PDFlib GmbH München, Germany

www.pdflib.com

PDFlib GmbH
Tal 40, 80331 München, Germany
http://www.pdflib.com

phone +49 • 89 • 29 16 46 87
fax +49 • 89 • 29 16 46 86

If you have questions check the PDFlib mailing list and archive at http://groups.yahoo.com/group/pdflib

Licensing contact: sales@pdflib.com
Support for commercial PDFlib licensees: support@pdflib.com (please include your license number)

Author: Thomas Merz
Design and illustrations: Alessio Leonardi
Quality control (manual): Katja Karsunke, Rainer Schaaf, Kurt Stützer
Quality control (software): a cast of thousands

Contents

o Applying the PDFlib License Key

All binary PDFlib and PDI versions supplied by PDFlib GmbH can be used as fully functional evaluation versions regardless of whether or not you obtained a commercial license. However, unlicensed versions will display a *www.pdflib.com* demo stamp (the »nagger«) cross all generated pages. Companies which are seriously interested in PDFlib licensing and wish to get rid of the nagger during the evaluation phase or for prototype demos can submit their company and project details to *sales@pdflib.com,* and request a temporary license key. Once you purchased a PDFlib or PDI license key you must apply it in order to get rid of the demo stamp. There are several methods available:

- Add a line to your script or program which sets the license key at runtime:

  ```
  PDF_set_parameter(p, "license", "...your license key...");
  ```

  The *license* parameter must be set only once, immediately after instantiating the PDFlib object (i.e., after *PDF_new( )* or equivalent call).
- Enter the license key in a text file according to the following format:

  ```
  PDFlib license file 1.0
  # Licensing information for PDFlib GmbH products
  PDFlib   5.0.2   ...your license key...
  ```

  The license file may contain license keys for multiple PDFlib GmbH products on separate lines. Next, you must inform PDFlib about the license file, either by setting the *licensefile* parameter immediately after instantiating the PDFlib object (i.e., after *PDF_new( )* or equivalent call) as follows:

  ```
  PDF_set_parameter(p, "licensefile", "/path/to/license/file");
  ```

  or by setting the environment variable *PDFLIBLICENSEFILE* with a command similar to the following:

  ```
  export PDFLIBLICENSEFILE=/path/to/license/file
  ```

Note that PDFlib, PDFlib+PDI, and PDFlib Personalization Server (PPS) are different products which require different license keys although they are delivered in a single package. PDFlib+PDI license keys will also be valid for PDFlib, but not vice versa, and PPS license keys will be valid for PDFlib+PDI and PDFlib. All license keys are platform-dependent, and can only be used on the platform for which they have been purchased.

Accumulating individual CPU keys. If you purchased multiple CPU licenses, but didn't obtain these with a single order, but with multiple 1-CPU orders, you can accumulate all keys in the license file by entering all at the same time. The other methods for applying license keys do not support CPU accumulation.

Evaluating features which are not yet licensed. You can fully evaluate all feature by using the software without any license key applied. However, once you applied a valid license key for a particular product using features of a higher category will no longer be available. For example, if you installed a valid PDFlib license key the PDI functionality will no longer be available for testing. Similarly, after installing a PDFlib+PDI license key the personalization features (block functions) will no longer be available.

When a license key for a product has already been installed set a 0 dummy license key to enable functionality of a higher product class for evaluation:

```
PDF_set_parameter(p, "license", "0");
```

This will enable the previously disabled functions, and re-activate the demo stamp across all pages.

Licensing options. Different licensing options are available for PDFlib use on one or more servers, and for redistributing PDFlib with your own products. We also offer support and source code contracts. Licensing details and the PDFlib purchase order form can be found in the PDFlib distribution. Please contact us if you are interested in obtaining a commercial PDFlib license, or have any questions:

PDFlib GmbH, Licensing Department
Tal 40, 80331 München, Germany
http://www.pdflib.com
phone +49 • 89 • 29 16 46 87, fax +49 • 89 • 29 16 46 86
Licensing contact: *sales@pdflib.com*
Support for PDFlib licensees: *support@pdflib.com*

1 Introduction

1.1 PDFlib Programming

What is PDFlib? PDFlib is a library which allows you to generate files in Adobe's Portable Document Format (PDF). PDFlib acts as a backend to your own programs. While you (the programmer) are responsible for retrieving 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 internal details of PDF. Our binary packages contain different products in a single library:

- PDFlib offers many useful functions for creating text, graphics, images, and hypertext elements in PDF.
- The optional add-on PDF Import Library (PDI) can be used to integrate pages from existing PDF documents into the generated output.
- The PDFlib Personalization Server (PPS) offers powerful block processing functions, and includes the PDFlib Block plugin for Adobe Acrobat.

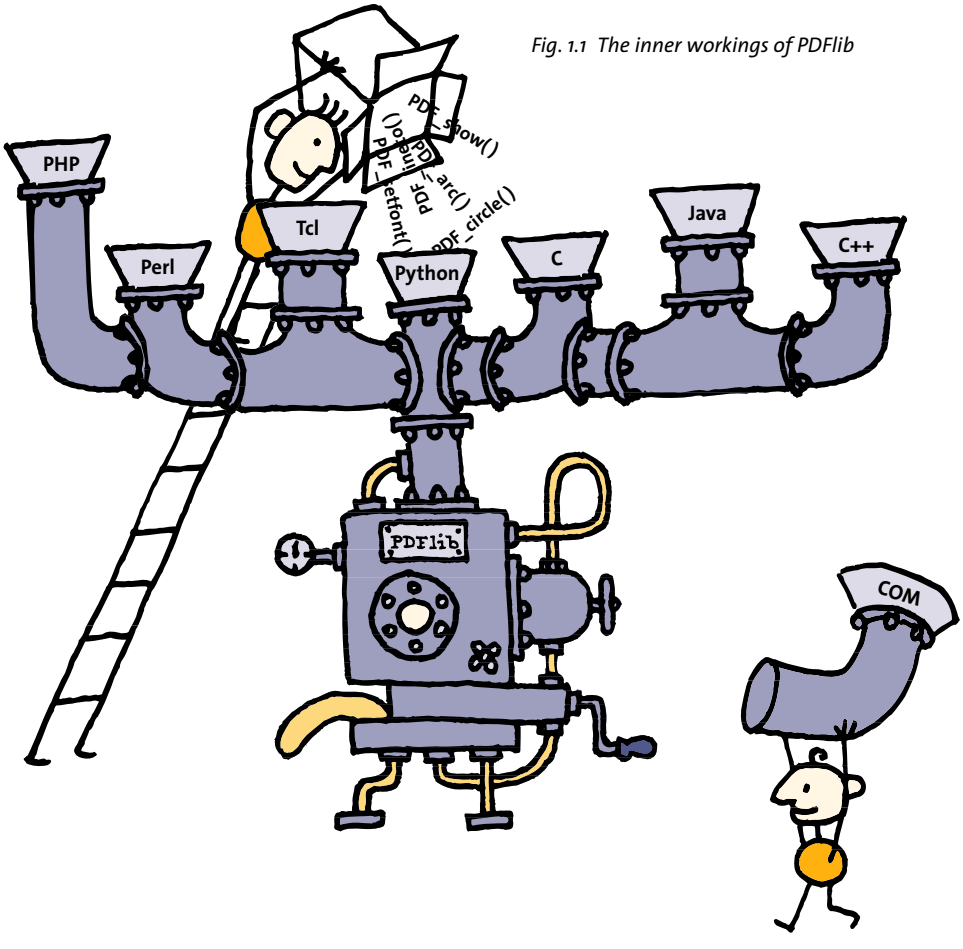


Fig. 1.1 The inner workings of PDFlib

How can I use PDFlib? PDFlib is available on a variety of platforms, including Unix, Windows, Mac OS, and EBCDIC-based systems such as IBM eServer iSeries and zSeries. PDFlib itself is written in the C language, but it can be also accessed from several other languages and programming environments which are called language bindings. These language bindings cover all major Web and stand-alone application languages currently in use. The Application Programming Interface (API) is easy to learn, and is identical for all bindings. Currently the following bindings are supported:

- COM for use with Visual Basic, Active Server Pages with VBScript or JScript, Borland Delphi, Windows Script Host, and other environments
- ANSI C
- ANSI C++
- Cobol (IBM eServer zSeries)
- Java, including servlets
- .NET for use with C#, VB.NET, ASP.NET, and other environments
- PHP hypertext processor
- Perl
- Python
- RPG (IBM eServer iSeries)
- Tcl

What can I use PDFlib for? PDFlib's primary target is creating dynamic PDF within your own software, or on the World Wide Web. Similar to HTML pages dynamically generated on the Web server, you can 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:

- PDFlib can be integrated directly in the application generating the data, eliminating the convoluted creation path application–PostScript–Acrobat Distiller–PDF.
- As an implication of this straightforward process, PDFlib is the fastest PDF-generating method, making it perfectly suited for the Web.
- PDFlib's thread-safety as well as its robust memory and error handling support the implementation of high-performance server applications.
- PDFlib is available for a variety of operating systems and development environments.

Requirements for using PDFlib. PDFlib makes PDF generation possible without wading through the PDF specification. While PDFlib tries to hide technical PDF details from the user, a general understanding of PDF is useful. In order to make the best use of PDFlib, application programmers should ideally be familiar with the basic 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 shouldn't have much trouble adapting to the PDFlib API as described in this manual.

About this manual. This manual describes the API implemented in PDFlib. It does not describe the process of building the library binaries. Functions not described in this manual are unsupported, and should not be used. This manual does not attempt to explain Acrobat features. Please refer to the Acrobat product literature, and the material cited at the end of this manual for further reference. The PDFlib distribution contains additional examples for calling PDFlib functions.

1.2 PDFlib Features

Table 1.1 lists the major PDFlib API features for generating and importing PDF.

Table 1.1 Features of PDFlib, PDFlib+PDI, and the PDFlib Personalization Server (PPS)

topic	V 5.0	features
PDF output		PDF documents of arbitrary length, directly in memory (for Web servers) or on disk file
		arbitrary page size–each page may have a different size
		compression for text, vector graphics, image data, and file attachments
		compatibility modes for PDF 1.3, 1.4, and 1.5 (Acrobat 4, 5, and 6)
PDF input		import pages from existing PDF documents (PDFlib+PDI only)
	X	flexible PDF page placement and formatting
Blocks	X	PDF personalization with PDFlib blocks for text, image, and PDF data (PPS only)
	X	PDFlib Block plugin for Acrobat to create PDFlib blocks
Graphics		common vector graphics primitives: lines, curves, arcs, rectangles, etc.
		vector paths for stroking, filling, and clipping
	X	smooth shadings (color blends)
		pattern fills and strokes
		efficiently re-use text or vector graphics with templates
	X	explicit graphics state parameters for text knockout, overprinting etc.
	X	transparency (opacity) and blend modes
Color		grayscale, RGB, and CMYK color
	X	CIE L*a*b* color
	X	ICC-based color with standard ICC color profiles
	X	built-in PANTONE® and HKS® spot color tables
		user-defined spot colors
	X	default gray, RGB, and CMYK color spaces to remap device-dependent colors
	X	rendering intent for text, graphics, and raster images
Prepress	X	generate output conforming to PDF/X-1, PDF/X-1a, and PDF/X-3
	X	embed output intent ICC profile or reference standard output intent
	X	copy output intent from imported PDF documents (PDFlib+PDI only)
Fonts		text output in different fonts; underlined, overlined, and strikeout text
	X	text column formatting and text line positioning options
	X	built-in metrics and kerning information for all glyphs in the PDF core fonts
		font embedding
		fonts can be pulled from the Windows or Mac host system
	X	TrueType (ttf and ttc) and PostScript Type 1 fonts (pfb and pfa, plus lwfn on the Mac)
		OpenType fonts (ttf, otf) with PostScript or TrueType outlines
		AFM and PFM PostScript font metrics files
	X	kerning for PostScript, TrueType, and OpenType fonts
	X	subsetting for TrueType and OpenType fonts
	X	user-defined (Type 3) fonts
	X	TrueType and OpenType glyph id addressing for advanced typesetting applications
	X	proportional widths for standard CJK fonts
		retrieve character metrics for exact formatting

Table 1.1 Features of PDFlib, PDFlib+PDI, and the PDFlib Personalization Server (PPS)

topic	V 5.0	features
		fetch code pages from the system (Windows, IBM eServer iSeries and zSeries)
Internationalization and Unicode		support for a variety of encodings (both built-in and user-defined)
		standard CJK font and CMap support for Chinese, Japanese, and Korean text
	X	custom CJK fonts in the TrueType and OpenType formats with Unicode encoding
		Euro character (subject to availability of the Euro glyph in the used font)
		built-in international standards and vendor-specific code pages
	X	Unicode for page descriptions (UTF-8 and UCS-2 formats, little- and big-endian)
		Unicode for hypertext features
	X	embed Unicode information in PDF for proper text extraction in Acrobat
Security	X	generate output with 40-bit or 128-bit encryption
	X	generate output with permission settings
	X	import encrypted documents (master password required; PDI only)
Hypertext		page transition effects such as shades and mosaic
		nested bookmarks
		PDF links, launch links (other document types), and Web links
		document information: four standard fields (Title, Subject, Author, Keywords) plus unlimited number of user-defined info fields
		file attachments and note annotations
	X	named destinations for links, bookmarks, and document open action
	X	viewer preferences (hide menu bar, etc.)
	X	more bookmark targets
	X	user coordinates (instead of default coordinates) for hypertext elements
Images	X	embed BMP, GIF (non-interlaced), PNG, TIFF, JPEG, and CCITT raster images
	X	automatic detection of image file formats (file format sniffing)
	X	use all kinds of image data from file or from memory
		efficiently re-use image data, e.g., for repeated logos on each page
	X	transparent (masked) images including soft masks
		image masks (transparent images with a color applied)
		colorize images with a spot color
	X	flexible image placement and formatting
	X	honor embedded ICC profiles in JPEG, TIFF, and PNG
	X	apply an external ICC profile to an image
	X	image interpolation (smooth images with low resolution)
Programming		language bindings for Cobol, COM, C, C++, Java, .NET, Perl, PHP, Python, RPG, Tcl
		thread-safe for deployment in multi-threaded server applications
		configurable error handler and memory management for C and C++
		exception handling integrated with the host language's native exception handling
	X	virtual file system for supplying data in memory, e.g., images, fonts, ICC profiles

1.3 Availability of Features in different Products

Table 1.2 details the availability of features in the open source edition of PDFlib and various commercial products.

Table 1.2 Availability of features in different products

Feature	API functions and parameters	PDFlib Lite (open source)	PDFlib	PDFlib+PDI	PDFlib Personalization Server (PPS)
basic PDF generation	(all except those listed below)	X	X	X	X
COM and .NET language bindings		–	X	X	X
works on EBCDIC-based systems		–	X	X	X
encryption (password protection and permission settings)	userpassword, masterpassword, permissions	–	X	X	X
font subsetting	PDF_load_font( ) with subsetting option	–	X	X	X
kerning	PDF_load_font( ) with kerning option	–	X	X	X
access Mac and Windows host fonts	PDF_load_font( )	–	X	X	X
access system encodings on Windows, iSeries, and zSeries	PDF_load_font( )	–	X	X	X
Unicode encoding and ToUnicode CMaps for PS, TT and OT fonts	PDF_load_font( ) with encoding = unicode, autocidfont, unicodemap	–	X	X	X
proportional glyph widths for standard CJK fonts with UCS2 CMaps	PDF_load_font( ) with a UCS2-compatible CMap	–	X	X	X
glyph ID addressing	PDF_load_font( ) with encoding = glyphid	–	X	X	X
extended encoding support for PostScript-based OpenType fonts	PDF_load_font( )	–	X	X	X
spot color	PDF_makespotcolor( )	–	X	X	X
PDF/X support	PDF_process_pdi( ), pdfx	–	X	X	X
ICC profile support	PDF_load_iccprofile( ), PDF_setcolor( ) with iccbasedgray/rgb/cmyk, PDF_load_image( ) with honoriccprofile option, honoriccprofile	–	X	X	X
CIE L*a*b* color	PDF_setcolor( ) with type = lab	–	X	X	X
default color spaces	defaultgray/rgb/cmyk	–	X	X	X
PDF import (PDI)	PDF_open_pdi( ), PDF_open_pdi_callback( ), PDF_open_pdi_page( ), PDF_fit_pdi_page( ), PDF_process_pdi( ), PDF_get_pdi_value( ), PDF_get_pdi_parameter( )	–	–	X	X
variable data processing and personalization with blocks	PDF_fill_textblock( ), PDF_fill_imageblock( ), PDF_fill_pdfblock( )	–	–	–	X
query standard and custom block properties	PDF_get_pdi_value( ), PDF_get_pdi_parameter( ) with vdp/Blocks keys	–	–	–	X
PDFlib Block plugin for Acrobat	interactively create PDFlib blocks	–	–	–	X

1.4 Acrobat Versions and PDFlib Features

At the user's option PDFlib generates output according to PDF 1.3 (Acrobat 4), PDF 1.4 (Acrobat 5), or PDF 1.5 (Acrobat 6). In PDF 1.3 compatibility mode the PDFlib features listed in Table 1.3 will not be available. Trying to use one of these features in PDF 1.3 mode will result in an exception.

Table 1.3 PDFlib features which are not available in PDF 1.3 compatibility mode

Feature	***PDFlib API functions and parameters***
smooth shadings (color blends)	*PDF_shading_pattern( ), PDF_shfill( ), PDF_shading( )*
soft masks	*PDF_load_image( ) with the masked option referring to an image with more than 1 bit pixel depth*
128-bit encryption	*userpassword, masterpassword, permissions*
extended permission settings (see Table 3.12)	*permissions*
certain CMaps for CJK fonts (see Table 4.6)	*PDF_load_font( )*
certain settings in explicit graphics states, mostly related to transparency	*PDF_create_gstate( ) with options alphaisshape, blendmode, opacityfill, opacitystroke, textknockout*

2 PDFlib Language Bindings

2.1 Overview

While the C programming language has been one of the cornerstones of systems and applications software development for decades, a whole slew of other languages have been around for quite some time which are either related to new programming paradigms (such as C++), open the door to powerful platform-independent scripting capabilities (such as Perl, Tcl, and Python), promise a new quality in software portability (such as Java), or provide the glue among many different technologies while being practically platform-specific (such as COM).

Naturally, the question arises how to support so many languages with a single library. Fortunately, all modern language environments are extensible in some way or another.

Availability and platforms. All PDFlib features are available on all platforms and in all language bindings (with a few minor exceptions which are noted in the manual). Table 2.1 lists the language/platform combinations we used for testing.

Table 2.1 Tested language and platform combinations

language	*Unix (Linux, Solaris, HP-UX, Mac OS X, AIX, IRIX a.o.)*	*Windows*	*IBM eServer iSeries and zSeries*
Cobol	–	–	*ILE Cobol*
COM	–	*ASP (PWS, IIS 4, 5, 6)* *WSH (VBScript 5, JScript 5)* *Visual Basic 6.0, Borland Delphi 5 – 7*	–
ANSI C	*gcc 2/3, HP C, IBM C 6, Sun Workshop 6, and other ANSI C compilers*	*Microsoft Visual C++ 6, 7* *Metrowerks CodeWarrior 7, 8* *Borland C++ Builder 5*	*IBM c89* *SAS C for MVS*
ANSI C++	*gcc 2/3 and other ANSI C++ compilers*	*Microsoft Visual C++ 6, 7* *Metrowerks CodeWarrior 7, 8*	*IBM c89*
Java	*JDK 1.1.8, 1.2.2, 1.3, 1.4*	*Sun JDK 1.1.8, 1.2.2, 1.3, 1.4* *ColdFusion MX*	*JDK 1.3.1*
.NET	–	*.NET Framework 1.0, 1.1:* *C#, VB.NET, ASP.NET*	–
Perl	*Perl 5.6 – 5.8*	*Perl 5.6 – 5.8*	–
PHP	*PHP 4.1.0 – 4.3.5*	*PHP 4.1.0 – 4.3.5*	–
Python	*Python 1.6, 2.0 – 2.3*	*Python 1.6, 2.0 – 2.3*	–
RPG	–	–	*ILE RPG*
Tcl	*Tcl 8.3.2 and 8.4.4*	*Tcl 8.3.2 and 8.4.4*	–

PDFlib on embedded systems. It shall be noted that PDFlib can also be used on embedded systems, and has been ported to the Windows CE, QNX, and EPOC environments as well as custom embedded systems. For use with restricted environments certain features are configurable in order to reduce PDFlib's resource requirements. If you are interested in details please contact us via *sales@pdflib.com*.

2.2 Cobol Binding

2.2.1 Special Considerations for Cobol

The PDFlib API functions for Cobol are not available under the names documented in Chapter 7, but use abbreviated function names instead. The short function names are not documented here, but can be found in a separate cross-reference listing *(xref.txt)*. For example, instead of using *PDF_load_font( )* the short form *PDLODFNT* must be used.

PDFlib clients written in Cobol are statically linked to the PDFLBCOB object. It in turn dynamically loads the PDLBDLCB Load Module (DLL), which in turn dynamically loads the PDFlib Load Module (DLL) upon the first call to PDNEW (which corresponds to *PDF_new( )*). The instance handle of the newly allocated PDFlib internal structure is stored in the *P* parameter which must be provided to each call that follows.

The PDLBDLCB load module provides the interfaces between the 8-character Cobol functions and the core PDFlib routines. It also provides the mapping between PDFlib's asynchronous exception handling and the monolithic »check each function's return code« method that Cobol expects.

Note PDLBDLCB and PDFLIB must be made available to the COBOL program through the use of a STEPLIB.

