they may be useful programs in themselves, and they provide sample code for the application programmer looking forward to using the PDFlib API.

hello [filename]
This is the »Hello, World!« sample discussed in Section 2.1, »A Programming Classic in C«.

If a file name is supplied on the command line, the named file will be generated. Otherwise, the file name `hello.pdf` is used.

pdfdemo [filename]
This is a sample PDFlib application which employs many features of PDFlib such as text and graphics output, embedding, rotating and scaling of several raster images. As well as supplying sample C code, pdfdemo serves as a test bed for the library.

If a fil name is supplied on the command line, the named file will be generated. Otherwise, the file name `demo.pdf` is used.

pdfclock [-c pagecount] -o filename
Generate a PDF with an analog clock face showing the current time. `filename` is the name of the PDF file to be generated. If supplied on the command line, `pagecount` specifies the number of pages to be generated. Pages are generated with a delay of approximately one second. Since the PDF output specifies a page advance delay with a wipe page transition, the output somehow resembles a real analog clock. Try it!

pdfgraph -b -o filename datafile
An interpreter for a mini-language describing vector-oriented graphics. Simple lines and color settings can be used to generate nice little demos or real graphs. `filename` is the name of the PDF file to be generated, `datafile` contains the drawing instructions. The `-b` option specifies binary PDF output.

```
text2pdf [options] [textfile]
-b            binary mode (default: ASCII)
-f fontname   name of font to use
-h height     page height in points
-m margin     margin size in points
-s size       font size
-o filename   PDF output file name
-w width      page width in points
```

```
-I path       path to AFM and font directory
```

Convert the text in the supplied text file to PDF. Several aspects of the generated PDF can be controlled via command line options.

```
imagepdf [options] imagefile(s)
-a          ASCII mode (default: binary)
-c          print caption
-o <file>   output file
```

Convert one or more graphics file(s) to PDF. Each file is placed on a separate page of appropriate size. For easy reference a bookmark with the file name points to each page.

Note that the range of supported graphics file formats depends on your particular PDFlib installation. Supported formats may include JPEG, GIF, and TIFF.

5 Environment Bindings

PDFlib may be used in different environments, e.g., language bindings, Web integration techniques, scripting languages. Several bindings are supported in the distributed PDFlib package. Some hints related to these bindings are given in this section.

5.1 C and C++ Language Bindings

This is probably the most familiar binding to most programmers. Simply use this manual to fabricate PDFlib C API calls from within your plain old C program. Using some precaution, PDFlib may also be used from within C++.

5.2 Perl, Tcl, and Python Bindings

Based on a cute facility called SWIG[1] (Simplified Wrapper and Interface Generator) written by Dave Beazley <beazley@cs.utah.edu>, PDFlib can easily be integrated into the Perl, Tcl, and Python scripting languages. This means you can call PDFlib API functions without any C programming by simply writing a couple of script language instructions. PDFlib scripting greatly simplifies small to medium programming tasks and is appropriate in many application areas where the development, build, and debug overhead of C is considered too high.

Note that you don't need to install SWIG in order to make use of PDFlib scripting. All necessary files are contained within the PDFlib distribution. SWIG support for PDFlib was initiated and in its basic parts implemented by Rainer Schaaf <Rainer.Schaaf@t-online.de> – thanks, Rainer[2]!

Implementing Scripting Support. Scripting support is implemented through an interface definition file (supported with PDFlib), shared libraries, and some helper files such as a module definition for Perl.

Restriction:

- Currently the activation of scripting support in PDFlib is not as much automated as might be desirable.
- Scripting support may not be available on all platforms.
- There is currently no support for exception handling in PDFlib scripting.

PDFlib Script Programming. In order to avoid duplicating the PDFlib API reference manual for all supported scripting languages, this manual is considered authoritative not only for the C binding but also for the scripting

1. More information on SWIG can be found at http://www.cs.utah.edu/~beazley/SWIG/swig.html
2. On a totally unrelated note, Rainer and his wonderful family live in a nice house close to the Alps – definitely a great place for biking!

languages. Of course, the script programmer has to mentally adapt certain conventions and syntactical issues from C to the relevant scripting language. However, translating C API calls to, say, Perl calls should be a straightforward process. Indeed, I was able to translate a C PDFlib application to Perl by simply deleting the include directives and adding dollar signs to all variable names!

The »Hello world!« and PDFclock examples are available in the distribution as Perl, Tcl, and Python versions.

5.3 Common Gateway Interface (CGI)

Due to the general nature of CGI, no special action must be taken in order to use PDFlib within a CGI environment. The BINDINGS/CGI subdirectory contains a very small example for integrating PDFlib programs in a Web server environment. The MIME type for PDF files is as follows:

```
application/pdf
```

The `clock.cgi` sample generates a temporary PDF file using the pdfclock demo client and sends the PDF data to the Web browser using the CGI interface. After sending the data the temporary file is deleted:

```
#!/bin/sh
TMP=/var/tmp/clock.$$.pdf

echo Content-type: application/pdf
echo

pdfclock -o $TMP
cat $TMP
rm $TMP
exit 0
```

Of course, the necessary HTTP header(s) may also be generated directly by a PDFlib application. The above CGI shell script is mainly for instructional purposes.

5.4 Functional Programming

If you are interested in functional programming, you may want to take a look at fun->pdf: Functional Writers of Portable Documents. Jan Skibinski implemented an interface to PDFlib for several functional programming languages:

- Equational Functional Language Q
- Ocaml
- Haskell

For more information on fun->pdf, check out

```
http://www.numeric-quest.com/funpdf
```

5.5 Future Bindings

Several other language and environment bindings are under consideration for future PDFlib versions:

- Windows DLL
- ISAPI module for Microsoft Internet Information Server

6 The PDFlib License

PDFlib is subject to the »ALADDIN FREE PUBLIC LICENSE«.[1] The complete text of the license agreement can be found in the file LICENSE. In short and non-legal terms:

- You may use and distribute PDFlib non-commercially.
- You may develop free software with PDFlib.
- You may NOT sell any software based on PDFlib without prior written permission of the author.
- If you write PDFlib programs and sell them, you need a commercial PDFlib license.