Data types. The data types used in the PDFlib API reference must be mapped to Cobol data types as in the following samples (taken from the *hello* example below):

```
05  PDFLIB-A4-WIDTH         USAGE COMP-1 VALUE 5.95E+2. // float
05  WS-INT                  PIC S9(9) BINARY.           // int
05  WS-FLOAT                COMP-1.                     // float
05  WS-STRING               PIC X(128).                 // const char *
05  P                       PIC S9(9) BINARY.           // long *
05  RETURN-RC               PIC S9(9) BINARY.           // int *
```

All Cobol strings passed to the PDFlib API should be defined with one extra byte of storage for the expected LOW-VALUES (NULL) terminator.

Return values. The return value of PDFlib API functions will be supplied in an additional *ret* parameter which is passed by reference. It will be filled with the result of the respective function call. A zero return value means the function call executed just fine; other values signal an error, and PDF generation cannot be continued.

Functions which do not return any result (C functions with a void return type) don't use this additional parameter.

Error handling. PDFlib exception handling is not available in the Cobol language binding. Instead, all API functions support an additional return code *(rc)* parameter which signals errors. The *rc* parameter is passed by reference, and will be used to report problems. A non-zero value indicates that the function call failed.

2.2.2 The »Hello world« Example in Cobol

The following example shows a simple Cobol program which links against PDFlib. Note that it does not do any error checking:

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.     HELLO.

 ENVIRONMENT DIVISION.

 DATA DIVISION.
 WORKING-STORAGE SECTION.

 01  PDFLIB-PAGE-SIZE-CONSTANTS.
     05  PDFLIB-A4-WIDTH         USAGE COMP-1 VALUE 5.95E+2.
     05  PDFLIB-A4-HEIGHT        USAGE COMP-1 VALUE 8.42E+2.

 01  PDFLIB-CALL-AREA.
     05  P                       PIC S9(9) BINARY.
     05  RC                      PIC S9(9) BINARY.
     05  PDFLIB-RETURN-LONG      PIC S9(9) BINARY.
     05  PDFLIB-RETURN-CHAR      PIC X(64) VALUE SPACES.

 01  WS-WORK-FIELDS.
     05  WS-INT                  PIC S9(9) BINARY.
     05  WS-FONT                 PIC S9(9) BINARY.
     05  WS-FLOAT                COMP-1.
     05  WS-FLOAT2               COMP-1.
     05  WS-STRING               PIC X(128).
     05  WS-STRING2              PIC X(128).
     05  WS-PDF-ERR-STRING       PIC X(128).
     05  WS-NULL                 PIC X(1) VALUE LOW-VALUES.

 PROCEDURE DIVISION.

*   CREATE A PDF OBJECT
     CALL "PDNEW" USING          P,
                                 RC.

*   OPEN PDF FILE
     STRING 'HELLO.PDF' LOW-VALUES
            DELIMITED BY SIZE INTO WS-STRING.
     CALL "PDOPNFIL" USING       P,
                                 WS-STRING,
                                 PDFLIB-RETURN-LONG,
                                 RC.

     IF PDFLIB-RETURN-LONG = -1
         CALL "PDERRMSG" USING   P,
                                 WS-PDF-ERR-STRING,
                                 RC.

         DISPLAY WS-PDF-ERR-STRING.

         MOVE +8 TO RETURN-CODE
         GOBACK.

*   SET PDF INFORMATION
     STRING 'Creator' LOW-VALUES
            DELIMITED BY SIZE INTO WS-STRING.
     STRING 'Hello.cbl' LOW-VALUES
            DELIMITED BY SIZE INTO WS-STRING2.
     CALL "PDSETINF" USING       P,
```

```
                                WS-STRING,
                                WS-STRING2,
                                RC.

     STRING 'Author' LOW-VALUES
            DELIMITED BY SIZE INTO WS-STRING.
     STRING 'Thomas Merz' LOW-VALUES
            DELIMITED BY SIZE INTO WS-STRING2.
     CALL "PDSETINF" USING      P,
                                WS-STRING
                                WS-STRING2,
                                RC.

     STRING 'Title' LOW-VALUES
            DELIMITED BY SIZE INTO WS-STRING.
     STRING 'Hello, world (COBOL)!' LOW-VALUES
            DELIMITED BY SIZE INTO WS-STRING2.
     CALL "PDSETINF" USING      P,
                                WS-STRING
                                WS-STRING2,
                                RC.

*    START A NEW PAGE
     CALL "PDBGNPAG" USING      P,
                                PDFLIB-A4-WIDTH,
                                PDFLIB-A4-HEIGHT,
                                RC.

*    FONT PROCESSING
     MOVE 0 TO WS-INT.
     STRING 'Helvetica-Bold' LOW-VALUES
            DELIMITED BY SIZE INTO WS-STRING.
     STRING 'host' LOW-VALUES
            DELIMITED BY SIZE INTO WS-STRING2.
     CALL "PDLODFNT" USING      P,
                                WS-STRING
                                WS-INT,
                                WS-STRING2,
                                WS-NULL,
                                PDFLIB-RETURN-LONG,
                                RC.

     MOVE PDFLIB-RETURN-LONG    TO  WS-FONT.
     MOVE 24 TO WS-FLOAT.
     CALL "PDSETFNT" USING      P,
                                WS-FONT,
                                WS-FLOAT,
                                RC.

*    WRITE TO THE CURRENT PAGE OF THE PDF DOCUMENT
     MOVE 50 TO WS-FLOAT.
     MOVE 700 TO WS-FLOAT2.
     CALL "PDSETTP" USING       P,
                                WS-FLOAT,
                                WS-FLOAT2,
                                RC.

     STRING 'Hello, World!' LOW-VALUES
```

```
               DELIMITED BY SIZE INTO WS-STRING.
      CALL "PDSHOW" USING         P,
                                  WS-STRING,
                                  RC.

      STRING '(says COBOL)' LOW-VALUES
               DELIMITED BY SIZE INTO WS-STRING.
      CALL "PDCONT" USING         P,
                                  WS-STRING,
                                  RC.

*     END THE PAGE
      CALL "PDENDPAG" USING       P,
                                  RC.

*     CLOSE THE PDF DOCUMENT
      CALL "PDCLOSE" USING        P,
                                  RC.

*     DELETE THE PDF OBJECT
      CALL "PDDELETE" USING       P,
                                  RC.

*     END THE PROGRAM
      MOVE ZERO                   TO RETURN-CODE.
      GOBACK.

  END PROGRAM HELLO.
```

2.3 COM Binding

(This section is not included in this edition of the PDFlib manual.)

2.4 C Binding

2.4.1 Availability and Special Considerations for C

PDFlib itself is written in the ANSI C language. In order to use the PDFlib C binding, you can use a static or shared library (DLL on Windows and MVS), and you need the central PDFlib include file *pdflib.h* for inclusion in your PDFlib client source modules. Alternatively, *pdflibdl.h* can be used for dynamically loading the PDFlib DLL at runtime (see Section 2.4.3, »Using PDFlib as a DLL loaded at Runtime«, page 22).

2.4.2 The »Hello world« Example in C

The following example shows a simple C program which links against a static or shared/dynamic PDFlib library:

```
#include <stdio.h>
#include <stdlib.h>

#include "pdflib.h"

int
main(void)
```

```
{
    PDF *p;
    int font;

    /* create a new PDFlib object */
    if ((p = PDF_new()) == (PDF *) 0)
    {
        printf("Couldn't create PDFlib object (out of memory)!\n");
        return(2);
    }

    PDF_TRY(p) {
        /* open new PDF file */
        if (PDF_open_file(p, "hello.pdf") == -1) {
            printf("Error: %s\n", PDF_get_errmsg(p));
            return(2);
        }

        PDF_set_info(p, "Creator", "hello.c");
        PDF_set_info(p, "Author", "Thomas Merz");
        PDF_set_info(p, "Title", "Hello, world (C)!");

        PDF_begin_page(p, a4_width, a4_height);         /* start a new page*/

        /* Change "host" encoding to "winansi" or whatever you need! */
        font = PDF_load_font(p, "Helvetica-Bold", 0, "host", "");

        PDF_setfont(p, font, 24);
        PDF_set_text_pos(p, 50, 700);
        PDF_show(p, "Hello, world!");
        PDF_continue_text(p, "(says C)");
        PDF_end_page(p);                                /* close page */

        PDF_close(p);                                   /* close PDF document*/
    }

    PDF_CATCH(p) {
        printf("PDFlib exception occurred in hello sample:\n");
        printf("[%d] %s: %s\n",
            PDF_get_errnum(p), PDF_get_apiname(p), PDF_get_errmsg(p));
        PDF_delete(p);
        return(2);
    }

    PDF_delete(p);                      /* delete the PDFlib object */

    return 0;
}
```

2.4.3 Using PDFlib as a DLL loaded at Runtime

While most clients will use PDFlib as a statically bound library or a dynamic library which is bound at link time, you can also load the PDFlib DLL at runtime and dynamically fetch pointers to all API functions. This is especially useful to load the PDFlib DLL only on demand, and on MVS where the library is customarily loaded as a DLL at runtime without explicitly linking against PDFlib. PDFlib supports a special mechanism to facilitate this dynamic usage. It works according to the following rules:

- Include *pdflibdl.h* instead of *pdflib.h*.
- Use *PDF_new_dl( )* and *PDF_delete_dl( )* instead of *PDF_new( )* and *PDF_delete( )*.
- *Use PDF_TRY_DL( ) and PDF_CATCH_DL( )* instead of *PDF_TRY( )* and *PDF_CATCH( )*.
- Use function pointers for all other PDFlib calls.
- *PDF_get_opaque( )* must not be used.
- Compile the auxiliary module *pdflibdl.c* and link your application against it.

Note Loading the PDFlib DLL at runtime is supported on selected platforms only.

The following example loads the PDFlib DLL at runtime using this technique:

```
#include <stdio.h>
#include <stdlib.h>

#include "pdflibdl.h"

int
main(void)
{
    PDF *p;
    int font;
    PDFlib_api *PDFlib;

    /* load the PDFlib dynamic library and create a new PDFlib object*/
    if ((PDFlib = PDF_new_dl(&p)) == (PDFlib_api *) NULL)
    {
        printf("Couldn't create PDFlib object (DLL not found?)\n");
        return(2);
    }

    PDF_TRY_DL(PDFlib, p) {
        /* open new PDF file */
        if (PDFlib->PDF_open_file(p, "hellodl.pdf") == -1) {
            printf("Error: %s\n", PDFlib->PDF_get_errmsg(p));
            return(2);
        }

        PDFlib->PDF_set_info(p, "Creator", "hello.c");
        PDFlib->PDF_set_info(p, "Author", "Thomas Merz");
        PDFlib->PDF_set_info(p, "Title", "Hello, world (C DLL)!");

        PDFlib->PDF_begin_page(p, a4_width, a4_height);/* start a new page */

        /* Change "host" encoding to "winansi" or whatever you need! */
        font = PDFlib->PDF_load_font(p, "Helvetica-Bold", 0, "host", "");

        PDFlib->PDF_setfont(p, font, 24);
        PDFlib->PDF_set_text_pos(p, 50, 700);
        PDFlib->PDF_show(p, "Hello, world!");
        PDFlib->PDF_continue_text(p, "(says C DLL)");
        PDFlib->PDF_end_page(p);/* close page */

        PDFlib->PDF_close(p);/* close PDF document */
    }

    PDF_CATCH_DL(PDFlib, p) {
        printf("PDFlib exception occurred in hellodl sample:\n");
        printf("[%d] %s: %s\n",
```

```
        PDFlib->PDF_get_errnum(p), PDFlib->PDF_get_apiname(p),
        PDFlib->PDF_get_errmsg(p));
        PDF_delete_dl(PDFlib, p);
        return(2);
    }

    /* delete the PDFlib object and unload the library */
    PDF_delete_dl(PDFlib, p);

    return 0;
}
```

2.4.4 Error Handling in C

PDFlib supports structured exception handling with try/catch clauses. This allows C and C++ clients to catch exceptions which are thrown by PDFlib, and react on the exception in an adequate way. In the catch clause the client will have access to a string describing the exact nature of the problem, a unique exception number, and the name of the PDFlib API function which threw the exception. The general structure of a PDFlib C client program with exception handling looks as follows:

```
PDF_TRY(p)
{
        ...some PDFlib instructions...
}
PDF_CATCH(p)
{
        printf("PDFlib exception occurred in hello sample:\n");
        printf("[%d] %s: %s\n",
            PDF_get_errnum(p), PDF_get_apiname(p), PDF_get_errmsg(p));
        PDF_delete(p);
        return(2);
}

PDF_delete(p);
```

Note PDF_TRY/PDF_CATCH are implemented as tricky preprocessor macros. Accidentally omitting one of these will result in compiler error messages which may be difficult to comprehend. Make sure to use the macros exactly as shown above, with no additional code between the TRY and CATCH clauses (except PDF_CATCH( )).

If you want to leave a try clause before its end you must inform the exception machinery before, using the PDF_EXIT_TRY( ) macro. No other PDFlib function must be called between this macro and the end of the try block.

An important task of the catch clause is to clean up PDFlib internals using *PDF_delete( )* and the pointer to the PDFlib object. *PDF_delete( )* will also close the output file if necessary. PDFlib functions other than *PDF_delete( )*, *PDF_get_opaque( )* and the exception functions *PDF_get_errnum( ), PDF_get_apiname( ),* and *PDF_get_errmsg( )* must not be called from within a client-supplied error handler. After fatal exceptions the PDF document cannot be used, and will be left in an incomplete and inconsistent state. Obviously, the appropriate action when an exception occurs is completely application specific.

For C and C++ clients which do not catch exceptions, the default action upon exceptions is to issue an appropriate message on the standard error channel, and exit on fatal errors. The PDF output file will be left in an incomplete state! Since this may not be ade-

quate for a library routine, for serious PDFlib projects it is strongly advised to leverage PDFlib's exception handling facilities. A user-defined catch clause may, for example, present the error message in a GUI dialog box, and take other measures instead of aborting.

Old-style error handlers. In addition to structured exception handling PDFlib also supports the notion of a client-supplied callback function which be called when an exception occurs. However, this method is considered obsolete and supported for compatibility reasons only. Error handlers will be ignored in *PDF_TRY* blocks.

2.4.5 Memory Management in C

In order to allow for maximum flexibility, PDFlib's internal memory management routines (which are based on standard C *malloc/free)* can be replaced by external procedures provided by the client. These procedures will be called for all PDFlib-internal memory allocation or deallocation. Memory management routines can be installed with a call to *PDF_new2( )*, and will be used in lieu of PDFlib's internal routines. Either all or none of the following routines must be supplied:

- an allocation routine
- a deallocation (free) routine
- a reallocation routine for enlarging memory blocks previously allocated with the allocation routine.

The signatures of the memory routines can be found in Section 7.2, »General Functions«, page 148. These routines must adhere to the standard C *malloc/free/realloc* semantics, but may choose an arbitrary implementation. All routines will be supplied with a pointer to the calling PDFlib object. The only exception to this rule is that the very first call to the allocation routine will supply a PDF pointer of NULL. Client-provided memory allocation routines must therefore be prepared to deal with a NULL PDF pointer.

Using the *PDF_get_opaque( )* function, an opaque application specific pointer can be retrieved from the PDFlib object. The opaque pointer itself is supplied by the client in the *PDF_new2( )* call. The opaque pointer is useful for multi-threaded applications which may want to keep a pointer to thread- or class specific data inside the PDFlib object, for use in memory management or error handling.

2.5 C++ Binding

2.5.1 Availability and Special Considerations for C++

In addition to the *pdflib.h* C header file, an object-oriented wrapper for C++ is supplied for PDFlib clients. It requires the *pdflib.hpp* header file, which in turn includes *pdflib.h*. The corresponding *pdflib.cpp* module should be linked against the application which in turn should be linked against the generic PDFlib C library.

Using the C++ object wrapper replaces the *PDF_* prefix in all PDFlib function names with a more object-oriented approach. Keep this in mind when reading the PDFlib API descriptions in this manual which are documented in C style.

2.5.2 The »Hello world« Example in C++

```
#include <iostream>

#include "pdflib.hpp"

int
main(void)
{
    try {
        int font;
        PDFlib p;       // the PDFlib object

        // Open new PDF file
        if (p.open_file("hello.pdf") == -1) {
            cerr << "Error: " << p.get_errmsg() << endl;
            return 2;
        }

        p.set_info("Creator", "hello.cpp");
        p.set_info("Author", "Thomas Merz");
        p.set_info("Title", "Hello, world (C++)!");

        // start a new page
        p.begin_page((float) a4_width, (float) a4_height);

        // Change "host" encoding to "winansi" or whatever you need!
        font = p.load_font("Helvetica-Bold", "host", "");

        p.setfont(font, 24);
        p.set_text_pos(50, 700);
        p.show("Hello, world!");
        p.continue_text("(says C++)");
        p.end_page();                       // finish page
        p.close();                          // close PDF document
    }
    catch (PDFlib::Exception &ex) {
        cerr << "PDFlib exception occurred in hello sample: " << endl;
        cerr << "[" << ex.get_errnum() << "] " << ex.get_apiname()
            << ": " << ex.get_errmsg() << endl;
        return 2;
    }

    return 0;
}
```

2.5.3 Error Handling in C++

In order to provide extended error information the PDFlib class provides a public *PDFlib::Exception* class which exposes methods for retrieving the detailed error message, the exception number, and the name of the PDFlib API function which threw the exception.

Native C++ exceptions thrown by PDFlib routines will behave as expected. The following code fragment will catch exceptions thrown by PDFlib:

```
try {
        ...some PDFlib instructions...
catch (PDFlib::Exception &ex) {
```

```
        cerr << "PDFlib exception occurred in hello sample: " << endl;
        cerr << "[" << ex.get_errnum() << "] " << ex.get_apiname()
            << ": " << ex.get_errmsg() << endl;
        return 2;
}
```

2.5.4 Memory Management in C++

Client-supplied memory management for the C++ binding works the same as with the C language binding.

The PDFlib constructor accepts an optional error handler, optional memory management procedures, and an optional opaque pointer argument. Default NULL arguments are supplied in *pdflib.hpp* which will result in PDFlib's internal error and memory management routines becoming active. All memory management functions must be »C« functions, not C++ methods.

2.6 Java Binding

Java supports a portable mechanism for attaching native language code to Java programs, the Java Native Interface (JNI). The JNI provides programming conventions for calling native C or C++ routines from within Java code, and vice versa. Each C routine has to be wrapped with the appropriate code in order to be available to the Java VM, and the resulting library has to be generated as a shared or dynamic object in order to be loaded into the Java VM.

PDFlib supplies JNI wrapper code for using the library from Java. This technique allows us to attach PDFlib to Java by loading the shared library from the Java VM. The actual loading of the library is accomplished via a static member function in the *pdflib* Java class. Therefore, the Java client doesn't have to bother with the specifics of shared library handling.

Taking into account PDFlib's stability and maturity, attaching the native PDFlib library to the Java VM doesn't impose any stability or security restrictions on your Java application, while at the same time offering the performance benefits of a native implementation. Regarding portability remember that PDFlib is available for all platforms where there is a Java VM!

2.6.1 Installing the PDFlib Java Edition

For the PDFlib binding to work, the Java VM must have access to the PDFlib Java wrapper and the PDFlib Java package.

The PDFlib Java package. PDFlib is organized as a Java package with the following package name:

```
com.pdflib.pdflib
```

This package is available in the *pdflib.jar* file and contains a single class called *pdflib*. You can generate an abbreviated HTML-based version of the PDFlib API reference (this manual) using the *javadoc* utility since the PDFlib class contains the necessary *javadoc* comments. *javadoc*-generated documentation is contained in the PDFlib binary distribution. Comments on using PDFlib with specific Java IDEs may be found in text files in the distribution set.

In order to supply this package to your application, you must add *pdflib.jar* to your *CLASSPATH* environment variable, add the option *-classpath pdflib.jar* in your calls to the Java compiler and runtime, or perform equivalent steps in your Java IDE. In the JDK you can configure the Java VM to search for native libraries in a given directory by setting the *java.library.path* property to the name of the directory, e.g.

```
java -Djava.library.path=. pdfclock
```

You can check the value of this property as follows:

```
System.out.println(System.getProperty("java.library.path"));
```

In addition, the following platform-dependent steps must be performed:

Unix. The library *libpdf_java.so* (on Mac OS X: *libpdf_java.jnilib)* must be placed in one of the default locations for shared libraries, or in an appropriately configured directory.

Windows. The library *pdf_java.dll* must be placed in the Windows system directory, or a directory which is listed in the PATH environment variable.

PDFlib servlets and Java application servers. PDFlib is perfectly suited for server-side Java applications, especially servlets. The PDFlib distribution contains examples of PDFlib Java servlets which demonstrate the basic use. When using PDFlib with a specific servlet engine the following configuration issues must be observed:

- The directory where the servlet engine looks for native libraries varies among vendors. Common candidate locations are system directories, directories specific to the underlying Java VM, and local directories of the servlet engine. Please check the documentation supplied by the vendor of your servlet engine.
- Servlets are often loaded by a special class loader which may be restricted, or use a dedicated classpath. For some servlet engines it is required to define a special engine classpath to make sure that the PDFlib package will be found.

More detailed notes on using PDFlib with specific servlet engines and Java application servers can be found in additional documentation in the PDFlib distribution.

Note Since the EJB (Enterprise Java Beans) specification disallows the use of native libraries, PDFlib cannot be used within EJBs.

2.6.2 The »Hello world« Example in Java

```
import java.io.*;
import com.pdflib.pdflib;
import com.pdflib.PDFlibException;

public class hello
{
    public static void main (String argv[])
    {
        int font;
        pdflib p = null ;

        try{
            p = new pdflib();
```

```
            if (p.open_file("hello.pdf") == -1) {
                throw new Exception("Error: " + p.get_errmsg());
            }

            p.set_info("Creator", "hello.java");
            p.set_info("Author", "Thomas Merz");
            p.set_info("Title", "Hello world (Java)");

            p.begin_page(595, 842);

            /* Change "host" encoding to "winansi" or whatever you need! */
            font = p.load_font("Helvetica-Bold", "host", "");

            p.setfont(font, 18);

            p.set_text_pos(50, 700);
            p.show("Hello world!");
            p.continue_text("(says Java)");
            p.end_page();

            p.close();

        } catch (PDFlibException e) {
            System.err.print("PDFlib exception occurred in hello sample:\n");
            System.err.print("[" + e.get_errnum() + "] " + e.get_apiname() +
                    ": " + e.getMessage() + "\n");

        } catch (Exception e) {
            System.err.println(e.getMessage());

        } finally {
            if (p != null) {
                p.delete();             /* delete the PDFlib object */
            }
        }
    }
}
```

2.6.3 Error Handling in Java

The Java binding installs a special error handler which translates PDFlib errors to native Java exceptions. In case of an exception PDFlib will throw a native Java exception of the following class:

```
PDFlibException
```

The Java exceptions can be dealt with by the usual try/catch technique:

```
try {

...some PDFlib instructions...