Note that this is only a 10-second-description which is not legally binding. Only the text in the LICENSE file is considered to completely describe the licensing conditions. Please contact the author for details on using PDFlib commercially:

Consulting ℮ Publishing
Thomas Merz
Tal 40
80331 München, Germany

http://www.ifconnection.de/~tm
tm@muc.de
fax +49/89/29 16 46 86

1. The license text was devised by L. Peter Deutsch of Aladdin Enterprises (Menlo Park, CA). Thanks Peter for making available the text!

7 References

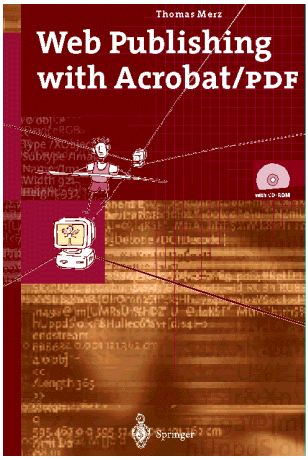
Although this manual is intended to be self-contained with respect to PDFlib programming, it is highly recommended to obtain a copy of the PDF specification for a deeper understanding and more detailed information:

[1] Portable Document Format Reference Manual, Version 1.2
Available from

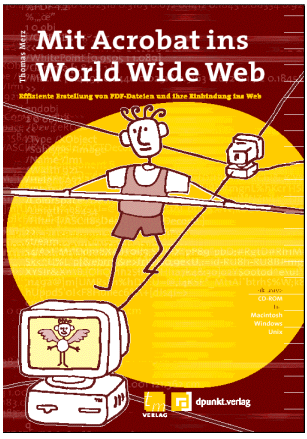
`http://www.adobe.com/supportservice/devrelations/PDFs/TN/PDFSPEC.PDF`

[2] The following book by the author of PDFlib is available in English and German editions. It describes all aspects of integrating Acrobat in the WWW:

English edition:
Thomas Merz:
Web Publishing with Acrobat/PDF.
With CD-ROM.
Springer-Verlag Heidelberg Berlin New York
1998
ISBN 3-540-63762-1
orders@springer.de



German edition:
Thomas Merz:
Mit Acrobat ins World Wide Web.
Effiziente Erstellung von PDF-Dateien und
ihre Einbindung ins Web.
Mit CD-ROM.
ISBN 3-9804943-1-4
Thomas Merz Verlag 1998
80331 München, Tal 40
Fax +49/89/29 16 46 86
http://www.ifconnection.de/~tm



8 Revision History

Version information on PDFlib itself can be found in the distribution. This chapter documents changes to the PDFlib Reference Manual (this document).

22 September 1997

- First public release of PDFlib version 0.4 and this manual.

25 February 1998

- Slightly expanded the manual to cover PDFlib version 0.5.

08 July 1998

- First attempt at describing PDFlib scripting support in PDFlib 0.6.

PostScript & Acrobat/PDF

Applications, Troubleshooting, and Cross-Platform Publishing

Originally entitled the »PostScript and Acrobat Bible« in German, this handbook achieves the seemingly impossible: it covers this tricky and technical field in an entertaining manner without getting bogged down in PostScript programming. The author genuinely wants to assist in overcoming cross-platform barriers using MS-DOS, Windows, Macintosh or Unix and, accordingly, neither the book nor the tools are limited to one particular platform or operating system. The 9 chapters plus 3 appendixes run the entire gamut, from the very basics right up to Ghostscript. The whole book is creatively designed, making use of comical illustrations. In short, essential reading for all technically minded users of PostScript and Acrobat/PDF.

Examples

- How to port EPS files or fonts from Mac to Windows to Unix
- Pros and cons of different PostScript drivers for Windows
- How to install PostScript fonts in the X Window System
- How to interpret and fix PostScript error messages
- How to edit or create PostScript fonts
- How to make EPS files editable again
- What's the Control-D business with PostScript files?
- How to make use of Level 2 without a Level 2 savvy driver
- How to create hypertext features in PDF files automatically
- How to use PDF files without Acrobat software
- Linking PDF files to the World Wide Web
- Performance optimization and prepress issues

Contents

Basics – Between Monitor and Printer – Encapsulated PostScript (EPS) – PostScript Fonts – PostScript Level 2 – Gray Levels and Color – Display PostScript – Adobe Acrobat and PDF – Miscellaneous – Software on the CD-ROM – Ghostscript Manual – Character Sets

PostScript & Acrobat/PDF
Applications, Troubleshooting, and Cross-Platform Publishing

By Thomas Merz. 420 pages including CD-ROM for MS-DOS/Windows/Macintosh/Unix
ISBN 3-540-60854-0, Springer-Verlag, Heidelberg, Berlin, New York
Springer-Verlag New York Inc., 175 Fifth Avenue, New York, NY 10010, U.S.A. Email: orders@springer.de

German Edition: Die PostScript- & Acrobat-Bibel
Was Sie schon immer über PostScript und Acrobat/PDF wissen wollten

Von Thomas Merz. 444 Seiten zweifarbig, 137 Abbildungen, Hardcover; DM 89,-
CD-ROM für MS-DOS/Windows/Macintosh/Unix beiliegend
ISBN 3-9804943-0-6, Thomas Merz Verlag, München, Fax +49/89/29 16 46 86

Index

R

S

T

W