} catch (PDFlibException e) {
            System.err.print("PDFlib exception occurred in hello sample:\n");
            System.err.print("[" + e.get_errnum() + "] " + e.get_apiname() +
                            ": " + e.getMessage() + "\n");

} catch (Exception e) {
```

```
        System.err.println(e.getMessage());

} finally {
        if (p != null) {
            p.delete();                     /* delete the PDFlib object */
        }
}
```

Since PDFlib declares appropriate *throws* clauses, client code must either catch all possible PDFlib exceptions, or declare those itself.

2.7 .NET Binding

(This section is not included in this edition of the PDFlib manual.)

2.8 Perl Binding

Perl[1] supports a mechanism for extending the language interpreter via native C libraries. The PDFlib wrapper for Perl consists of a C wrapper file and a Perl package module. The C module is used to build a shared library which the Perl interpreter loads at runtime, with some help from the package file. Perl scripts refer to the shared library module via a *use* statement.

2.8.1 Installing the PDFlib Perl Edition

The Perl extension mechanism loads shared libraries at runtime through the DynaLoader module. The Perl executable must have been compiled with support for shared libraries (this is true for the majority of Perl configurations).

For the PDFlib binding to work, the Perl interpreter must access the PDFlib Perl wrapper and the module file *pdflib_pl.pm*. In addition to the platform-specific methods described below you can add a directory to Perl's *@INC* module search path using the *-I* command line option:

```
perl -I/path/to/pdflib hello.pl
```

Unix. Perl will search both *pdflib_pl.so* (on Mac OS X: *pdflib_pl.dylib)* and *pdflib_pl.pm* in the current directory, or the directory printed by the following Perl command:

```
perl -e 'use Config; print $Config{sitearchexp};'
```

Perl will also search the subdirectory *auto/pdflib_pl*. Typical output of the above command looks like

```
/usr/lib/perl5/site_perl/5.8/i686-linux
```

Windows. PDFlib supports the ActiveState port of Perl 5 to Windows, also known as ActivePerl.[2] Both *pdflib_pl.dll* and *pdflib_pl.pm* will be searched in the current directory, or the directory printed by the following Perl command:

```
perl -e "use Config; print $Config{sitearchexp};"
```

1. See http://www.perl.com
2. See http://www.activestate.com

Typical output of the above command looks like

```
C:\Program Files\Perl5.8\site\lib
```

2.8.2 The »Hello world« Example in Perl

```
use pdflib_pl 5.0.2;

$p = PDF_new();

eval {
        if (PDF_open_file($p, "hello.pdf") == -1) {
            printf("Error: %s\n", PDF_get_errmsg($p));
            exit;
        }

        PDF_set_info($p, "Creator", "hello.pl");
        PDF_set_info($p, "Author", "Thomas Merz");
        PDF_set_info($p, "Title", "Hello world (Perl)");

        PDF_begin_page($p, 595, 842);
        $font = PDF_load_font($p, "Helvetica-Bold", "host", "");

        PDF_setfont($p, $font, 18.0);

        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);
};

if ($@) {
        printf("hello: PDFlib Exception occurred:\n");
        printf(" $@\n");
        exit;
}

PDF_delete($p);
```

2.8.3 Error Handling in Perl

The Perl binding installs a special error handler which translates PDFlib errors to native Perl exceptions. The Perl exceptions can be dealt with by applying the appropriate language constructs, i.e., by bracketing critical sections:

```
eval {
        ...some PDFlib instructions...
};
die "Exception caught" if $@;
```

2.9 PHP Binding

2.9.1 Installing the PDFlib PHP Edition

Detailed information about the various flavors and options for using PDFlib with PHP[1], including the question of whether or not to use a loadable PDFlib module for PHP, can be found in the *readme.txt* file which is part of the PDFlib distribution.

Note *We do not recommend using the PDFlib COM edition with PHP. Use the native PDFlib binding for PHP instead.*

You must configure PHP so that it knows about the external PDFlib library. You have two choices:

- Add one of the following lines in *php.ini:*

```
extension=libpdf_php.so        ; for Unix
extension=libpdf_php.dylib     ; for Mac OS X
extension=libpdf_php.dll       ; for Windows
```

  PHP will search the library in the directory specified in the *extension_dir* variable in *php.ini* on Unix, and in the standard system directories on Windows. You can test which version of the PHP PDFlib binding you have installed with the following one-line PHP script:

```
<?phpinfo()?>
```

  This will display a long info page about your current PHP configuration. On this page check the section titled *pdf*. If this section contains *PDFlib GmbH Version* (and the PDFlib version number) you are using the supported new PDFlib wrapper. The unsupported old wrapper will display *PDFlib Version* instead.
- Load PDFlib at runtime with one of the following lines at the start of your script:

```
dl("libpdf_php.so");           # for Unix
dl("libpdf_php.dll");          # for Windows
```

Modified error return for PDFlib functions in PHP. Since PHP uses the convention of returning the value 0 (FALSE) when an error occurs within a function, all PDFlib functions have been adjusted to return 0 instead of -1 in case of an error. This difference is noted in the function descriptions in Section 7, »API Reference for PDFlib, PDI, and PPS«, page 147. However, take care when reading the code fragment examples in Section 3, »PDFlib Programming«, page 41 since these use the usual PDFlib convention of returning -1 in case of an error.

File name handling in PHP. Unqualified file names (without any path component) and relative file names for PDF, image, font and other disk files are handled differently in Unix and Windows versions of PHP:

- PHP on Unix systems will find files without any path component in the directory where the script is located.
- PHP on Windows will find files without any path component only in the directory where the PHP DLL is located.

1. See http://www.php.net

In order to provide platform-independent file name handling use of PDFlib's Search-Path facility (see Section 3.1.6, »Resource Configuration and File Searching«, page 47) is strongly recommended.

2.9.2 The »Hello world« Example in PHP

```
<?php
$p = PDF_new();
/*  open new PDF file; insert a file name to create the PDF on disk */
if (PDF_open_file($p, "") == 0) {
        die("Error: " . PDF_get_errmsg($p));
}

PDF_set_info($p, "Creator", "hello.php");
PDF_set_info($p, "Author", "Rainer Schaaf");
PDF_set_info($p, "Title", "Hello world (PHP)");

PDF_begin_page($p, 595, 842); /* start a new page     */

# Change "host" encoding to "winansi" or whatever you need!
$font = PDF_load_font($p, "Helvetica-Bold", "host", "");
PDF_setfont($p, $font, 24.0);

PDF_set_text_pos($p, 50, 700);
PDF_show($p, "Hello world!");
PDF_continue_text($p, "(says PHP)");

PDF_end_page($p);           /* close page*/
PDF_close($p);              /* close PDF document*/

$buf = PDF_get_buffer($p);
$len = strlen($buf);

header("Content-type: application/pdf");
header("Content-Length: $len");
header("Content-Disposition: inline; filename=hello.pdf");
print $buf;

PDF_delete($p);             /* delete the PDFlib object */
?>
```

2.9.3 Error Handling in PHP

When a PDFlib exception occurs, a PHP exception is thrown. Unfortunately, PHP does not yet support structured exception handling: there is no way to catch exceptions and act appropriately. Do not disable PHP exceptions when using PDFlib, or you will run into serious trouble.

PDFlib warnings (nonfatal errors) are mapped to PHP warnings, which can be disabled in *php.ini*. Alternatively, warnings can be disabled at runtime with a PDFlib function like in any other language binding:

```
PDF_set_parameter($p, "warning", "false");
```

2.10 Python Binding

2.10.1 Installing the PDFlib Python Edition

The Python[1] extension mechanism works by loading shared libraries at runtime. For the PDFlib binding to work, the Python interpreter must have access to the PDFlib Python wrapper:

Unix. The library *pdflib_py.so* (on Mac OS X: *pdflib_py.dylib)* will be searched in the directories listed in the PYTHONPATH environment variable.

Windows. The library *pdflib_py.dll* will be searched in the directories listed in the PYTHONPATH environment variable.

2.10.2 The »Hello world« Example in Python

```
from sys import *
from pdflib_py import *

p = PDF_new()

if PDF_open_file(p, "hello.pdf") == -1:
    print "Error: " + PDF_get_errmsg(p) + "\n"
    exit(2);

PDF_set_info(p, "Author", "Thomas Merz")
PDF_set_info(p, "Creator", "hello.py")
PDF_set_info(p, "Title", "Hello world (Python)")

PDF_begin_page(p, 595, 842)
font = PDF_load_font(p, "Helvetica-Bold", "host", "")

PDF_setfont(p, font, 18.0)

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)

PDF_delete(p);
```

2.10.3 Error Handling in Python

The Python binding installs a special error handler which translates PDFlib errors to native Python exceptions. The Python exceptions can be dealt with by the usual try/catch technique:

```
try:
        ...some PDFlib instructions...
except:
        print 'Exception caught!'
```

1. See http://www.python.org

2.11 RPG Binding

PDFlib provides a */copy* module that defines all prototypes and some useful constants needed to compile ILE-RPG programs with embedded PDFlib functions.

Since all functions provided by PDFlib are implemented in the C language, you have to add *x'00'* at the end of each string value passed to a PDFlib function. All strings returned from PDFlib will have this terminating *x'00'* as well.

2.11.1 Compiling and Binding RPG Programs for PDFlib

Using PDFlib functions from RPG requires the compiled PDFlib service program. To include the PDFlib definitions at compile time you have to specify the name in the D specs of your ILE-RPG program:

```
d/copy QRPGLESRC,PDFLIB
```

If the PDFlib source file library is not on top of your library list you have to specify the library as well:

```
d/copy PDFsrclib/QRPGLESRC,PDFLIB
```

Before you start compiling your ILE-RPG program you have to create a binding directory that includes the PDFLIB service program shipped with PDFlib. The following example assumes that you want to create a binding directory called PDFLIB in the library PDFLIB:

```
CRTBNDDIR BNDDIR(PDFLIB/PDFLIB) TEXT('PDFlib Binding Directory')
```

After creating the binding directory you need to add the PDFLIB service program to your binding directory. The following example assumes that you want to add the service program PDFLIB in the library PDFLIB to the binding directory created earlier.

```
ADDBNDDIRE BNDDIR(PDFLIB/PDFLIB) OBJ((PDFLIB/PDFLIB *SRVPGM))
```

Now you can compile your program using the *CRTBNDRPG* command (or option 14 in PDM):

```
CRTBNDRPG PGM(PDFLIB/HELLO) SRCFILE(PDFLIB/QRPGLESRC) SRCMBR(*PGM) DFTACTGRP(*NO)
BNDDIR(PDFLIB/PDFLIB)
```

2.11.2 The »Hello world« Example in RPG

```
*****************************************************************************************
     d/copy QRPGLESRC,PDFLIB
*****************************************************************************************
     d p               S                 *
     d font            s             10i 0
      *
     d error           s             50
     d errmsg_p        s               *
     d errmsg          s            200    based(errmsg_p)
      *
     d filename        s            256
     d fontname        s             50
     d fontenc         s             50
     d infokey         s             50
     d infoval         s            200
```

```
     d text            s            200
     d n               s              1    inz(x'00')
******************************************************************************************
     c                   clear                   error
      *
      *    Init on PDFlib
     c                   eval      p=pdf_new
     c                   if        p=*null
     c                   eval      error='Couldn''t create PDFlib object '+
     c                                   '(out of memory)!'
     c                   exsr      exit
     c                   endif
      *
      *    Open new pdf file
     c                   eval      filename='hello.pdf'+x'00'
     c                   if        PDF_open_file(p:filename) = -1
     c                   exsr      geterrmsg
     c                   exsr      exit
     c                   endif
      *    Set info "Creator"
     c                   eval      infokey='Creator'+x'00'
     c                   eval      infoval='hello.rpg'+x'00'
     c                   callp     PDF_set_info(p:infokey:infoval)
      *    Set info "Author"
     c                   eval      infokey='Author'+x'00'
     c                   eval      infoval='Thomas Merz'+x'00'
     c                   callp     PDF_set_info(p:infokey:infoval)
      *    Set info "Title"
     c                   eval      infokey='Title'+x'00'
     c                   eval      infoval='Hello, world (RPG)!'+x'00'
     c                   callp     PDF_set_info(p:infokey:infoval)
      *    Start a new page
     c                   callp     PDF_begin_page(p:a4_width:a4_height)
      *
     c                   eval      fontname='Helvetica-Bold'+x'00'
      *    Change "host" encoding to "ebcdic" or whatever you need!
     c                   eval      fontenc='host'+x'00'
     c                   eval      font=PDF_load_font(p:fontname:0:fontenc:n)
      *
     c                   callp     PDF_setfont(p:font:24)
     c                   callp     PDF_set_text_pos(p:50:700)
      *
     c                   eval      text='Hello world!'+x'00'
     c                   callp     PDF_show(p:text)
     c                   eval      text='(says ILE RPG)'+x'00'
     c                   callp     PDF_continue_text(p:text)
      *    Close page
     c                   callp     PDF_end_page(p)
      *    Close PDF document
     c                   callp     PDF_close(p)
      *    Delete the PDF object
     c                   callp     PDF_delete(p)
      *
     c                   exsr      exit
******************************************************************************************
     c     geterrmsg     begsr
     c                   eval      errmsg_p=PDF_get_errmsg(p)
     c                   if        errmsg_p<>*NULL
```

```
     c                   eval      error=%subst(errmsg:1:%scan(x'00':errmsg)-1)
     c                   endif
     c                   endsr
******************************************************************************************
     c     exit          begsr
     c                   if        error<>*blanks
     c                   eval      error='Error: '+error
     c     error         dsply
     c                   endif
     c                   seton                                        lr
     c                   return
     c                   endsr
```

You can compile this program as follows:

```
CRTBNDDIR BNDDIR(PDFLIB/PDFLIB) TEXT('PDFlib Binding Directory')
ADDBNDDIRE BNDDIR(PDFLIB/PDFLIB) OBJ((PDFLIB/PDFLIB *SRVPGM))
CRTBNDRPG PGM(PDFLIB/HELLO) SRCFILE(PDFLIB/QRPGLESRC) SRCMBR(*PGM) DFTACTGRP(*NO) +
        BNDDIR(PDFLIB/PDFLIB)
```

2.11.3 Error Handling in RPG

PDFlib clients written in ILE-RPG can install an error handler in PDFlib which will be activated when an exception occurs. Since ILE-RPG translates all procedure names to uppercase, the name of the error handler procedure should be specified in uppercase. The following skeleton demonstrates this technique:

```
******************************************************************************************
     d/copy QRPGLESRC,PDFLIB
******************************************************************************************
     d p               S               *
     d font            s             10i 0
      *
     d error           s             50
      *
     d errhdl          s               *   procptr
      *
      *  Prototype for exception handling procedure
      *
     d errhandler      PR
     d  p                              *   value
     d  type                         10i 0 value
     d  shortmsg                     2048
******************************************************************************************
     c                   clear                   error
      *
*        Set the procedure pointer to the ERRHANDLER procedure.
*
     c                   eval      errhdl=%paddr('ERRHANDLER')
      *
     c                   eval      p=pdf_new2(errhdl:*null:*null:*null:*null)

...PDFlib instructions...

     c                   callp     PDF_delete(p)
      *
     c                   exsr      exit
******************************************************************************************
```

```
     c     exit          begsr
     c                   if        error<>*blanks
     c     error         dsply
     c                   endif
     c                   seton                                        lr
     c                   return
     c                   endsr
******************************************************************************************
      *  If any of the PDFlib functions will cause an exception, first the error handler
      *  will be called and after that we will get a regular RPG exception.
     c     *pssr         begsr
     c                   exsr      exit
     c                   endsr
******************************************************************************************
      *  Exception Handler Procedure
      *  This procedure will be linked to PDFlib by passing the procedure pointer to
      *  PDF_new2. This procedure will be called when a PDFlib exception occurs.
      *
******************************************************************************************
     p errhandler      B
     d errhandler      PI
     d  p                              *   value
     d  type                         10i 0 value
     d  c_message                    2048
      *
     d  length         s             10i 0
      *
      *    Chop off the trailing x'00' (we are called by a C program)
      *    and set the error (global) string
     c                   clear                   error
     c     x'00'         scan      c_message     length                    50
     c                   sub       1             length
     c                   if        *in50 and length>0
     c                   if        length>%size(error)
     c                   eval      error=c_message
     c                   else
     c                   eval      error=%subst(c_message:1:length)
     c                   endif
     c                   endif
      *
      *    Always call PDF_delete to clean up PDFlib
     c                   callp     PDF_delete(p)
      *
     c                   return
      *
     p errhandler      E
```

2.12 Tcl Binding

2.12.1 Installing the PDFlib Tcl Edition

The Tcl [1]extension mechanism works by loading shared libraries at runtime. For the PDFlib binding to work, the Tcl shell must have access to the PDFlib Tcl wrapper shared library and the package index file *pkgIndex.tcl*. You can use the following idiom in your

1. See http://dev.scriptics.com

script to make the library available from a certain directory (this may be useful if you want to deploy PDFlib on a machine where you don't have root privilege for installing PDFlib):

```
lappend auto_path /path/to/pdflib
```

Unix. The library *pdflib_tcl.so* (on Mac OS X: *pdflib_tcl.dylib)* must be placed in one of the default locations for shared libraries, or in an appropriately configured directory. Usually both *pkgIndex.tcl* and *pdflib_tcl.so* will be placed in the directory

```
/usr/lib/tcl8.3/pdflib
```

Windows. The files *pkgIndex.tcl* and *pdflib_tcl.dll* will be searched for in the directories

```
C:\Program Files\Tcl\lib\pdflib
C:\Program Files\Tcl\lib\tcl8.3\pdflib
```

2.12.2 The »Hello world« Example in Tcl

```
package require pdflib 5.0.2

set p [PDF_new]

if {[PDF_open_file $p "hello.pdf"] == -1} {
        puts stderr "Error:  [PDF_get_errmsg $p]"
        exit
}

PDF_set_info $p "Creator" "hello.tcl"
PDF_set_info $p "Author" "Thomas Merz"
PDF_set_info $p "Title" "Hello world (Tcl)"

PDF_begin_page $p 595 842
set font [PDF_load_font $p "Helvetica-Bold" "host" "" ]

PDF_setfont $p $font 18.0

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

PDF_delete $p
```

2.12.3 Error Handling in Tcl

The Tcl binding installs a special error handler which translates PDFlib errors to native Tcl exceptions. The Tcl exceptions can be dealt with by the usual try/catch technique:

```
if [ catch { ...some PDFlib instructions... } result ] {
        puts stderr "Exception caught!"
        puts stderr $result
}
```

3 PDFlib Programming

3.1 General Programming

3.1.1 PDFlib Program Structure and Function Scopes

PDFlib applications must obey certain structural rules which are very easy to understand. Writing applications according to these restrictions is straightforward. For example, you don't have to think about opening a document first before closing it. Since the PDFlib API is very closely modelled after the document/page paradigm, generating documents the »natural« way usually leads to well-formed PDFlib client programs.

PDFlib enforces correct ordering of function calls with a strict scoping system. The function descriptions document the allowed scope for a particular function. Calling a function from a different scope will trigger a PDFlib exception. PDFlib will also throw an exception if bad parameters are supplied by a library client.

The function descriptions in Chapter 7 reference these scopes; the scope definitions can be found in Table 3.1. Figure 3.1 depicts the nesting of scopes. PDFlib will throw an exception if a function is called outside the allowed scope. You can query the current scope with the *scope* parameter.

Table 3.1 Function scope definitions

scope name	*definition*
path	*started by one of PDF_moveto( ), PDF_circle( ), PDF_arc( ), PDF_arcn( ), or PDF_rect( ) terminated by any of the functions in Section 7.4.6, »Path Painting and Clipping«, page 180*
page	*between PDF_begin_page( ) and PDF_end_page( ), but outside of path scope*
template	*between PDF_begin_template( ) and PDF_end_template( ), but outside of path scope*
pattern	*between PDF_begin_pattern( ) and PDF_end_pattern( ), but outside of path scope*
font	*between PDF_begin_font( ) and PDF_end_font( ), but outside of glyph scope*
glyph	*between PDF_begin_glyph( ) and PDF_end_glyph( ), but outside of path scope*
document	*between PDF_open_*( ) and PDF_close( ), but outside of page, template, pattern, and font scope*
object	*in Java: the lifetime of the pdflib object, but outside of document scope; in other bindings between PDF_new( ) and PDF_delete( ), but outside of document scope*
null	*outside of object scope*
any	*when a function description mentions »any« scope it actually means any except null, since a PDFlib object doesn't even exist in null scope.*

3.1.2 Parameters

PDFlib's operation can be controlled by a variety of global parameters. These will retain their settings across the life span of the PDFlib object, or until they are explicitly changed by the client. The functions can be used for dealing with PDFlib parameters:

- *PDF_set_parameter( )* can be used to set parameters of type string.
- *PDF_set_value( )* can be used to set parameters with numerical values.
- *PDF_get_parameter( )* can be used to query parameters of type string.
- *PDF_get_value( )* can be used to query the values of numerical parameters.

Details of parameter names and possible values can be found in Chapter 7.

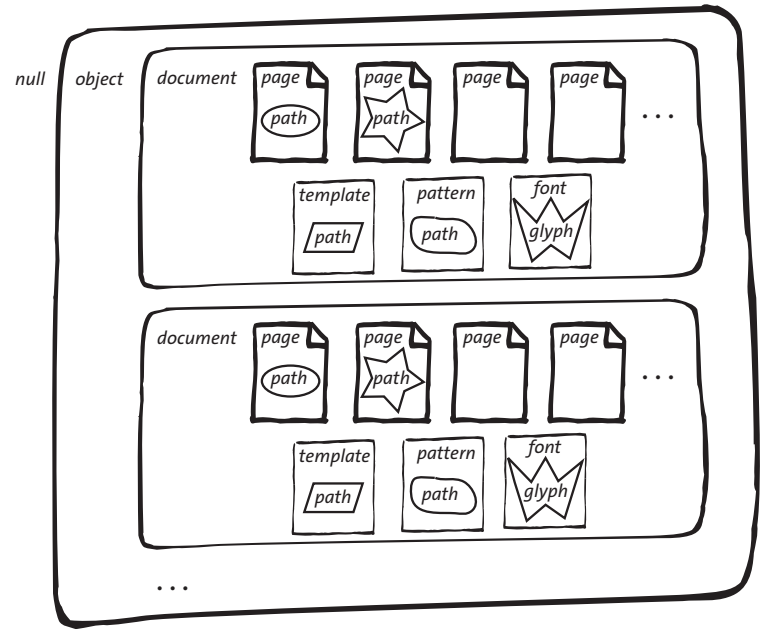


Fig. 3.1
Nesting of scopes

3.1.3 Exception Handling

Errors of a certain kind are called exceptions in many languages for good reasons – they are mere exceptions, and are not expected to occur very often during the lifetime of a program. The general strategy is to use conventional error reporting mechanisms (read: special error return codes) for function calls which may go wrong often times, and use a special exception mechanism for those rare occasions which don't warrant cluttering the code with conditionals. This is exactly the path that PDFlib goes: Some operations can be expected to go wrong rather frequently, for example:

- Trying to open an output file for which one doesn't have permission
- Trying to open an input PDF with a wrong file name
- Trying to open a corrupt image file

PDFlib signals such errors by returning a special value (usually –1, but 0 in the PHP binding) as documented in the API reference. Other events may be considered harmful, but will occur rather infrequently, e.g.

- running out of virtual memory
- scope violations (e.g., closing a document before opening it)
- supplying wrong parameters to PDFlib API functions (e.g., trying to draw a circle with a negative radius)

When PDFlib detects such a situation, an exception will be thrown instead of passing a special error return value to the caller. In the C programming language, which doesn't natively support exceptions, the client can install a custom routine (called an error handler) which will be called in case of an exception. However, the recommended method is to make use of *PDF_TRY( )/PDF_CATCH( )* blocks as detailed in Section 2.4.4, »Error Handling in C«, page 24.

It is important to understand that the generated PDF document cannot be finished after an exception occurred. The only method which can safely be called after an exception is *PDF_delete( )*. In the C language binding *PDF_get_apiname( ), PDF_get_errnum( ),* and *PDF_get_errmsg( )* may also be called. Calling any other PDFlib methods after an exception may lead to unexpected results. The exception (or data passed to the C error handler) will contain the following information:

- A unique error number (see Table 3.2);
- The name of the PDFlib API function which caused the exception;
- A descriptive text containing details of the problem;

C language clients can fetch this information using dedicated functions *(PDF_get_errnum( ), PDF_get_apiname( ),* and *PDF_get_errmsg( ))*, while in other languages it will be part of the exception object.

Table 3.2 Ranges of PDFlib exception numbers

error ranges	*reasons*
1000 – 1999	*(PDCORE library): memory, I/O, arguments, parameters/values, options*
2000 – 2999	*(PDFlib library): configuration, scoping, graphics and text, color, images, fonts, encodings, hypertext, PDF/X*
4000 – 4999	*(PDF import library PDI): configuration and parameter, corrupt PDF (file, object, or stream level)*

Disabling exceptions. Some exceptions can be disabled. These fall into two categories: non-fatal errors (warnings) and errors which may or may not justify an exception depending on client preferences.

Warnings generally indicate some problem in your PDFlib code which you should investigate more closely. However, processing may continue in case of non-fatal errors. For this reason, you can suppress warnings using the following function call:

```
PDF_set_parameter(p, "warning", "false");
```

The suggested strategy is to enable warnings during the development cycle (and closely examine possible warnings), and disable warnings in a production system.

Certain operations may be considered fatal for some clients, while others are prepared to deal with the situation. In these cases the behavior of the respective PDFlib API function changes according to a parameter. This distinction is implemented for loading fonts, images, PDF import documents, and ICC profiles. For example, if a font cannot be loaded due to some configuration problem one client may simply give up, while another may choose another font instead. When the parameter *fontwarning* is set to true, an exception will be thrown when the font cannot be loaded. Otherwise the function will return an error code instead. The parameter can be set as follows:

```
PDF_set_parameter(p, "fontwarning", "false");
```

Querying the reason of a failed function call. As noted above, the generated PDF output document must always be abandoned when an exception occurs. Some clients, however, may prefer to continue the document by adjusting some parameters. For example, when a particular font cannot be loaded most clients will give up the document, while others may prefer to work with a different font. This distinction can be achieved with the *fontwarning* etc. parameters. In this case it may be desirable to retrieve the error message that would have been part of the exception. In this situation the functions

PDF_get_errnum( ), *PDF_get_errmsg( )*, and *PDF_get_apiname( )* may be called immediately after a failed function call, i.e., a function call which returned with a -1 (in PHP: 0) error value.

The following code fragments summarize different strategies with respect to exception handling. The examples try to load and embed a font, assuming that this font is not available.

If the *fontwarning* parameter is *true* (which is the default) the document must be abandoned:

```
PDF_set_parameter(p, "fontwarning", "true");
font = PDF_load_font(p, "MyFontName", 0, "winansi", "embed");
/* unless an exception was thrown the font handle is valid;
 * when an exception occurred the PDF output cannot be continued
 */
```

If the *fontwarning* parameter is *false* the return value must be checked for validity:

```
PDF_set_parameter(p, "fontwarning", "false");
font = PDF_load_font(p, "MyFontName", 0, "winansi", "embed";
if (font == -1) {
        /* font handle is invalid, but the PDF output can be continued. */
        /* Try a different font or give up */
        ...
}
/* font handle is valid; continue as usual */
```

If the *fontwarning* parameter is *false* and the return value indicates failure, the reason of the failure can be queried in order to properly deal with the situation:

```
PDF_set_parameter(p, "fontwarning", "false");
font = PDF_load_font(p, "MyFontName", 0, "winansi", "embed";
if (font == -1) {
        /* font handle is invalid; find out what happened. */
        errmsg = PDF_get_errmsg(p);
        /* Log error message */
        /* Try a different font or give up */
        ...
}
/* font handle is valid; continue as usual */
```

3.1.4 Option Lists

Option lists are a powerful yet easy method to control PDFlib operations. Instead of requiring a multitude of function parameters, many PDFlib API methods support option lists, or optlists for short. These are strings which may contain an arbitrary number of options. Optlists support various data types and composite data like arrays. In most languages optlists can easily be constructed by concatenating the required keywords and values. C programmers may want to use the *sprintf( )* function in order to construct optlists. An optlist is a string containing one or more pairs of the form

```
name value(s)
```

Names and values, as well as multiple name/value pairs can be separated by arbitrary whitespace characters (space, tab, carriage return, newline) and/or an equal sign '='.

Simple values. Simple values may use any of the following data types:

- Boolean: *true* or *false*; if the value of a boolean option is omitted, the value *true* is assumed. As a shorthand notation *noname* can be used instead of *name false*.
- String: strings containing whitespace must be bracketed with *{* and *}*. An empty string can be constructed with *{ }*. The characters *{*, *}*, and \ must be preceded by a an additional \ character if they are supposed to be part of the string.
- Keyword: one of a predefined list of fixed keywords
- Float and integer: decimal floating point or integer numbers; point and comma can be used as decimal separators.
- Handle: several PDFlib-internal object handles, e.g., font handles, image handles. Technically these are always integer values.

Depending on the type and interpretation of an option additional restrictions may apply. For example, integer or float options may be restricted to a certain range of values; handles must be valid for the corresponding type of object, etc. Conditions for options are documented in their respective function descriptions in Chapter 7. Some examples for simple values (the first line shows a string containing a blank character):

```
PDF_open_pdi( ):        password {secret string}
PDF_create_gstate( ):   linewidth 0.5 blendmode overlay opacityfill 0.75
PDF_load_font( ):       embedding=true subsetting=true subsetlimit=50 kerning=false
PDF_load_font( ):       embedding subsetting subsetlimit=50 nokerning
```

List values. List values consist of multiple values, which may be simple values or list values in turn. Lists are bracketed with *{* and *}*. Some examples for list values:

```
PDF_fit_image( ):       boxsize {500 600} position {50 0}
PDF_create_gstate( ):   dasharray {11 22 33}
```

Color values. Color values are lists consisting of a color space keyword and a list with a varying number of float values depending on the particular color space. The color space keywords and semantics are the same as in *PDF_setcolor( )* (see Section 7.5.1, »Setting Color and Color Space«, page 182):

- The color space keywords *gray, rgb,* and *cmyk* can be supplied along with one, three, or four float values.
- The color space keyword *lab* can be supplied along with three float values.
- The color space keyword *spot* can be supplied along with a spot color handle. Alternatively, the color space keyword *spotname* can be supplied along with a spot color name and a float value containing the color tint.
- The color space keywords *iccbasedgray, iccbasedrgb,* and *iccbasedcmyk* can be supplied along with one, three, or four float values.
- The color space keyword *none* can be supplied to specify the absence of color.

As detailed in the respective function descriptions in Chapter 7, a particular option list may only supply a subset of the keywords presented above. Some examples for color values:

```
PDF_fill_textblock( ):  strokecolor { rgb 1 0 0 }
PDF_fill_textblock( ):  bordercolor none
```

3.1.5 The PDFlib Virtual File System (PVF)

In addition to disk files a facility called *PDFlib Virtual File System* (PVF) allows clients to directly supply data in memory without any disk files involved. This offers performance benefits and can be used for data fetched from a database which does not even exist on an isolated disk file, as well as other situations where the client already has the required data available in memory as a result of some processing.

PVF is based on the concept of named virtual read-only files which can be used just like regular file names with any API function. They can even be used in UPR configuration files. Virtual file names can be generated in an arbitrary way by the client. Obviously, virtual file names must be chosen such that name clashes with regular disk files are avoided. For this reason a hierarchical naming convention for virtual file names is recommended as follows (*filename* refers to a name chosen by the client which is unique in the respective category). It is also recommended to keep standard file name suffixes:

- Raster image files: */pvf/image/filename*
- font outline and metrics files (it is recommended to use the actual font name as the base portion of the file name): */pvf/font/filename*
- ICC profiles: */pvf/iccprofile/filename*
- Encodings and codepages: */pvf/codepage/filename*
- PDF documents: */pvf/pdf/filename*

When searching for a named file PDFlib will first check whether the supplied file name refers to a known virtual file, and then try to open the named file on disk.

Lifetime of virtual files. Some functions will immediately consume the data supplied in a virtual file, while others will read only parts of the file, with other fragments being used at a later point in time. For this reason close attention must be paid to the lifetime of virtual files. PDFlib will place an internal lock on every virtual file, and remove the lock only when the contents are no longer needed. Unless the client requested PDFlib to make an immediate copy of the data (using the *copy* option in *PDF_create_pvf( )*), the virtual file's contents must only be modified, deleted, or freed by the client when it is no longer locked by PDFlib. PDFlib will automatically delete all virtual files in *PDF_delete( )*. However, the actual file contents (the data comprising a virtual file) must always be freed by the client.

Different strategies. PVF supports different approaches with respect to managing the memory required for virtual files. These are governed by the fact that PDFlib may need access to a virtual file's contents after the API call which accepted the virtual file name, but never needs access to the contents after *PDF_close( )*. Remember that calling *PDF_delete_pvf( )* does not free the actual file contents (unless the *copy* option has been supplied), but only the corresponding data structures used for PVF file name administration. This gives rise to the following strategies:

- Minimize memory usage: it is recommended to call *PDF_delete_pvf( )* immediately after the API call which accepted the virtual file name, and another time after *PDF_close( )*. The second call is required because PDFlib may still need access to the data so that the first call refuses to unlock the virtual file. However, in some cases the first call will already free the data, and the second call doesn't do any harm. The client may free the file contents only when *PDF_delete_pvf( )* succeeded.
- Optimize performance by reusing virtual files: some clients may wish to reuse some data (e.g., font definitions) within various output documents, and avoid multiple

create/delete cycles for the same file contents. In this case it is recommended not to call *PDF_delete_pvf( )* as long as more PDF output documents using the virtual file will be generated.

- Lazy programming: if memory usage is not a concern the client may elect not to call *PDF_delete_pvf( )* at all. In this case PDFlib will internally delete all pending virtual files in *PDF_delete( )*.

In all cases the client may free the corresponding data only when *PDF_delete_pvf( )* returned successfully, or after *PDF_delete( )*.

3.1.6 Resource Configuration and File Searching

In most advanced applications PDFlib needs access to resources such as font file, encoding definition, ICC color profiles, etc. In order to make PDFlib's resource handling platform-independent and customizable, a configuration file can be supplied for describing the available resources along with the names of their corresponding disk files. In addition to a static configuration file, dynamic configuration can be accomplished at runtime by adding resources with *PDF_set_parameter( )*. For the configuration file we dug out a simple text format called *Unix PostScript Resource* (UPR) which came to life in the era of Display PostScript, and is still in use on several systems. However, we extended the original UPR format for our purposes. The UPR file format as used by PDFlib will be described below. There is a utility called *makepsres* (often distributed as part of the X Window System) which can be used to automatically generate UPR files from PostScript font outline and metrics files.

Resource categories. The resource categories supported by PDFlib are listed in Table 3.3. Other resource categories may be present in the UPR file for compatibility with Display PostScript installations, but they will silently be ignored.

Table 3.3 Resource categories supported in PDFlib

resource category name	*explanation*
SearchPath	*Relative or absolute path name of directories containing data files*
FontAFM	*PostScript font metrics file in AFM format*
FontPFM	*PostScript font metrics file in PFM format*
FontOutline	*PostScript, TrueType or OpenType font outline file*
Encoding	*text file containing an 8-bit encoding or code page table*
HostFont	*Name of a font installed on the system. The value can be encoded in ASCII or UTF-8 with initial BOM. The latter can be useful for localized host font names.*
ICCProfile	*name of an ICC color profile*
StandardOutputIntent	*name of a standard output condition for PDF/X*

Redundant resource entries should be avoided. For example, do not include multiple entries for a certain font's metrics data. Also, the font name as configured in the UPR file should exactly match the actual font name in order to avoid confusion (although PDFlib does not enforce this restriction).

In Mac OS Classic the colon character ':' must be used as a directory separator. The font names of resource-based PostScript Type 1 fonts (LWFN fonts) must be specified using the full path including volume name, for example:

```
Foo-Italic=Classic:Data:Fonts:FooIta
```

The UPR file format. UPR files are text files with a very simple structure that can easily be written in a text editor or generated automatically. To start with, let's take a look at some syntactical issues:

- Lines can have a maximum of 255 characters.
- A backslash '\' escapes newline characters. This may be used to extend lines.
- An isolated period character ' . ' serves as a section terminator.
- All entries are case-sensitive.
- Comment lines may be introduced with a percent '%' character, and terminated by the end of the line.
- Whitespace is ignored everywhere except in resource names and file names.

UPR files consist of the following components:

- A magic line for identifying the file. It has the following form:

  ```
  PS-Resources-1.0
  ```

- A section listing all resource categories described in the file. Each line describes one resource category. The list is terminated by a line with a single period character. Available resource categories are described below.
- A section for each of the resource categories listed at the beginning of the file. Each section starts with a line showing the resource category, followed by an arbitrary number of lines describing available resources. The list is terminated by a line with a single period character. Each resource data line contains the name of the resource (equal signs have to be quoted). If the resource requires a file name, this name has to be added after an equal sign. The *SearchPath* (see below) will be applied when PDFlib searches for files listed in resource entries.

File searching and the SearchPath resource category. PDFlib reads a variety of data items, such as raster images, font outline and metrics information, encoding definitions, PDF documents, and ICC color profiles from disk files. In addition to relative or absolute path names you can also use file names without any path specification. The *SearchPath* resource category can be used to specify a list of path names for directories containing the required data files. When PDFlib must open a file it will first use the file name exactly as supplied and try to open the file. If this attempt fails PDFlib will try to open the file in the directories specified in the *SearchPath* resource category one after another until it succeeds. *SearchPath* entries can be accumulated, and will be searched in reverse order (paths set at a later point in time will searched before earlier ones). This feature can be used to separate the PDFlib application from platform-specific file system schemes. In order to disable the search you can use a fully specified path name in the PDFlib functions.

On Windows PDFlib will initialize the *SearchPath* resource category with an entry read from the following registry entry:

```
HKLM\SOFTWARE\PDFlib\PDFlib\5.0.2\SearchPath
```

This registry entry may contain a list of path names separated by a semicolon ';' character.

On IBM iSeries the *SearchPath* resource category will be initialized with the following values:

```
/pdflib/5.0.2/fonts
/pdflib/5.0.2/bind/data
```

On MVS the SearchPath feature is not supported.

Sample UPR file. The following listing gives an example of a UPR configuration file as used by PDFlib. It describes some font metrics and outline files plus a custom encoding:

```
PS-Resources-1.0
SearchPath
FontAFM
FontPFM
FontOutline
Encoding
ICCProfile
.
SearchPath
/usr/local/lib/fonts
Classic:Data:Fonts
C:/psfonts/pfm
C:/psfonts
/users/kurt/my_images
.
FontAFM
Code-128=Code_128.afm
.
FontPFM
Foobar-Bold=foobb___.pfm

Mistral=c:/psfonts/pfm/mist____.pfm
.
FontOutline
Code-128=Code_128.pfa
ArialMT=Arial.ttf
.
Encoding
myencoding=myencoding.enc
.
ICCProfile
highspeedprinter=cmykhighspeed.icc
.
```

Searching for the UPR resource file. If only the built-in resources (e.g., PDF core font, built-in encodings, sRGB ICC profile) or system resources (host fonts) are to be used, a UPR configuration file is not required, since PDFlib will find all necessary resources without any additional configuration.

If other resources are to be used you can specify such resources via calls to *PDF_set_parameter( )* (see below) or in a UPR resource file. PDFlib reads this file automatically when the first resource is requested. The detailed process is as follows:

- If the environment variable *PDFLIBRESOURCE* is defined PDFlib takes its value as the name of the UPR file to be read. If this file cannot be read an exception will be thrown.
- If the environment variable *PDFLIBRESOURCE* is not defined PDFlib tries to open a file with the following name:

  ```
  upr (on MVS; a dataset is expected)
  ```

```
pdflib/<version>/fonts/pdflib.upr (on IBM eServer iSeries)
pdflib.upr (Windows, Unix, and all other systems)
```

If this file cannot be read no exception will be thrown.

- On Windows PDFlib will additionally try to read the registry entry

```
HKLM\SOFTWARE\PDFlib\PDFlib\5.0.2\resourcefile
```

The value of this entry (which will be created by the PDFlib installer, but can also be created by other means) will be taken as the name of the resource file to be used. If this file cannot be read an exception will be thrown.

- The client can force PDFlib to read a resource file at runtime by explicitly setting the *resourcefile* parameter:

```
PDF_set_parameter(p, "resourcefile", "/path/to/pdflib.upr");
```

This call can be repeated arbitrarily often; the resource entries will be accumulated.

Configuring resources at runtime. In addition to using a UPR file for the configuration, it is also possible to directly configure individual resources within the source code via the *PDF_set_parameter( )* function. This function takes a category name and a corresponding resource entry as it would appear in the respective section of this category in a UPR resource file, for example:

```
PDF_set_parameter(p, "FontAFM", "Foobar-Bold=foobb___.afm")
PDF_set_parameter(p, "FontOutline", "Foobar-Bold=foobb___.pfa")
```

3.1.7 Generating PDF Documents in Memory

In addition to generating PDF documents on a file, PDFlib can also be instructed to generate the PDF directly in memory *(in-core).* This technique offers performance benefits since no disk-based I/O is involved, and the PDF document can, for example, directly be streamed via HTTP. Webmasters will be especially happy to hear that their server will not be cluttered with temporary PDF files. Unix users can write the generated PDF to the *stdout* channel and consume it in a pipe process by supplying »–« as filename for *PDF_open_file( )*.

You may, at your option, periodically collect partial data (e.g., every time a page has been finished), or fetch the complete PDF document in one big chunk at the end (after *PDF_close( )*). Interleaving production and consumption of the PDF data has several advantages. Firstly, since not all data must be kept in memory, the memory requirements are reduced. Secondly, such a scheme can boost performance since the first chunk of data can be transmitted over a slow link while the next chunk is still being generated. However, the total length of the generated data will only be known when the complete document is finished.

The active in-core PDF generation interface. In order to generate PDF data in memory, simply supply an empty filename to *PDF_open_file( ),* and retrieve the data with *PDF_get_buffer( )*:

```
PDF_open_file(p, "")
...create document...
PDF_close(p);

buf = PDF_get_buffer(p, &size);
```

```
... use the PDF data contained in the buffer ...
PDF_delete(p);
```

Note The PDF data in the buffer must be treated as binary data.

This is considered »active« mode since the client decides when he wishes to fetch the buffer contents. Active mode is available for all supported language bindings.

Note C and C++ clients must not free the returned buffer.

The passive in-core PDF generation interface. In »passive« mode, which is only available in the C and C++ language bindings, the user installs (via *PDF_open_mem( ))* a callback function which will be called at unpredictable times by PDFlib whenever PDF data is waiting to be consumed. Timing and buffer size constraints related to flushing (transferring the PDF data from the library to the client) can be configured by the client in order to provide for maximum flexibility. Depending on the environment, it may be advantageous to fetch the complete PDF document at once, in multiple chunks, or in many small segments in order to prevent PDFlib from increasing the internal document buffer. The flushing strategy can be set using *PDF_set_parameter( )* and the *flush* parameter values detailed in Table 3.4.

Table 3.4 Controlling PDFlib's flushing strategy with the flush parameter

flush parameter	*flushing strategy*	*benefits*
none	*flush only once at the end of the document*	*complete PDF document can be fetched by the client in one chunk*
page	*flush at the end of each page*	*generating and fetching pages can be nicely interleaved*
content	*flush after all fonts, images, file attachments, and pages*	*even better interleaving, since large items won't clog the buffer*
heavy	*always flush when the internal 64 KB document buffer is full*	*PDFlib's internal buffer will never grow beyond a fixed size*

3.1.8 Using PDFlib on EBCDIC-based Platforms

The operators and structure elements in the PDF file format are based on ASCII, making it difficult to mix text output and PDF operators on EBCDIC-based platforms such as IBM eServer iSeries 400 and zSeries S/390. However, a special mainframe version of PDFlib has been carefully crafted in order to allow mixing of ASCII-based PDF operators and EBCDIC (or other) text output. The EBCDIC-safe version of PDFlib is available for various operating systems and machine architectures.

In order to leverage PDFlib's features on EBCDIC-based platforms the following items are expected to be supplied in EBCDIC text format (more specifically, in code page 037 on iSeries, and code page 1047 on zSeries):

- PFA font files, UPR configuration files, AFM font metrics files
- encoding and code page files
- string parameters to PDFlib functions
- input and output file names
- environment variables (if supported by the runtime environment)
- PDFlib error messages will also be generated in EBCDIC format (except in Java).

If you prefer to use input text files (PFA, UPR, AFM, encodings) in ASCII format you can set the *asciifile* parameter to *true* (default is *false)*. PDFlib will then expect these files in ASCII encoding. String parameters will still be expected in EBCDIC encoding, however.

In contrast, the following items must always be treated in binary mode (i.e., any conversion must be avoided):

- PDF input and output files
- PFB font outline and PFM font metrics files
- TrueType and OpenType font files
- image files and ICC profiles

3.2 Page Descriptions

3.2.1 Coordinate Systems

PDF's default coordinate system is used within PDFlib. The default coordinate system (or default *user space)* has the origin in the lower left corner of the page, and uses the DTP point as unit:

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

The first coordinate increases to the right, the second coordinate increases upwards. PDFlib client programs may change the default user space by rotating, scaling, translating, or skewing, resulting in new user coordinates. The respective functions for these transformations are *PDF_rotate( )*, *PDF_scale( )*, *PDF_translate( ),* and *PDF_skew( )*. If the user space has been transformed, all coordinates in graphics and text functions must be supplied according to the new coordinate system. The coordinate system is reset to the default coordinate system at the start of each page.

Using metric coordinates. Metric coordinates can easily be used by scaling the coordinate system. The scaling factor is derived from the definition of the DTP point given above:

```
PDF_scale(p, 28.3465, 28.3465);
```

After this call PDFlib will interpret all coordinates (except for hypertext features, see below) in centimeters since *72/2.54 = 28.3465*.

Coordinates for hypertext elements. PDF always expects coordinates for hypertext functions, such as the rectangle coordinates for creating text annotations, links, and file annotations in the default coordinate system, and not in the (possibly transformed) user coordinate system. Since this is very cumbersome PDFlib offers automatic conversion of user coordinates to the format expected by PDF. This automatic conversion is activated by setting the *usercoordinates* parameter to *true:*

```
PDF_set_parameter(p, "usercoordinates", "true");
```

Since PDF supports only hypertext rectangles with edges parallel to the page edges, the supplied rectangles must be modified when the coordinate system has been transformed by scaling, rotating, translating, or skewing it. In this case PDFlib will calculate the smallest enclosing rectangle with edges parallel to the page edges, transform it to default coordinates, and use the resulting values instead of the supplied coordinates.

The overall effect is that you can use the same coordinate systems for both page content and hypertext elements when the *usercoordinates* parameter has been set to *true*.

Visualizing coordinates. In order to assist PDFlib users in working with PDF's coordinate system, the PDFlib distribution contains the PDF file *grid.pdf* which visualizes the coordinates for several common page sizes. Printing the appropriately sized page on transparent material may provide a useful tool for preparing PDFlib development.

Acrobat 5/6 (full version only, not the free Reader) also has a helpful facility. Simply choose *Window, Info* to display a measurement palette which uses points as units. Note

that the coordinates displayed refer to an origin in the top left corner of the page, and not PDF's default origin in the lower left corner.

Don't be mislead by PDF printouts which seem to experience wrong page dimensions. These may be wrong because of some common reasons:

- The *Shrink oversized pages to paper size* option has been checked in Acrobat's print dialog, resulting in scaled print output.
- Non-PostScript printer drivers are not always able to retain the exact size of printed objects.

Rotating objects. It is important to understand that objects cannot be modified once they have been drawn on the page. Although there are PDFlib functions for rotating, translating, scaling, and skewing the coordinate system, these do not affect existing objects on the page but only subsequently drawn objects. Rotating text, images, and imported PDF pages is easily accomplished with the *rotate* option of the *PDF_fit_textline( ), PDF_fit_image( ),* and *PDF_fit_pdi_page( )* functions.

The following example generates some horizontal text, and rotates the coordinate system in order to show rotated text. The save/restore nesting makes it easy to continue with horizontal text in the original coordinate system after the vertical text is done:

```
PDF_set_text_pos(p, 50, 600);
PDF_show(p, "This is horizontal text");
textx = PDF_get_value(p, "textx", 0);       /* determine text position*/
texty = PDF_get_value(p, "texty", 0);       /* determine text position */

PDF_save(p);
        PDF_translate(p, textx, texty);     /* move origin to end of text */
        PDF_rotate(p, 45);                  /* rotate coordinates */
        PDF_set_text_pos(p, 18, 0);         /* provide for distance from horiz. text */
        PDF_show(p, "rotated text");
PDF_restore(p);

PDF_continue_text(p, "horizontal text continues");
```

Using top-down coordinates. Unlike PDF's bottom-up coordinate system some graphics environments use top-down coordinates which may be preferred by some developers. Such a coordinate system can easily be established using PDFlib's transformation functions. However, since the transformations will also affect text output additional calls are required in order to avoid text being displayed in a mirrored sense.

In order to facilitate the use of top-down coordinates PDFlib supports a special mode in which all relevant coordinates will be interpreted differently: instead of working with the default PDF coordinate system, with the origin *(0, 0)* at the lower left corner of the page and *y* coordinates increasing upwards, a modified coordinate system will be used which has its origin at the upper left corner of the page with *y* coordinates increasing downwards. This top-down coordinate system can be activated with the *topdown* parameter:

```
PDF_set_parameter(p, "topdown", "true")
```

A different coordinate system can be established for each page, but the *topdown* parameter must not be set within a page description (but only between pages). The *topdown* feature has been designed to make it quite natural for PDFlib users to work in a top-

down coordinate system. For the sake of completeness we'll list the detailed consequences of establishing a top-down coordinate system below.

»Absolute« coordinates will be interpreted in the user coordinate system without any modification:

- All function parameters which are designated as »coordinates« in the function descriptions. Some examples: *x, y* in *PDF_moveto( ); x, y in PDF_circle( ), x, y* (but not *width* and *height!) in PDF_rect( ); llx, lly, urx, ury in PDF_add_note( )*).

»Relative« coordinate values will be modified internally to match the top-down system:

- Text (with positive font size) will be oriented towards the top of the page;
- When the manual talks about »lower left« corner of a rectangle, box etc. this will be interpreted as you see it on the page;
- When a rotation angle is specified the center of the rotation is still the origin *(0, 0)* of the user coordinate system. The visual result of a clockwise rotation will still be clockwise.

3.2.2 Page Sizes and Coordinate Limits

Standard page formats. For the convenience of PDFlib users, Table 3.5 lists common standard page sizes[1].

Table 3.5 Common standard page size dimensions in points

format	width	height	format	width	height	format	width	height
A0	2380	3368	A4	595	842	letter	612	792
A1	1684	2380	A5	421	595	legal	612	1008
A2	1190	1684	A6	297	421	ledger	1224	792
A3	842	1190	B5	501	709	11 x 17	792	1224

The PDFlib header file *pdflib.h* provides macro definitions for page width and height values for the most common page formats. These may be used in calls to *PDF_begin_page( )*. They are called *<format>_width*, *<format>_height*, where *<format>* is one of the formats in Table 3.5 (in lowercase).

Page size limits. Although PDF and PDFlib don't impose any restrictions on the usable page size, Acrobat implementations suffer from architectural limits regarding the page size. Note that other PDF interpreters may well be able to deal with larger or smaller document formats. PDFlib will throw an exception if Acrobat's page size limits are exceeded. The page size limits for Acrobat are shown in Table 3.6.

Table 3.6 Minimum and maximum page size of Acrobat

PDF viewer	minimum page size	maximum page size
Acrobat 4 and above	1/24" = 3 pt = 0.106 cm	200" = 14400 pt = 508 cm

Different page size boxes. While many PDFlib developers only specify the width and height of a page, some advanced applications (especially for prepress work) may want to specify one or more of PDF's additional box entries. PDFlib supports all of PDF's box

1. More information about ISO, Japanese, and U.S. standard formats can be found at the following URLs: http://www.twics.com/~eds/paper/papersize.html, http://www.cl.cam.ac.uk/~mgk25/iso-paper.html

entries. The following entries, which may be useful in certain environments, can be specified by PDFlib clients (definitions taken from the PDF reference):

- MediaBox: this is used to specify the width and height of a page, and describes what we usually consider the page size.
- CropBox: the region to which the page contents are to be clipped; Acrobat uses this size for screen display and printing.
- TrimBox: the intended dimensions of the finished (possibly cropped) page;
- ArtBox: extent of the page's meaningful content. It is rarely used by application software;
- BleedBox: the region to which the page contents are to be clipped when output in a production environment. It may encompass additional bleed areas to account for inaccuracies in the production process.

PDFlib will not use any of these values apart from recording it in the output file. By default PDFlib generates a MediaBox according to the specified width and height of the page, but does not generate any of the other entries. The following code fragment will start a new page and set the four values of the CropBox:

```
PDF_begin_page(p, 595, 842);            /* start a new page */
PDF_set_value(p, "CropBox/llx", 10);
PDF_set_value(p, "CropBox/lly", 10);
PDF_set_value(p, "CropBox/urx", 500);
PDF_set_value(p, "CropBox/ury", 800);
```

Number of pages in a document. There is no limit in PDFlib regarding the number of generated pages in a document. PDFlib generates PDF structures which allow Acrobat to efficiently navigate documents with hundreds of thousands of pages.

Output accuracy and coordinate range. PDFlib's numerical output accuracy has been carefully chosen to match the requirements of PDF and the supported environments, while at the same time minimizing output file size. As detailed in Table 3.7 PDFlib's accuracy depends on the absolute value of coordinates. While most developers may safely ignore this issue, demanding applications should take care in their scaling operations in order to not exceed PDF's built-in coordinate limits.

Table 3.7 Output accuracy and coordinate range

absolute value	*output*
0 ... 0.000015	*0*
0.000015 ... 32767.999999	*rounded to four decimal digits*
32768 ... 2^{31} - 1	*rounded to next integer*
>= 2^{31}	*an exception will be raised*

3.2.3 Paths

A path is a shape made of an arbitrary number of straight lines, rectangles, or curves. A path may consist of several disconnected sections, called subpaths. There are several operations which can be applied to a path (see Section 7.4.6, »Path Painting and Clipping«, page 180):

- Stroking draws a line along the path, using client-supplied parameters (e.g., color, line width) for drawing.

- Filling paints the entire region enclosed by the path, using client-supplied parameters for filling.
- Clipping reduces the imageable area for subsequent drawing operations by replacing the current clipping area (which is the page size by default) with the intersection of the current clipping area and the area enclosed by the path.
- Merely terminating the path results in an invisible path, which will nevertheless be present in the PDF file. This will only rarely be required.

It is an error to construct a path without applying any of the above operations to it. PDFlib's scoping system ensures that clients obey to this restriction. These rules may easily be summarized as »don't change the appearance within a path description«.

Merely constructing a path doesn't result in anything showing up on the page; you must either fill or stroke the path in order to get visible results:

```
PDF_moveto(p, 100, 100);
PDF_lineto(p, 200, 100);
PDF_stroke(p);
```

Most graphics functions make use of the concept of a current point, which can be thought of as the location of the pen used for drawing.

3.2.4 Templates

Templates in PDF. PDFlib supports a PDF feature with the technical name *form XObjects*. However, since this term conflicts with interactive forms we refer to this feature as *templates*. A PDFlib template can be thought of as an off-page buffer into which text, vector, and image operations are redirected (instead of acting on a regular page). After the template is finished it can be used much like a raster image, and placed an arbitrary number of times on arbitrary pages. Like images, templates can be subjected to geometrical transformations such as scaling or skewing. When a template is used on multiple pages (or multiply on the same page), the actual PDF operators for constructing the template are only included once in the PDF file, thereby saving PDF output file size. Templates suggest themselves for elements which appear repeatedly on several pages, such as a constant background, a company logo, or graphical elements emitted by CAD and geographical mapping software. Other typical examples for template usage include crop and registration marks or custom Asian glyphs.

Using templates with PDFlib. Templates can only be *defined* outside of a page description, and can be *used* within a page description. However, templates may also contain other templates. Obviously, using a template within its own definition is not possible. Referring to an already defined template on a page is achieved with the *PDF_fit_image()* function just like images are placed on the page (see Section 5.3, »Placing Images and Imported PDF Pages«, page 121). The general template idiom in PDFlib looks as follows:

```
/* define the template */
template = PDF_begin_template(p, template_width, template_height);
...place marks on the template using text, vector, and image functions...
PDF_end_template(p);
...
PDF_begin_page(p, page_width, page_height);
/* use the template */
PDF_fit_image(p, template, (float) 0.0, (float) 0.0, "");
```

```
...more page marking operations...
PDF_end_page(p);
...
PDF_close_image(p, template);
```

All text, graphics, and color functions can be used on a template. However, the following functions must not be used while constructing a template:

- The functions in Section 7.6, »Image and Template Functions«, page 188, except *PDF_place_image( ), PDF_fit_image( ),* and *PDF_close_image( )*. This is not a big restriction since images can be opened outside of a template definition, and freely be used within a template (but not opened).
- The functions in Section 7.9, »Hypertext Functions«, page 204. Hypertext elements must always be defined on the page where they should appear in the document, and cannot be generated as part of a template.

Template support in third-party software. Templates (form XObjects) are an integral part of the PDF specification, and can be perfectly viewed and printed with Acrobat. However, not all PDF consumers are prepared to deal with this construct. For example, the Acrobat plugin Enfocus PitStop 5.0 can only move templates, but cannot access individual elements within a template. On the other hand, Adobe Illustrator 9 and 10 fully support templates.

3.3 Working with Color

3.3.1 Color and Color Spaces

PDFlib clients may specify the colors used for filling and stroking the interior of paths and text characters. Colors may be specified in several color spaces:

- Gray values between 0=black and 1=white;
- RGB triples, i.e., three values between 0 and 1 specifying the percentage of red, green, and blue; (0, 0, 0)=black, (1, 1, 1)=white;
- Four CMYK values between 0=no color and 1=full color, representing cyan, magenta, yellow, and black values; (0, 0, 0, 0)=white, (0, 0, 0, 1)=black. Note that this is different from the RGB specification.
- Device-independent colors in the CIE L*a*b* color space are specified by a luminance value and two color values (see Section 3.3.6, »Device-Independent CIE L*a*b* Color«, page 65).
- ICC-based colors are specified with the help of an ICC profile (see Section 3.3.4, »Color Management and ICC Profiles«, page 63).
- Spot color (separation color space): a predefined or arbitrarily named custom color with an alternate representation in one of the other color spaces above; this is generally used for preparing documents which are intended to be printed on an offset printing machine with one or more custom colors. The tint value (percentage) ranges from 0=no color to 1=maximum intensity of the spot color. See Section 3.3.3, »Spot Colors«, page 60, for a list of spot color names.
- Patterns: tiling with an object composed of arbitrary text, vector, or image graphics (see Section 3.3.2, »Patterns and Smooth Shadings«, page 59).
- Shadings (smooth blends) provide a gradual transition between two colors, and are based on another color space (see Section 3.3.2, »Patterns and Smooth Shadings«, page 59).
- The indexed color space is a not really a color space on its own, but rather an efficient coding of another color space. It will automatically be generated when an indexed (palette-based) image is imported.

The default color for stroke and fill operations is black.

3.3.2 Patterns and Smooth Shadings

As an alternative to solid colors, patterns and shadings are special kinds of colors which can be used to fill or stroke arbitrary objects.

Patterns. A pattern is defined by an arbitrary number of painting operations which are grouped into a single entity. This group of objects can be used to fill or stroke arbitrary other objects by replicating (or tiling) the group over the entire area to be filled or the path to be stroked. Working with patterns involves the following steps:

- First, the pattern must be defined between *PDF_begin_pattern( )* and *PDF_end_pattern( )*. Most graphics operators can be used to define a pattern.
- The pattern handle returned by *PDF_begin_pattern( )* can be used to set the pattern as the current color using *PDF_setcolor( )*.

Depending on the *painttype* parameter of *PDF_begin_pattern( )* the pattern definition may or may not include its own color specification. If *painttype* is 1, the pattern defini-

tion must contain its own color specification and will always look the same; if *painttype* is 2, the pattern definition must not include any color specification. Instead, the current fill or stroke color will be applied when the pattern is used for filling or stroking.

Note Patterns can also be defined based on a smooth shading (see below).

Smooth shadings. Smooth shadings, also called color blends or gradients, provide a continuous transition from one color to another. Both colors must be specified in the same color space. PDFlib supports two different kinds of geometry for smooth shadings:

- axial shadings are defined along a line;
- radial shadings are defined between two circles.

Shadings are defined as a transition between two colors. The first color is always taken to be the current fill color; the second color is provided in the *c1, c2, c3,* and *c4* parameters of *PDF_shading( )*. These numerical values will be interpreted in the first color's color space according to the description of *PDF_setcolor( )*.

Calling *PDF_shading( )* will return a handle to a shading object which can be used in two ways:

- Fill an area with *PDF_shfill( )*. This method can be used when the geometry of the object to be filled is the same as the geometry of the shading. Contrary to its name this function will not only fill the interior of the object, but also affects the exterior. This behavior can be modified with *PDF_clip( )*.
- Define a shading pattern to be used for filling more complex objects. This involves calling *PDF_shading_pattern( )* to create a pattern based on the shading, and using this pattern to fill or stroke arbitrary objects.

3.3.3 Spot Colors

PDFlib supports spot colors (technically known as Separation color space in PDF, although the term separation is generally used with process colors, too) which can be used to print custom colors outside the range of colors mixed from process colors. Spot colors are specified by name, and in PDF are always accompanied by an alternate color which closely, but not exactly, resembles the spot color. Acrobat will use the alternate color for screen display and printing to devices which do not support spot colors (such as office printers). On the printing press the requested spot color will be applied in addition to any process colors which may be used in the document. This requires the PDF files to be post-processed by a process called color separation.

Note Color separation is outside the scope of PDFlib. Acrobat 6, additional software for Acrobat 5 (such as the ARTS PDF Crackerjack[1] plugin), or in-RIP separation is required to separate PDFs.

Note Some spot colors do not display correctly on screen in Acrobat 5 when Overprint Preview is turned on. They can be separated and printed correctly, though.

PDFlib supports various built-in spot color libraries as well as custom (user-defined) spot colors. When a spot color name is requested with *PDF_makespotcolor( )* PDFlib will first check whether the requested spot color can be found in one of its built-in libraries. If so, PDFlib will use built-in values for the alternate color. Otherwise the spot color is assumed to be a user-defined color, and the client must supply appropriate alternate color values (via the current color). By default, built-in spot colors can not be redefined

1. See http://www.artspdf.com

with custom alternate values. However, this behavior can be changed with the *spotcolorlookup* parameter. Spot colors can be tinted, i.e., they can be used with a percentage between 0 and 1.

PDFlib will automatically generate suitable alternate colors for built-in spot colors when a PDF/X conformance level has been selected (see Section 3.4, »PDF/X Support«, page 67). For custom spot colors it is the user's responsibility to provide alternate colors which are compatible with the selected PDF/X conformance level.

Note *Built-in spot color data and the corresponding trademarks have been licensed by PDFlib GmbH from the respective trademark owners for use in PDFlib software.*

PANTONE® colors. PANTONE Colors are well-known and widely used on a world-wide basis. PDFlib fully supports the PANTONE MATCHING SYSTEM®, totalling ca. 20 000 swatches. All color swatch names from the following digital color libraries can be used (sample swatch names are provided in parentheses):

- PANTONE solid coated *(PANTONE 185 C)*
- PANTONE solid uncoated *(PANTONE 185 U)*
- PANTONE solid matte *(PANTONE 185 M)*
- PANTONE process coated *(PANTONE DS 35-1 C)*
- PANTONE process uncoated *(PANTONE DS 35-1 U)*
- PANTONE process coated EURO *(PANTONE DE 35-1 C)*
- PANTONE pastel coated *(PANTONE 9461 C)*
- PANTONE pastel uncoated *(PANTONE 9461 U)*
- PANTONE metallic coated *(PANTONE 871 C)*
- PANTONE solid to process coated *(PANTONE 185 PC)*
- PANTONE solid to process coated EURO *(PANTONE 185 EC)*
- PANTONE hexachrome® coated *(PANTONE H 305-1 C)*
- PANTONE hexachrome® uncoated *(PANTONE H 305-1 U)*
- PANTONE solid in hexachrome coated *(PANTONE 185 HC)*

Old color name prefixes CV, CVV, CVU, CVC, and CVP will also be accepted, and changed to the corresponding new color names unless the *preserveoldpantonenames* parameter is true. The PANTONE prefix must always be provided in the swatch name as shown in the examples. Generally, PANTONE Color names must be constructed according to the following scheme:

```
PANTONE <id> <paperstock>
```

where *<id>* is the identifier of the color (e.g., 185) and *<paperstock>* the abbreviation of the paper stock in use (e.g., *C* for coated). A single space character must be provided between all components constituting the swatch name. Requesting a spot color name starting with the PANTONE prefix where the name does not represent a valid PANTONE Color will results in a non-fatal exception (which can be disabled by setting the *warning* parameter to *false)*. The following code snippet demonstrates the use of a PANTONE Color with a tint value of 70 percent:

```
spot = PDF_makespotcolor(p, "PANTONE 281 U", 0);
PDF_setcolor(p, "fill", "spot", spot, 0.7, 0, 0);
```

Note PANTONE® Colors displayed here may not match PANTONE-identified standards. Consult current PANTONE Color Publications for accurate color. PANTONE® and other Pantone, Inc. trademarks are the property of Pantone, Inc. © Pantone, Inc., 2003.

Note PANTONE® Colors are currently not supported in the PDF/X-1 and PDF/X-1a modes.

HKS® colors. The HKS color system is widely used in Germany and other European countries. PDFlib fully supports HKS colors, including those from the new *HKS 3000 plus* palettes. All color swatch names from the following digital color libraries *(Farbfächer)* can be used (sample swatch names are provided in parentheses):

- HKS K *(Kunstdruckpapier)* for gloss art paper, 88 colors *(HKS 43 K)*
- HKS N *(Naturpapier)* for natural paper, 88 colors *(HKS 43 N)*
- HKS E *(Endlospapier)* for continuous stationary/coated, 90 colors *(HKS 43 E)*
- HKS Ek *(Endlospapier)* for continuous stationary/uncoated, 88 colors *(HKS 43 E)*
- HKS En: identical to *HKS E* *(HKS 43 En)*
- HKS Z *(Zeitungspapier)* for newsprint, 50 colors *(HKS 43 Z)*

The HKS prefix must always be provided in the swatch name as shown in the examples. Generally, HKS color names must be constructed according to one of the following schemes:

```
HKS <id> <paperstock>
```

where *<id>* is the identifier of the color (e.g., 43) and *<paperstock>* the abbreviation of the paper stock in use (e.g., *N* for natural paper). A single space character must be provided between the *HKS*, *<id>*, and *<paperstock>* components constituting the swatch name. Requesting a spot color name starting with the HKS prefix where the name does not represent a valid HKS color results in a non-fatal exception (which can be disabled by setting the *warning* parameter to *false)*. The following code snippet demonstrates the use of an HKS color with a tint value of 70 percent:

```
spot = PDF_makespotcolor(p, "HKS 38 E", 0);
PDF_setcolor(p, "fill", "spot", spot, 0.7, 0, 0);
```

User-defined spot colors. In addition to built-in spot colors as detailed above, PDFlib supports custom spot colors. These can be assigned an arbitrary name (which must not conflict with the name of any built-in color, however) and an alternate color which will be used for screen preview or low-quality printing, but not for high-quality color separations. The client is responsible for providing suitable alternate colors for custom spot colors.

There is no separate PDFlib function for setting the alternate color for a new spot color; instead, the current fill color will be used. Except for an additional call to set the alternate color, defining and using custom spot colors works similarly to using built-in spot colors:

```
PDF_setcolor(p, "fill", "cmyk", 0.2, 1.0, 0.2, 0);  /* define alternate CMYK values */
spot = PDF_makespotcolor(p, "CompanyLogo", 0);      /* derive a spot color from it */
PDF_setcolor(p, "fill", "spot", spot, 1, 0, 0);     /* set the spot color */
```

3.3.4 Color Management and ICC Profiles

The International Color Consortium (ICC)[1] defined a file format for specifying color characteristics of input and output devices. These ICC color profiles are considered an industry standard, and are supported by all major color management system and application vendors. PDFlib supports color management with ICC profiles in the following areas:

- Define ICC-based color spaces for text and vector graphics on the page.
- Process ICC profiles embedded in imported image files.
- Apply an ICC profile to an imported image (possibly overriding an ICC profile embedded in the image).
- Define default color spaces for mapping grayscale, RGB, or CMYK data to ICC-based color spaces.
- Define a PDF/X output intent by means of an external ICC profile.

Color management does not change the number of components in a color specification (e.g., from RGB to CMYK).

Searching for ICC profiles. PDFlib will search for ICC profiles according to the following steps, using the *profilename* parameter supplied to *PDF_load_iccprofile( )*:

- If *profilename = sRGB*, PDFlib will use its internal sRGB profile (see below), and terminate the search.
- Check whether there is a resource named *profilename* in the *ICCProfile* resource category. If so, use its value as file name in the following steps. If there is no such resource, use *profilename* as a file name directly.
- Use the file name determined in the previous step to locate a disk file by trying the following combinations one after another:

```
<filename>
<filename>.icc
<filename>.icm
<colordir>/<filename>
<colordir>/<filename>.icc
<colordir>/<filename>.icm
```

On Windows 2000/XP *colordir* designates the directory where device-specific ICC profiles are stored by the operating system (typically *C:\WINNT\system32\spool\drivers\color)*. On Mac OS X the following paths will be tried for *colordir:*

```
/System/Library/ColorSync/Profiles
/Library/ColorSync/Profiles
/Network/Library/ColorSync/Profiles
~/Library/ColorSync/Profiles
```

On other systems the steps involving *colordir* will be omitted.

Acceptable ICC profiles. The type of acceptable ICC profiles depends on the *usage* parameter supplied to *PDF_load_iccprofile( )*:

- If *usage = outputintent,* only output device (printer) profiles will be accepted.

1. See http://www.color.org

- If *usage = iccbased,* input, display and output device (scanner, monitor, and printer) profiles plus color space conversion profiles will be accepted. They may be specified in the gray, RGB, CMYK, or Lab color spaces.

The sRGB color space and sRGB ICC profile. PDFlib supports the industry-standard RGB color space called sRGB (formally IEC 61966-2-1)[1]. sRGB is supported by a variety of software and hardware vendors and is widely used for simplified color management for consumer RGB devices such as digital still cameras, office equipment such as color printers, and monitors. PDFlib supports the sRGB color space and includes the required ICC profile data internally. Therefore an sRGB profile must not be configured explicitly by the client, but it is always available without any additional configuration. It can be requested by calling *PDF_load_iccprofile( )* with *profilename = sRGB*.

3.3.5 Working with ICC Profiles

Using embedded profiles in images (ICC-tagged images). Some images may contain embedded ICC profiles describing the nature of the image's color values. For example, an embedded ICC profile can describe the color characteristics of the scanner used to produce the image data. PDFlib can handle embedded ICC profiles in the PNG, JPEG, and TIFF image file formats. If the *honoriccprofile* option or parameter is set to true (which is the default) the ICC profile embedded in an image will be extracted from the image, and embedded in the PDF output such that Acrobat will apply it to the image. This process is sometimes referred to as tagging an image with an ICC profile. PDFlib will not alter the image's pixel values.

The *image:iccprofile* parameter can be used to obtain an ICC profile handle for the profile embedded in an image. This may be useful when the same profile shall be applied to multiple images.

In order to check the number of color components in an unknown ICC profile use the *icccomponents* parameter.

Applying external ICC profiles to images (tagging). As an alternative to using ICC profiles embedded in an image, an external profile may be applied to an individual image by supplying a profile handle along with the *iccprofile* option to *PDF_load_image( )*.

In order to apply certain ICC profiles to all images, the *image:iccprofile* parameter can be used. As opposed to setting default color spaces (see below) these parameters affect only images, but not text and vector graphics.

ICC-based color spaces for page descriptions. The color values for text and vector graphics can directly be specified in the ICC-based color space specified by a profile. The color space must first be set by supplying the ICC profile handle as value to one of the *setcolor:iccprofilegray, setcolor:iccprofilergb, setcolor:iccprofilecmyk* parameters. Subsequently ICC-based color values can be supplied to *PDF_setcolor( )* along with one of the color space keywords *iccbasedgray*, *iccbasedrgb*, or *iccbasedcmyk:*

```
icchandle = PDF_load_iccprofile(...)
if (icchandle == -1) {
        return;
}
```

1. See http://www.srgb.com

```
PDF_set_value(p, "setcolor:iccprofilecmyk", icchandle);
PDF_setcolor(p, "fill", "iccbasedcmyk", 0, 1, 0, 0);
```

Mapping device colors to ICC-based default color spaces. PDF provides a feature for mapping device-dependent gray, RGB, or CMYK colors in a page description to device-independent colors. This can be used to attach a precise colorimetric specification to color values which otherwise would be device-dependent. Mapping color values this way is accomplished by supplying a DefaultGray, DefaultRGB, or DefaultCMYK color space definition. In PDFlib it can be achieved by setting the *defaultgray, defaultrgb,* or *defaultcmyk* parameters and supplying an ICC profile handle as the corresponding value. The following examples will set the sRGB color space as the default RGB color space for text, images, and vector graphics:

```
icchandle = PDF_load_iccprofile(p, "sRGB", 0, "usage=iccbased");
if (icchandle == -1) {
        return;
}
PDF_set_value(p, "defaultrgb", icchandle);
```

Defining PDF/X output intents. An output device (printer) profile can be used to specify an output condition for PDF/X. This is done by supplying *usage = outputintent* in the call to *PDF_load_iccprofile( )*. For details see Section 3.4.1, »Generating PDF/X-conforming Output«, page 67.

3.3.6 Device-Independent CIE L*a*b* Color

Device-independent color values can be specified in the CIE 1976 L*a*b* color space by supplying the color space name *lab* to *PDF_setcolor( )*. Colors in the L*a*b* color space are specified by a luminance value in the range 0-100, and two color values in the range -127 to 128. The illuminant used for the *lab* color space will be D50 (daylight 5000 K, 2° observer)

3.3.7 Rendering Intents

Although PDFlib clients can specify device-independent color values, a particular output device is not necessarily capable of accurately reproducing the required colors. In this situation some compromises have to be made regarding the trade-offs in a process called gamut compression, i.e., reducing the range of colors to a smaller range which can be reproduced by a particular device. The rendering intent can be used to control this process. Rendering intents can be specified for individual images by supplying the *renderingintent* parameter or option to *PDF_load_image( )*. In addition, rendering intents can be specified for text and vector graphics by supplying the *renderingintent* option to *PDF_create_gstate( )*. Table 3.8 lists the available rendering intents and their meanings.

Table 3.8 Rendering intents

rendering intent	*explanation*	*typical use*
Auto	*Do not specify any rendering intent in the PDF file, but use the device's default intent instead. This is the default.*	*unknown or unspecific uses*
AbsoluteColorimetric	*No correction for the device's white point (such as paper white) is made. Colors which are out of gamut are mapped to nearest value within the device's gamut.*	*exact reproduction of solid colors; not recommended for other uses.*

Table 3.8 Rendering intents

rendering intent	***explanation***	***typical use***
RelativeColorimetric	*The color data is scaled into the device's gamut, mapping the white points onto one another while slightly shifting colors.*	*vector graphics*
Saturation	*Saturation of the colors will be preserved while the color values may be shifted.*	*business graphics*
Perceptual	*Color relationships are preserved by modifying both in-gamut and out-of-gamut colors in order to provide a pleasing appearance.*	*scanned images*

3.4 PDF/X Support

The PDF/X standards series strives to provide a consistent and robust subset of PDF which can be used to deliver data suitable for commercial printing. PDFlib can generate output conforming to the following flavors of PDF/X:

- PDF/X-1 and PDF/X-1a, both defined in ISO 15930-1:2001
- PDF/X-3 as defined in ISO 15930-3:2002

PDFlib will support PDF/X-2 (ISO 15930-2) as soon as this standard has been finalized.

Note PANTONE® Colors are currently not supported in the PDF/X-1 and PDF/X-1a modes.

3.4.1 Generating PDF/X-conforming Output

Creating PDF/X-conforming output with PDFlib is achieved by the following means:

- PDFlib will automatically take care of several formal settings for PDF/X, such as PDF version number and PDF/X conformance keys.
- The PDFlib client must explicitly use certain function calls or parameter settings as detailed in Table 3.9.
- The PDFlib client must refrain from using certain function calls and parameter settings as detailed in Table 3.10.
- Additional rules apply when importing pages from existing PDF/X-conforming documents (see Section 3.4.2, »Importing PDF/X Documents with PDI«, page 69).

Required operations. Table 3.9 lists all operations required to generate PDF/X-compatible output. The items apply to all PDF/X conformance levels unless otherwise noted. Not calling one of the required functions while in PDF/X mode will trigger an exception.

Table 3.9 Operations which must be applied for PDF/X compatibility

Item	*PDFlib function and parameter requirements for PDF/X compatibility*
conformance level	*The pdfx parameter must be set to the required PDF/X conformance level before calling PDF_open_file( ).*
output condition (output intent)	*PDF_load_iccprofile( ) with usage = outputintent or PDF_process_pdi( ) with action = copyoutputintent must be called exactly once for each document. If spot colors from one of the built-in color libraries are used an output intent ICC profile must be embedded (using a standard output condition is not allowed in these cases).* *PDF/X-1 and PDF/X-1a: the output device must be a monochrome or CMYK device;* *PDF/X-3: the output device must be a monochrome, RGB, or CMYK device. If ICC-based colors or Lab colors are used in the file, an output device ICC profile must be embedded.*
font embedding	*Set the embedding option of PDF_load_font( ) to true to enable font embedding.*
page sizes	*The page boxes, which are settable via the CropBox, BleedBox, TrimBox, and ArtBox parameters, must satisfy all of the following requirements:* ► *The TrimBox or ArtBox must be set, but not both of these box entries. If both TrimBox and ArtBox are missing PDFlib will take the CropBox (if present) as the TrimBox, and the MediaBox if the CropBox is also missing.* ► *The BleedBox, if present, must fully contain the ArtBox and TrimBox.* ► *The CropBox, if present, must fully contain the ArtBox and TrimBox.*
grayscale color	*PDF/X-3: the defaultgray parameter must be set if grayscale images are used or PDF_setcolor( ) is used with a gray color space, and the PDF/X output condition is not a CMYK or grayscale device.*
RGB color	*PDF/X-3: the defaultrgb parameter must be set if RGB images are used or PDF_setcolor( ) is used with an RGB color space, and the PDF/X output condition is not an RGB device.*

Table 3.9 Operations which must be applied for PDF/X compatibility

Item	PDFlib function and parameter requirements for PDF/X compatibility
CMYK color	PDF/X-3: the defaultcmyk parameter must be set if cmyk images are used or PDF_setcolor( ) is used with a CMYK color space, and the PDF/X output condition is not a CMYK device.
document info keys	The Creator and Title info keys must be set with PDF_set_info( ).

Prohibited operations. Table 3.10 lists all operations which are prohibited when generating PDF/X-compatible output. The items apply to all PDF/X conformance levels unless otherwise noted. Calling one of the prohibited functions while in PDF/X mode will trigger an exception. However, images with unacceptable compression (GIF and LZW-compressed TIFF images) will not result in an exception subject to the *imagewarning* parameter. Similarly, if an imported PDF page doesn't match the current PDF/X conformance level, the corresponding PDI call will fail without an exception (subject to the *pdiwarning* parameter).

Table 3.10 Operations which must be avoided to achieve PDF/X compatibility

Item	PDFlib functions and parameters to be avoided for PDF/X compatibility
grayscale color	PDF/X-1: the defaultgray parameter must be avoided.
RGB color	PDF/X-1 and PDF/X-1a: RGB images and the defaultrgb parameter must be avoided.
CMYK color	PDF/X-1: the defaultcmyk parameter must be avoided.
ICC-based color	PDF/X-1 and PDF/X-1a: the iccbasedgray/rgb/cmyk color space in PDF_setcolor( ) and the setcolor:iccprofilegray/rgb/cmyk parameters must be avoided.
Lab color	PDF/X-1 and PDF/X-1a: the Lab color space in PDF_setcolor( ) must be avoided.
annotations	Annotations inside the BleedBox (or TrimBox/ArtBox if no BleedBox is present) must be avoided: PDF_attach_file( ), PDF_add_note( ), PDF_add_pdflink( ), PDF_add_locallink( ), PDF_add_launchlink( ), PDF_add_weblink( ).
images	GIF images and LZW-compressed TIFF images must be avoided. PDF/X-1 and PDF/X-1a: images with RGB, ICC-based, or Lab color must be avoided. For colorized images the alternate color of the spot color used must satisfy the same conditions.
document info keys	Trapped info key values other than True or False for PDF_set_info( ) must be avoided.
security	PDF/X-1: userpassword parameter and permissions parameter value noprint must be avoided; PDF/X-1a and PDF/X-3: userpassword, masterpassword, and permissions parameters must be avoided.
PDF version	Setting the PDF version number must be avoided since PDFlib will do this automatically. Since PDF/X-1, PDF/X-1a, and PDF/X-3 are based on PDF 1.3 operations that require PDF 1.4 or above (such as transparency settings or soft masks) must be avoided (see Table 1.3)
PDF import (PDI)	Imported documents must conform to the same PDF/X level as the output document, and must have been prepared according to the same output intent.

Standard output conditions. The output condition defines the intended target device, which is mainly useful for reliable proofing. The output intent can either be specified by an ICC profile or by supplying the name of a standard output intent. Table 3.11 lists the names of standard output intents known to PDFlib. Additional standard output intents can be defined using the *StandardOutputIntent* resource category (see Section 3.1.6, »Resource Configuration and File Searching«, page 47). It is the user's responsibility to

add only those names as standard output intents which can be recognized by PDF/X-processing software.

Table 3.11 Standard output intents for PDF/X

Output intent	*description*
CGATS TR 001	*SWOP (publication) printing in USA*
OF COM PO P1 F60	*ISO 12647-2, positive plates, paper type 1 (gloss-coated)*
OF COM PO P2 F60	*ISO 12647-2, positive plates, paper type 2 (matte-coated)*
OF COM PO-P3 F60[1]	*ISO 12647-2, positive plates, paper type 3 (light weight coated web)*
OF COM PO P4 F60	*ISO 12647-2, positive plates, paper type 4 (uncoated white offset)*
OF COM NE P1 F60	*ISO 12647-2, negative plates, paper type 1 (gloss-coated)*
OF COM NE P2 F60	*ISO 12647-2, negative plates, paper type 2 (matte-coated)*
OF COM NE P3 F60	*ISO 12647-2, negative plates, paper type 3 (light weight coated web)*
OF COM NE P4 F60	*ISO 12647-2, negative plates, paper type 4 (uncoated white offset)*
SC GC2 CO F30	*ISO 12647-5, gamut class 2, conventional UV or water-based air dried*
Ifra_NP_40lcm_neg+CTP_05.00	*Coldset offset (computer to plate)*

1. Although the dash character between Po and P3 may look inconsistent, it is actually required by the standard.

3.4.2 Importing PDF/X Documents with PDI

Special rules apply when pages from an existing PDF document will be imported into a PDF/X-conforming output document (see Section 5.2, »Importing PDF Pages with PDI (PDF Import Library)«, page 118, for details on the PDF import library PDI). All imported documents must conform to the same PDF/X conformance level as the generated output document. If a certain PDF/X conformance level is configured in PDFlib and the imported documents also adhere to this level, the generated output is guaranteed to comply with the same PDF/X conformance level. Imported documents which do not adhere to the chosen PDF/X level will be rejected.

If multiple PDF/X documents are imported, they must all have been prepared for the same output condition. While PDFlib can correct certain items, it is not intended to work as a full PDF/X validator or to enforce full PDF/X compatibility for imported documents. For example, PDFlib will not embed fonts which are missing from imported PDF pages, and does not apply any color correction to imported pages.

If you want to combine imported pages such that the resulting PDF output document conforms to the same PDF/X conformance level and output condition as the input document(s), you can query the PDF/X status of the imported PDF as follows:

```
pdfxlevel = PDF_get_pdi_parameter(p, "pdfx", doc, -1, 0, &len);
```

This statement will retrieve a string designating the PDF/X conformance level of the imported document if it conforms to an ISO PDF/X level, or *none* otherwise. The returned string can be used to set the PDF/X conformance level of the output document appropriately:

```
PDF_set_parameter(p, "pdfx", pdfxlevel);
```

In addition to querying the PDF/X conformance level you can also copy the PDF/X output intent from an imported document as follows:

```
doc = PDF_process_pdi(p, doc, -1, "action copyoutputintent");
```

This can be used as an alternative to setting the output intent via *PDF_load_iccprofile( )*, and will copy the imported document's output intent to the generated output document, regardless of whether it is defined by a standard name or an ICC profile. The output intent of the generated output document must be set exactly once, either by copying an imported document's output intent, or by setting it explicitly using *PDF_load_iccprofile( )* with the *usage* option set to *outputintent*.

3.5 Passwords and Permissions

3.5.1 Strengths and Weaknesses of PDF Security Features

PDF supports various security features which aid in protecting document contents. They are based on Acrobat's standard encryption handler which uses symmetric encryption. Both Acrobat Reader and the full Acrobat product support the following security features:

- Permissions restrict certain actions for the PDF document, such as printing or extracting text.
- The user password is required to open the file.
- The master password is required to change any security settings, i.e. permissions, user or master password. Files with user and master passwords can be opened for reading or printing with either password.

If a file has a user or master password or any permissions restrictions set, it will be encrypted.

Cracking protected PDF documents. The length of the encryption keys used for protecting documents depends on the PDF compatibility level chosen by the client:

- For PDF versions up to and including 1.3 (i.e., Acrobat 4) the key length is 40 bits.
- For PDF version 1.4 the key length is 128 bits. This requires Acrobat 5 or above.

It is widely known that a key length of 40 bits for symmetrical encryption (as used in PDF) is not secure. Actually, using commercially available cracking software it is possible to disable 40-bit PDF security settings with a brute-force attack within days or weeks, depending on the length and quality of the password. For maximum security we recommend the following:

- Use 128-bit encryption (i.e., PDF 1.4 compatibility setting) if at all possible. This requires Acrobat 5 or above for all users of the document.
- Passwords should be at least six characters long and should contain non-alphabetic characters. Passwords should definitely not resemble your spouse's or pet's name, your birthday etc. in order to prevent so-called dictionary attacks or password guessing. It is important to mention that even with 128-bit encryption short passwords can be cracked within minutes.

Access permissions. Setting some access restriction, such as *printing prohibited* will disable the respective function in Acrobat. However, this not necessarily holds true for third-party PDF viewers or other software. It is up to the developer of PDF tools whether or not access permissions will be honored. Indeed, several PDF tools are known to ignore permission settings altogether; commercially available PDF cracking tools can be used to disable any access restrictions. This has nothing to do with cracking the encryption; there is simply no way that a PDF file can make sure it won't be printed while it still remains viewable. This is actually documented in Adobe's own PDF reference:

There is nothing inherent in PDF encryption that enforces the document permissions specified in the encryption dictionary. It is up to the implementors of PDF viewers to respect the intent of the document creator by restricting user access to an encrypted PDF file according to the permissions contained in the file.

3.5.2 Protecting Documents with PDFlib

Passwords. Passwords can be set with the *userpassword* and *masterpassword* parameters. PDFlib interacts with the client-supplied passwords in the following ways:

- If a user password or permissions (see below), but no master password has been supplied, a regular user would be able to change the security settings. For this reason PDFlib considers this situation as an error.
- If user and master password are the same, a distinction between user and owner of the file would no longer be possible, again defeating effective protection. PDFlib considers this situation as an error.
- For both user and master passwords, up to a maximum of 32 characters will be used. Additional characters will be ignored, and do not affect encryption. Empty passwords are not allowed.

The supplied passwords will be used for all subsequently generated documents.

Permissions. Access restrictions can be set with the *permissions* parameter. It consists of a string with one or more access restrictions. Before setting the *permissions* parameter a master password must be set, because otherwise Acrobat users could easily remove the permission settings. By default, all actions are allowed. Specifying an access restriction will disable the respective feature in Acrobat. Access restrictions can be applied without any user password. The supplied permissions will be used for all subsequently generated documents. Multiple restriction keywords can be specified, separated with a blank character as in the following example:

```
PDF_set_parameter(p, "permissions", "noprint nocopy");
```

Table 3.12 lists all supported access restriction keywords. The *noforms, noaccessible, noassemble and nohiresprint* keywords require PDF 1.4 or above compatibility. They will be rejected otherwise.

Table 3.12 Access restriction keywords

keyword	*explanation*
noprint	*Acrobat will prevent printing the file.*
nomodify	*Acrobat will prevent users from adding form fields or making any other changes.*
nocopy	*Acrobat will prevent copying and extracting text or graphics, and will disable the accessibility interface*
noannots	*Acrobat will prevent adding or changing comments or form fields.*
noforms	*Acrobat will prevent form field filling, even if noannots hasn't been specified.*
noaccessible	*Acrobat will prevent extracting text or graphics for accessibility purposes (such as a screenreader program)*
noassemble	*Acrobat will prevent inserting, deleting, or rotating pages and creating bookmarks and thumbnails, even if nomodify hasn't been specified.*
nohiresprint	*Acrobat will prevent high-resolution printing. If noprint hasn't been specified printing is restricted to the »print as image« feature which prints a low-resolution rendition of the page.*

4 Text Handling

4.1 Overview of Fonts and Encodings

Font handling is one of the most complex aspects of page descriptions and document formats like PDF. In this section we will summarize PDFlib's main characteristics with regard to font and encoding handling (encoding refers to the mapping between individual bytes or byte combinations to the characters which they actually represent). Except where noted otherwise, PDFlib supports the same font formats on all platforms.

4.1.1 Supported Font Formats

PDFlib supports a variety of font types. This section summarizes the supported font types and notes some of the most important aspects of these formats.

PostScript Type 1 fonts. PostScript fonts can be packaged in various file formats, and are usually accompanied by a separate file containing metrics and other font-related information. PDFlib supports Mac and Windows PostScript fonts, and all common file formats for PostScript font outline and metrics data.

TrueType fonts. PDFlib supports vector-based TrueType fonts, but not those based on bitmaps. The TrueType font file must be supplied in Windows TTF or TTC format, or must be installed in the Mac or Windows operating system. Contrary to PostScript Type 1 fonts, TrueType and OpenType fonts do not require any additional metrics file since the metrics information is included in the font file itself.

OpenType fonts. OpenType is a modern font format which combines PostScript and TrueType technology, and uses a platform-independent file format. OpenType is natively supported on Windows 2000/XP, and Mac OS X. There are two flavors of OpenType fonts, both of which are supported by PDFlib:

- OpenType fonts with TrueType outlines *(*.ttf)* look and feel like usual TrueType fonts.
- OpenType fonts with PostScript outlines *(*.otf)* contain PostScript data in a TrueType-like file format. This flavor is also called CFF *(Compact Font Format)*.

Chinese, Japanese, and Korean (CJK) fonts. In addition to Acrobat's standard CJK fonts (see Section 4.7, »Chinese, Japanese, and Korean Text«, page 101), PDFlib supports custom CJK fonts in the TrueType and OpenType formats. Generally these fonts are treated similarly to Western fonts. However, certain restrictions apply.

Type 3 fonts. In addition to PostScript, TrueType, and OpenType fonts, PDFlib also supports the concept of user-defined (Type 3) PDF fonts. Unlike the common font formats, user-defined fonts are not fetched from an external source (font file or operating system services), but must be completely defined by the client by means of PDFlib's native text, graphics, and image functions. Type 3 fonts are useful for the following purposes:

- bitmap fonts,
- custom graphics, such as logos can easily be printed using simple text operators,

- Japanese gaiji (user-defined characters) which are not available in any predefined font or encoding.

4.1.2 Encodings

An encoding defines how the actual bytes in a string will be interpreted by PDFlib and Acrobat, and how they translate into text on a page. PDFlib supports a variety of encoding methods.

All supported encodings can be arbitrarily mixed in one document. You may even use different encodings for a single font, although the need to do so will only rarely arise.

Note *Not all encodings can be used with a given font. The user is responsible for making sure that the font contains all characters required by a particular encoding. This can even be problematic with Acrobat's core fonts (see Table 4.2).*

Identifying glyphs. There are three fundamentally different methods for identifying individual glyphs (representations of a character) in a font:

- PostScript Type 1 fonts are based on the concept of glyph names: each glyph is labelled with a unique name which can be used to identify the character, and construct code mappings which are suitable for a certain environment. While glyph names have served their purpose for quite some time they impose severe restrictions on modern computing because of their space requirements and because they do not really meet the requirements of international use (in particular CJK fonts).
- TrueType and OpenType fonts identify individual glyphs based on their Unicode values. This makes it easy to add clear semantics to all glyphs in a text font. However, there are no standard Unicode assignments for pi or symbol fonts. This implies some difficulties when using symbol fonts in a Unicode environment.
- Chinese, Japanese, and Korean OpenType fonts are based on the concept of Character IDs (CIDs). These are basically numbers which refer to a standard repository (called character complement) for the respective language.

There is considerable overlap among these concepts. For example, TrueType fonts may contain an auxiliary table of PostScript glyph names for compatibility reasons. On the other hand, Unicode semantics for many standard PostScript glyph names are available in the Adobe Glyph List (AGL). PDFlib supports all three methods (name-based, Unicode, CID).

8-Bit encodings. 8-bit encodings (also called single-byte encodings) map each byte in a text string to a single character, and are thus limited to 256 different characters at a time. 8-bit encodings used in PDFlib are based on glyph names or Unicode values, and can be drawn from various sources:

- A large number of predefined encodings according to Table 4.2. These cover the most important encodings currently in use on a variety of systems, and in a variety of locales.
- User-defined encodings which can be supplied in an external file or constructed dynamically at runtime with *PDF_encoding_set_char( )*. These encodings can be based on glyph names or Unicode values.
- Encodings pulled from the operating system, also known as *system encoding*. This feature is only available on Windows, IBM eServer iSeries, and zSeries.

- Abbreviated Unicode-based encodings which can be used to conveniently address any Unicode range of 256 consecutive characters with 8-bit values.
- Encodings specific to a particular font. These are also called *font-specific* or *builtin* encodings.

Wide-character addressing. In addition to 8-bit encodings, various other addressing schemes are supported which are much more powerful, and not subject to the 256 character limit.

- Purely Unicode-based addressing via the *unicode* encoding keyword. In this case the client directly supplies Unicode strings to PDFlib. The Unicode strings may be formatted according to one of several standard methods (such as UCS-2, UTF-8) and byte orderings (little-endian or big-endian).
- CMap-based addressing for a variety of Chinese, Japanese, and Korean standards. In combination with standard CJK fonts PDFlib supports all CMaps supported by Acrobat. This includes both Unicode-based CMaps and others (see Section 4.7, »Chinese, Japanese, and Korean Text«, page 101).
- Glyph id addressing for TrueType and OpenType fonts via the *glyphid* encoding keyword. This is useful for advanced text processing applications which need access to individual glyphs in a font without reference to any particular encoding scheme, or must address glyphs which do not have any Unicode mapping. The number of valid glyph ids in a font can be queried with the *fontmaxcode* parameter.

4.1.3 Support for the Unicode Standard

Unicode is a large character set which covers all current and many ancient languages and scripts in the world, and has significant support in many applications, operating systems, and programming languages. PDFlib supports the Unicode standard to a large extent. The following features in PDFlib are Unicode-enabled:

- Unicode can be supplied directly in page descriptions.
- Unicode can be supplied for various hypertext elements.
- Unicode strings for text on a page or hypertext elements can be supplied in UTF-8 or UTF-16 formats with any byte ordering.
- PDFlib will include additional information (a *ToUnicode CMap)* in the PDF output which helps Acrobat in assigning proper Unicode values for exporting text (e.g., via the clipboard) and searching for Unicode text.

4.2 Supported Font Formats

4.2.1 PostScript Fonts

PostScript font file formats. PDFlib supports the following file formats for PostScript Type 1 metrics and outline data on all platforms:

- The platform-independent AFM (Adobe Font Metrics) and the Windows-specific PFM (Printer Font Metrics) format for metrics information. Since PFM files do not describe the full character metrics but only the glyphs used in Windows (code page 1252), they can only be used for the *winansi* or *builtin* encodings, while AFM-based font metrics can be rearranged to any encoding supported by the font.
- The platform-independent PFA (Printer Font ASCII) and the Windows-specific PFB (Printer Font Binary) format for font outline information in the PostScript Type 1 format, (sometimes also called »ATM fonts«).
- On the Mac, resource-based PostScript Type 1 fonts, sometimes called LWFN (LaserWriter Font) fonts, are also supported.
- OpenType fonts with PostScript outlines *(*.otf)*.

If you can get hold of a PostScript font file, but not the corresponding metrics file, you can try to generate the missing metrics using one of several freely available utilities. For example, the T1lib package[1] contains the *type1afm* utility for generating AFM metrics from PFA or PFB font files. However, be warned that such conversions often result in font or encoding problems. For this reason it is recommended to use the font outline and metrics data as supplied by the font vendor.

PostScript font names. It is important to use the exact (case-sensitive) PostScript font name whenever a font is referenced in PDFlib unless you work with alias names (see Section 4.3.1, »How PDFlib Searches for Fonts«, page 80). There are several possibilities to find a PostScript font's exact name:

- Open the font outline file *(*.pfa* or **.pfb)*, and look for the string after the entry */FontName*. Omit the leading / character from this entry, and use the remainder as the font name.
- If you have ATM (Adobe Type Manager) installed or are working with Windows 2000/XP, you can double-click the font *(*.pfb)* or metrics *(*.pfm)* file, and will see a font sample along with the PostScript name of the font.
- Open the AFM metrics file and look for the string after the entry *FontName*.

Note The PostScript font name may differ substantially from the Windows font menu name, e.g. »AvantGarde-Demi« (PostScript name) vs. »AvantGarde, Bold« (Windows font menu name). Also, the font name as given in any Windows .inf file is not relevant for use with PDF.

PostScript glyph names. In order to write a custom encoding file or find fonts which can be used with one of the supplied encodings you will have to find information about the exact definition of the character set to be defined by the encoding, as well as the exact glyph names used in the font files. You must also ensure that a chosen font provides all necessary characters for the encoding. For example, the core fonts supplied with Acrobat 4/5 do not support ISO 8859-2 (Latin 2) nor Windows code page 1250. If you happen to have the FontLab[2] font editor (by the way, a great tool for dealing with all kinds of

1. See http://www.neuroinformatik.ruhr-uni-bochum.de/ini/PEOPLE/rmz/t1lib/t1lib.html

font and encoding issues), you may use it to find out about the encodings supported by a given font (look for »code pages« in the FontLab documentation).[1]

For the convenience of PDFlib users, the PostScript program *print_glyphs.ps* in the distribution fileset can be used to find the names of all characters contained in a PostScript font. In order to use it, enter the name of the font at the end of the PostScript file and send it (along with the font) to a PostScript Level 2 or 3 printer, convert it with Acrobat Distiller, or view it with a Level-2-compatible PostScript viewer. The program will print all glyphs in the font, sorted alphabetically by glyph name.

If a font does not contain a glyph required for a custom encoding, it will be missing from the PDF document.

4.2.2 TrueType and OpenType Fonts

TrueType and OpenType file formats. PDFlib supports the following file formats for TrueType and OpenType font s:

- Windows TrueType fonts *(*.ttf),* including CJK fonts
- TrueType collections *(*.ttc)* with multiple fonts in a single file (mostly used for CJK fonts)
- End-user defined character (EUDC) fonts *(*.tte)* created with Microsoft's *eudcedit.exe* tool.
- Platform-independent OpenType fonts with TrueType *(*.ttf)* or PostScript outlines *(*.otf),* including CJK fonts.
- On Mac OS any TrueType font installed on the system (including *.dfont)* can also be used in PDFlib.

TrueType and OpenType font names. It is important to specify the exact (case-sensitive) TrueType font name whenever a font is referenced in PDFlib unless you work with alias names (see Section 4.3.1, »How PDFlib Searches for Fonts«, page 80). In the generated PDF the name of a TrueType font may differ from the name used in PDFlib (or Windows). This is normal, and results from the fact that PDF uses the PostScript name of a TrueType font, which differs from its genuine TrueType name (e.g., *TimesNewRomanPSMT* vs. *Times New Roman).*

Note Contrary to PostScript fonts, TrueType and OpenType font names may contain blank characters.

Finding TrueType font names on Windows. You can easily find the name of an installed font by double-clicking the TrueType font file, and taking note of the full font name which will be displayed in the first line of the resulting window (without the *TrueType* or *OpenType* term in parentheses, of course). Do not use the entry in the second line after the label *Typeface name!* Also, some fonts may have parts of their name localized according to the respective Windows version in use. For example, the common font name portion *Bold* may appear as the translated word *Fett* on a German system. In order to retrieve the font data from the Windows system (host fonts) you must use the translated form of the font name in PDFlib. However, in order to retrieve the font data directly from file you must use the generic (non-localized) form of the font name.

2. See http://www.fontlab.com

1. Information about the glyph names used in PostScript fonts can be found at http://partners.adobe.com/asn/developer/typeforum/unicodegn.html (although font vendors are not required to follow these glyph naming recommendations).

If you want to examine TrueType fonts in more detail take a look at Microsoft's free »font properties extension«[1] which will display many entries of the font's TrueType tables in human-readable form.

Finding TrueType font names on the Mac. Generally, you can find the name of an installed font in the font menu of applications such as TextEdit on Mac OS X. However, this method does not always result in the proper font name as expected by PDFlib. For this reason we recommend Apple's freely available Font Tools[2]. This suite of command-line utilities contains a program called *ftxinstalledfonts* which is useful for determining the exact name of all installed fonts. In order to determine the font name expected by PDFlib, install Font Tools and issue the following statement on the command-line:

```
ftxinstalledfonts -f
```

4.2.3 User-Defined (Type 3) Fonts

Type 3 fonts in PDF (as opposed to PostScript Type 3 fonts) are not actually a file format. Instead, the glyphs in a Type 3 font must be defined at runtime with standard PDFlib graphics functions. Since all PDFlib features for vector graphics, raster images, and even text output can be used in Type 3 font definitions, there are no restrictions regarding the contents of the characters in a Type 3 font. Combined with the PDF import library PDI you can even import complex drawings as a PDF page, and use those for defining a character in a Type 3 font.

Note PostScript Type 3 fonts are not supported.

Type 3 fonts must completely be defined outside of any page (more precisely, the font definition must take place in *document* scope). The following example demonstrates the definition of a simple Type 3 font:

```
PDF_begin_font(p, "Fuzzyfont", 0, 0.001, 0.0, 0.0, 0.001, 0.0, 0.0, "");

PDF_begin_glyph(p, "circle", 1000, 0, 0, 1000, 1000);
PDF_arc(p, 500, 500, 500, 0, 360);
PDF_fill(p);
PDF_end_glyph(p);

PDF_begin_glyph(p, "ring", 400, 0, 0, 400, 400);
PDF_arc(p, 200, 200, 200, 0, 360);
PDF_stroke(p);
PDF_end_glyph(p);

PDF_end_font(p);
```

The font will be registered in PDFlib, and its name can be supplied to *PDF_load_font()* along with an encoding which contains the names of the glyphs in the Type 3 font. Please note the following when working with Type 3 fonts:

- Similar to patterns and templates, images cannot be opened within a glyph description. However, they can be opened before starting a glyph description, and placed within the glyph description. Alternatively, inline images may be used for small bitmaps to overcome this restriction.

1. See http://www.microsoft.com/typography/property/property.htm
2. See http://developer.apple.com/fonts/OSXTools.html

- Due to restrictions in PDF consumers all characters used with text output operators must actually be defined in the font: if character code *x* is to be displayed with *PDF_show( )* or a similar function, and the encoding contains *glyphname* at position *x,* then *glyphname* must have been defined via *PDF_begin_glyph( )*. This restriction affects only Type 3 fonts; missing glyphs in PostScript Type 1, TrueType, or OpenType fonts will simply be ignored.
- Some PDF consumers (this is not true for Acrobat) require a glyph named *.notdef* if codes will be used for which the corresponding glyph names are not defined in the font. The *.notdef* glyph must be present, but it may simply contain an empty glyph description.
- When normal bitmap data is used to define characters, unused pixels in the bitmap will print as white, regardless of the background. In order to avoid this and have the original background color shine through, use the *mask* parameter for constructing the bitmap image.
- The *interpolate* option for images may be useful for enhancing the screen and print appearance of Type 3 bitmap fonts.

4.3 Font Embedding and Subsetting

4.3.1 How PDFlib Searches for Fonts

Sources of font data. PDFlib can access font data from various sources:

- Disk-based font files which have been statically configured via a UPR configuration file (see Section 3.1.6, »Resource Configuration and File Searching«, page 47) or dynamically via *PDF_set_parameter( )* and the *FontOutline* resource category.
- Fonts which have been installed in the operating system. We refer to such fonts as *host fonts*. Instead of fiddling with font and configuration files simply install the font in the operating system (read: drop it into the appropriate *fonts* directory), and PDFlib will happily use it. Host fonts are available on Mac (only TrueType and OpenType, but not currently PostScript fonts) and Windows systems. They can explicitly be configured with the *HostFont* UPR resource category in order to control the search order. This feature can be used, for example, to prefer host fonts over the built-in core fonts.
- Font data passed by the client directly in memory by means of a PDFlib virtual file (PVF). This is useful for advanced applications which have the font data already loaded into memory and want to avoid unnecessary disk access by PDFlib (see Section 3.1.5, »The PDFlib Virtual File System (PVF)«, page 46 for details on virtual files).

Potential problem with Windows fonts. We'd like to alert users to a potential problem with font installation on Windows systems. If you install fonts via the *File, Install new font...* menu item (as opposed to dragging fonts to the Windows Fonts directory) there's a check box *Copy fonts to Fonts folder.* If this box is unchecked, Windows will only place a shortcut (link) to the original font file in the fonts folder. In this case the original font file must live in a directory which is accessible to the application using PDFlib. In particular, font files outside of the Windows Fonts directory may not be accessible to IIS with default security settings. Solution: either copy font files to the Fonts directory, or place the original font file in a directory where IIS has *read* permission.

Similar problems may arise with Adobe Type Manager (ATM) if the *Add without copying fonts* option is checked while installing fonts.

Font name aliasing. Since it can be difficult to find the exact internal name of a font, PDFlib supports font name aliasing for PostScript, TrueType, and OpenType fonts. With font name aliasing you can specify an arbitrary name as an alias for some font. The alias can be specified as a resource of type *HostFont, FontOutline, FontAFM,* and *FontPFM,* both in a UPR file or at runtime. The following sample defines an alias for a disk-based font:

```
PDF_set_parameter(p, "FontOutline", "x=DFHSMincho-W3.ttf");
font = PDF_load_font(p, "x", 0, "winansi", "");
```

Searching for fonts. The font name supplied to *PDF_load_font( )* can be encoded in ASCII, UTF-8, or UTF-16. However, not all encodings are supported for all font sources. The font is searched according to the following scheme:

- If the name is an alias (configured via a UPR file or a call to *PDF_set_parameter( ))* it can be encoded in ASCII or UTF-8. The name to which the alias refers will be used in the next steps to locate a font file (for disk-based fonts) or host font.

- If the name specifies a host font, it can be encoded in ASCII. On Windows UTF-8 and UTF-16 can also be used.
- If the font was not found as a (possibly localized) host font, and was not encoded in UTF-8 or UTF-16, a corresponding font file will be searched by applying the extension-based search described below.
- For TTC (TrueType Collection) fonts the name can be encoded in ASCII, UTF-8, or UTF-16, and will be matched against all names of all fonts in the TTC file.

Extension-based search for disk-based font files. When PDFlib searches for a font outline or metrics file on disk (as opposed to fetching host fonts directly from the operating system) it applies the following search algorithm if the font name consists of plain ASCII characters:

- When the font has been configured as a *FontAFM, FontPFM,* or *FontOutline* resource via UPR file or at runtime the configured file name will be used.
- If no file could be found, the following suffixes will be added to the font name, and the resulting file names tried one after the other to find the font metrics (and outline in the case of TrueType and OpenType fonts):

  ```
  .ttf .otf .afm .pfm .ttc
  .TTF .OTF .AFM .PFM .TTC
  ```

- If embedding is requested for a PostScript font, the following suffixes will be added to the font name and tried one after the other to find the font outline file:

  ```
  .pfa .pfb
  .PFA .PFB
  ```

- All trial file names above will be searched for »as is«, and then by prepending all directory names configured in the *SearchPath* resource category.

This means that PDFlib will find a font without any manual configuration provided the corresponding font file consists of the font name plus the standard file name suffix according to the font type, and is located in one of the *SearchPath* directories.

4.3.2 Font Embedding

The PDF core fonts. PDF viewers support a core set of 14 fonts which are assumed to be always available. Full metrics information for the core fonts is already built into the PDFlib binary so that no additional font files are required (unless the font is to be embedded). 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 replace one of the core fonts with a font installed on the system (host font) you must configure the font in the *HostFont* resource category. For example, the following line makes sure that instead of using the built-in core font data, the Symbol font will be taken from the host system:

```
PDF_set_parameter(p, "HostFont", "Symbol=Symbol");
```

PDF supports fonts outside the set of 14 core fonts in several ways. PDFlib is capable of embedding font outlines into the generated PDF output. Font embedding is controlled via the embedding option of *PDF_load_font( ),* although in some cases PDFlib will enforce font embedding (see below).

Alternatively, a font descriptor containing only the character metrics and some general information about the font (without the actual glyph outlines) can be embedded. If a font is not embedded in a PDF document, Acrobat will take it from the target system if available, or construct a substitute font according to the font descriptor. Table 4.1 lists different situations with respect to font usage, each of which poses different requirements on the font and metrics files required by PDFlib.

When a font with font-specific encoding (a symbol font) or one containing glyphs outside Adobe's Standard Latin character set is used, but not embedded in the PDF output, the resulting PDF will be unusable unless the font is already natively installed on the target system (since Acrobat can only simulate Latin text fonts). Such PDF files are inherently nonportable, although they may be of use in controlled environments, such as intra-corporate document exchange.

Table 4.1 Different font usage situations and required metrics and outline files

font usage	*font metrics file required?*	*font outline file required?*
one of the 14 core fonts	*no*	*no*[1]
TrueType or OpenType font installed on the Mac, or TrueType, OpenType, or PostScript fonts installed on the Windows system (host fonts)	*no*	*no*
non-core PostScript fonts	*PFM or AFM*	*PFB or PFA (only for font embedding)*
TrueType fonts	*no*	*TTF*
OpenType fonts with TrueType or PS outlines, including CJK TrueType and OpenType fonts	*no*	*TTF or OTF*
standard CJK fonts[2]	*no*	*no*

1. Font outlines may be supplied if embedding is desired
2. See Section 4.7, »Chinese, Japanese, and Korean Text«, page 101 for more information on CJK fonts.

Forced font embedding. PDF requires font embedding for certain combinations of font and encoding. PDFlib will therefore force font embedding (regardless of the *embedding* option) in the following cases:

- Using *glyphid* or *unicode* encoding with a TrueType or OpenType font with TT outlines.
- Using a TrueType font or an OpenType font with TrueType outlines with an encoding different from *winansi* and *macroman*.

Note that font embedding will not be enforced for OpenType fonts with PostScript outlines. The requirement for font embedding is caused by the internal conversion to a CID font, which can be disabled by setting the *autocidfont* parameter to *false*. Doing so will also disable forced embedding. Note that in this case not all Latin characters will be accessible, and characters outside the Adobe Glyph List (AGL) won't work at all.

Legal aspects of font embedding. It's important to note that mere possession of a font file may not justify embedding the font in PDF, even for holders of a legal font license. Many font vendors restrict embedding of their fonts. Some type foundries completely

forbid PDF font embedding, others offer special online or embedding licenses for their fonts, while still others allow font embedding provided subsetting is applied to the font. Please check the legal implications of font embedding before attempting to embed fonts with PDFlib. PDFlib will honor embedding restrictions which may be specified in a TrueType or OpenType font. If the embedding flag in a TrueType font is set to *no embedding*[1], PDFlib will honor the font vendor's request, and reject any attempt at embedding the font.

4.3.3 Font Subsetting

In order to decrease the size of the PDF output, PDFlib can embed only those characters from a font which are actually used in the document. This process is called font subsetting. It creates a new font which contains fewer glyphs than the original font, and omits font information which is not required for PDF viewing. Note, however, that Acrobat's TouchUp tool will refuse to work with text in subset fonts. Font subsetting is particularly important for CJK fonts. PDFlib supports subsetting for the following types of fonts:

- TrueType fonts,
- OpenType fonts with PostScript or TrueType outlines.

When a font for which subsetting has been requested is used in a document, PDFlib will keep track of the characters actually used for text output. There are several controls for the subsetting behavior:

- The default subsetting behavior is controlled by the *autosubsetting* parameter. If it is true, subsetting will be enabled for all fonts where subsetting is possible. The default value is true.
- If the *autosubsetting* parameter is false, but subsetting is desired for a particular font nevertheless, the *subsetting* option must be supplied to *PDF_load_font( )*.
- The *subsetlimit* parameter contains a percentage value. If a document uses more than this percentage of glyphs in a font, subsetting will be disabled for this particular font, and the complete font will be embedded instead. This saves some processing time at the expense of larger output files:

  ```
  PDF_set_value(p, "subsetlimit", 75);  /* set subset limit to 75% */
  ```

  The default value of *subsetlimit* is 100 percent. In other words, the subsetting option requested at *PDF_load_font( )* will be honored unless the client explicitly requests a lower limit than 100 percent.
- The *subsetminsize* parameter can be used to completely disable subsetting for small fonts. If the original font file is smaller than the value of *subsetminsize* in KB, font subsetting will be disabled for this font. The default value is 100 KB.

Embedding and subsetting TrueType fonts. The dependencies for TrueType handling are a bit confusing due to certain requirements in PDF. The following is a summary of the information in previous paragraphs.

If a TrueType font is used with an encoding different from *winansi* and *macroman* it will be converted to a CID font for PDF output by default. For encodings which contain only characters from the Adobe Glyph List (AGL) this can be prevented by setting the *autocidfont* parameter to *false*. If the font is converted to a CID font, it will always be embedded. Subsetting will be applied by default, unless the *autosubsetting* parameter is set

1. More specifically: if the fsType flag in the OS/2 table of the font has a value of 2.

to *false*, or the percentage of used glyphs is higher than the *subsetlimit* parameter, or the font file size is in KB smaller than the value of the *subsetminsize* parameter.

4.4 Encoding Details

4.4.1 8-Bit Encodings

Table 4.2 lists the predefined encodings in PDFlib, and details their use with several important classes of fonts. It is important to realize that certain scripts or languages have requirements which cannot be met by common fonts. For example, Acrobat's core fonts do not contain all characters required for ISO 8859-2, while PostScript 3, OpenType Pro, and TrueType »big fonts« do.

Note *The chartab example contained in the PDFlib distribution can be used to easily print character tables for arbitrary font/encoding combinations.*

Notes on the macroman encoding. This encoding reflects the Mac OS character set, albeit with the old currency symbol at position 219 = 0xDB, and not the Euro glyph as redefined by Apple (this incompatibility is dictated by the PDF specification). Also, this encoding does not include the Apple glyph and the mathematical symbols as defined in the Mac OS character set. The *macroman_euro* encoding is identical to *macroman* except that position 219 = 0xDB holds the Euro glyph instead of the currency symbol.

Host encoding. The special encoding *host* does not have any fixed meaning, but will be mapped to another 8-bit encoding depending on the current platform as follows:

- on Mac OS 9 it will be mapped to *macroman;*
- on IBM eServer iSeries and zSeries with MVS or USS it will be mapped to *ebcdic;*
- on Windows, Linux, Mac OS X and all other systems it will be mapped to *winansi;*

Host encoding is primarily useful for writing platform-independent test programs (like those contained in the PDFlib distribution and other simple applications. Host encoding is not recommended for production use, but should be replaced by whatever encoding is appropriate.

Automatic encoding. PDFlib supports a mechanism which can be used to specify the most natural encoding for certain environments without further ado. Supplying the keyword *auto* as an encoding name specifies a platform- and environment-specific 8-bit encoding as follows:

- On Windows: the current system code page (see below for details)
- On Unix and Mac OS X: *iso8859-1*
- On Mac OS Classic: *macroman*
- On IBM eServer iSeries: the current job's encoding *(IBMCCSIDoooooooooooo)*
- On IBM eServer zSeries: *ebcdic* (=code page 1047).

While automatic encoding is convenient in many circumstances, using this method will make your PDFlib client programs inherently non-portable.

Tapping system code pages. PDFlib can be instructed to fetch code page definitions from the system and transform it appropriately for internal use. This is very convenient since it frees you from implementing the code page definition yourself. Instead of supplying the name of a built-in or user-defined encoding for *PDF_load_font( )*, simply use an encoding name which is known to the system. This feature is only available on selected platforms, and the syntax for the encoding string is platform-specific:

Table 4.2 Predefined encodings and their use with several classes of fonts

code page	supported languages	PS Level 1/2, Acrobat 4/5[1]	PostScript 3[2]	OpenType Pro Fonts	»Big Fonts«, e.g., Tahoma
winansi	identical to cp1252, a superset of ISO 8859-1	yes	yes	yes	yes
macroman	Mac Roman encoding, i.e., the default Macintosh character set	yes	yes	yes	yes
macroman_ euro	similar to macroman, but includes the Euro glyph instead of currency	yes	yes	yes	yes
ebcdic	EBCDIC code page 1047	yes	yes	yes	yes
pdfdoc	PDFDocEncoding	yes	yes	yes	yes
iso8859-1 (Latin-1)	Western European languages (implemented as winansi)	yes	yes	yes	yes
iso8859-2 (Latin-2)	Slavic languages of Central Europe	no	yes	yes	yes
iso8859-3 (Latin-3)	Esperanto and Maltese	no	no	yes	yes
iso8859-4 (Latin-4)	Estonian, the Baltic languages, and Greenlandic	no	no	yes	yes
iso8859-5	Bulgarian, Russian, and Serbian	no	no	yes	yes
iso8859-6	Arabic	no	no	no	yes
iso8859-7	Modern Greek	no	no	1 missing	yes
iso8859-8	Hebrew and Yiddish	no	no	no	yes
iso8859-9 (Latin-5)	Western European and Turkish	5 missing	yes	yes	yes
iso8859-10 (Latin-6)	Nordic languages (variation of Latin-4)	no	no	1 missing	yes
iso8859-13 (Latin-7)	Baltic languages	no	yes	yes	yes
iso8859-14 (Latin-8)	Celtic	no	no	no	no
iso8859-15 (Latin-9)	Adds the Euro and some French and Finnish characters to Latin-1	Euro missing	yes	yes	yes
iso8859-16 (Latin-10)	Hungarian, Polish, Romanian, and Slovenian	no	yes	yes	yes
cp1250	Central European	no	yes	yes	yes
cp1251	Cyrillic	no	no	yes	yes
cp1252	Western European (implemented as winansi)	yes	yes	yes	yes
cp1253	Greek	no	no	1 missing	yes
cp1254	Turkish	5 missing	yes	yes	yes
cp1255	Hebrew	no	no	no	yes
cp1256	Arabic	no	no	no	5 missing
cp1257	Baltic	no	yes	yes	yes
cp1258	Viet Nam	no	no	no	yes

1. Original Adobe Latin character set (Type 1 Fonts since 1982)
2. Extended Adobe Latin character set (CE-Fonts) (Type 1 Fonts since PostScript 3)

- On Windows the encoding name is *cp<number>*, where *<number>* is the number of any code page installed in the system:

  ```
  PDF_load_font(p, "Helvetica", 0, "cp1250", "");
  ```

  Single-byte code pages will be transformed into an internal 8-bit encoding, while multi-byte code pages will always be mapped to *unicode*. This means that all strings for page descriptions must be supplied in Unicode by the client programmer.
- On IBM eServer iSeries any *Coded Character Set Identifier* (CCSID) can be used. The CCSID must be supplied as a string, and PDFlib will apply the prefix *IBMCCSID* to the supplied code page number. PDFlib will also add leading 0 characters if the code page number uses fewer than 5 characters. Supplying 0 (zero) as the code page number will result in the current job's encoding to be used:

  ```
  PDF_load_font(p, "Helvetica", 0, "273", "");
  ```

- On IBM eServer zSeries with USS or MVS any *Coded Character Set Identifier* (CCSID) can be used. The CCSID must be supplied as a string, and PDFlib will pass the supplied code page name to the system literally without applying any change:

  ```
  PDF_load_font(p, "Helvetica", 0, "IBM-273", "");
  ```

User-defined 8-bit encodings. In addition to predefined encodings PDFlib supports user-defined 8-bit encodings. These are the way to go if you want to deal with some character set which is not internally available in PDFlib, such as EBCDIC character sets different from the one supported internally in PDFlib. PDFlib supports encoding tables defined by PostScript glyph names, as well as tables defined by Unicode values.

The following tasks must be done before a user-defined encoding can be used in a PDFlib program (alternatively the encoding can also be constructed at runtime using *PDF_encoding_set_char( )*):

- Generate a description of the encoding in a simple text format.
- Configure the encoding in the PDFlib resource file (see Section 3.1.6, »Resource Configuration and File Searching«, page 47) or via *PDF_set_parameter( )*.
- Provide a font (metrics and possibly outline file) that supports all characters used in the encoding.

The encoding file simply lists glyph names and numbers line by line. The following excerpt shows the start of an encoding definition:

```
% Encoding definition for PDFlib, based on glyph names
% name          code    Unicode (optional)
space           32      0x0020
exclam          33      0x0021
...
```

The next example shows a snippet from a Unicode code page:

```
% Code page definition for PDFlib, based on Unicode values
% Unicode       code
0x0020          32
0x0021          33
...
```

More formally, the contents of an encoding or code page file are governed by the following rules:

- Comments are introduced by a percent '%' character, and terminated by the end of the line.
- The first entry in each line is either a PostScript glyph name or a hexadecimal Unicode value composed of a *ox* prefix and four hex digits (upper or lower case). This is followed by whitespace and a hexadecimal (0x00–0xFF) or decimal (0–255) character code. Optionally, name-based encoding files may contain a third column with the corresponding Unicode value.
- Character codes which are not mentioned in the encoding file are assumed to be undefined. Alternatively, a Unicode value of *0x0000* or the character name *.notdef* can be provided for unused slots.

As a naming convention we refer to name-based tables as encoding files *(*.enc)*, and Unicode-based tables as code page files *(*.cpg)*, although PDFlib treats both kinds in the same way, and doesn't care about file names. In fact, PDFlib will automatically convert between name-based encoding files and Unicode-based code page files whenever it is necessary. This conversion is based on Adobe's standard list of PostScript glyph names (the Adobe Glyph List, or AGL[1]), but non-AGL names can also be used. PDFlib will assign free Unicode values to these non-AGL names, and adjusts the values when reading an OpenType font file which includes a mapping from glyph names to Unicode values.

The AGL is built into PDFlib, and contains more than 1000 glyph names. Encoding files are required for PostScript fonts with non-standard glyph names, while code pages are more convenient when dealing with Unicode-based TrueType or OpenType fonts.

4.4.2 Symbol Fonts and Font-specific Encodings

Since Symbol or logo fonts (also called pi fonts) do not usually contain standard characters they must use a different encoding scheme compared to text fonts.

The builtin encoding for PostScript fonts. The encoding name *builtin* doesn't describe a particular character ordering but rather means »take this font as it is, and don't mess with the character set«. This concept is sometimes called a »font-specific« encoding and is very important when it comes to non-text fonts (such as logo and symbol fonts). It is also widely used (somewhat inappropriately) for non-Latin text fonts (such as Greek and Cyrillic). Such fonts cannot be reencoded using one of the standard encodings since their character names don't match those in these encodings. Therefore *builtin* must be used for all symbolic or non-text PostScript fonts, such as *Symbol* and *ZapfDingbats*. Non-text fonts can be recognized by the following entry in their AFM file:

```
EncodingScheme FontSpecific
```

Text fonts can be reencoded (adjusted to a certain code page or character set), while symbolic fonts can't, and must use *builtin* encoding instead.

The *builtin* encoding can not be used for user-defined (Type 3) fonts since these do not include any default encoding.

Note Unfortunately, many typographers and font vendors didn't fully grasp the concept of font specific encodings (this may be due to less-than-perfect production tools). For this reason, there are many Latin text fonts labeled as FontSpecific encoding, and many symbol fonts incorrectly labeled as text fonts.

1. The AGL can be found at http://partners.adobe.com/asn/developer/type/glyphlist.txt

Builtin encoding for TrueType fonts. TrueType fonts with non-text characters, such as the Wingdings font, must be used with *builtin* encoding. If a font requires *builtin* encoding but the client requested a different encoding PDFlib will enforce *builtin* encoding nevertheless.

Builtin encoding for OpenType fonts with PostScript outlines (*.otf). OTF fonts with non-text characters must be used with *builtin* encoding. Some OTF fonts contain an internal default encoding. PDFlib will detect this case, and dynamically construct an encoding which is suited for this particular font. The encoding name *builtin* will be modified to *builtin_<fontname>*. Although this new encoding name can be used in future calls to *PDF_load_font( )* it is only reasonable for use with the same font.

4.4.3 Glyph ID Addressing for TrueType and OpenType Fonts

In addition to 8-bit encodings, Unicode, and CMaps PDFlib supports a method of addressing individual characters within a font called glyph id addressing. In order to use this technique all of the following requirements must be met:

- The font is available in the TrueType or OpenType format.
- The font must be embedded in the PDF document (with or without subsetting).
- The developer is familiar with the internal numbering of glyphs within the font.

Glyph ids *(GIDs)* are used internally in TrueType and OpenType fonts, and uniquely address individual glyphs within a font. GID addressing frees the developer from any restriction in a given encoding scheme, and provides access to all glyphs which the font designer put into the font file. However, there is generally no relationship at all between GIDs and more common addresssing schemes, such as Windows encoding or Unicode. The burden of converting application-specific codes to GIDs is placed on the PDFlib user.

GID addressing is invoked by supplying the keyword *glyphid* as the *encoding* parameter of *PDF_load_font( )*. GIDs are numbered consecutively from 0 to the last glyph id value, which can be queried with the *fontmaxcode* parameter.

4.4.4 The Euro Glyph

The symbol denoting the European currency Euro raises a number of issues when it comes to properly displaying and printing it. In this section we'd like to give some hints so that you can successfully deal with the Euro character. First of all you'll have to choose an encoding which includes the Euro character and check on which position the Euro is located. Some examples:

- With *unicode* encoding use the character U+20AC.
- In *winansi* encoding the location is 0x80 (hexadecimal) or 128 (decimal).
- The common *iso8859-1* encoding does not contain the Euro character. However, the *iso8859-15* encoding is an extension of *iso8859-1* which adds the Euro character at 0xA4 (hexadecimal) or 164 (decimal).
- The original *macroman* encoding, which is still the same in PDF, does not contain the Euro character. However, Apple modified this encoding and replaced the old currency glyph which the Euro glyph at 0xDB (hexadecimal) or 219 (decimal). In order to use this modified Mac encoding use *macroman_euro* instead of *macroman*.

Next, you must choose a font which contains the Euro glyph. Many modern fonts include the Euro glyph, but not all do. Again, some examples:

- The built-in fonts in PostScript Level 1 and Level 2 devices do not contain the Euro character, while those in PostScript 3 devices usually do.
- If a font does not contain the Euro character you can use the Euro from the Symbol core font instead, which is located at position 0xA0 (hexadecimal) or 160 (decimal). It is available in the version of the Symbol font shipped with Acrobat 4.0 and above, and the one built into PostScript 3 devices.

4.5 Unicode Support

PDFlib supports the Unicode standard[1], almost identical to ISO 10646, for a variety of features related to page content and hypertext elements.

4.5.1 Unicode for Page Descriptions

Unicode strings can be supplied directly in page descriptions for use with the following kinds of fonts:

- PostScript fonts with *unicode* encoding. Up to 255 distinct Unicode values can be used. If more are requested they will be replaced with the *space* character. Since PFM metrics files support only *winansi* or *builtin* encoding, *unicode* encoding will always be mapped to *winansi* if a font with a PFM metrics file is used.
- TrueType and OpenType fonts with *unicode* encoding. For TrueType and OpenType fonts this will force font embedding.
- Standard CJK fonts with a Unicode-based CMap. Unicode-compatible CMaps are easily identified by the *Uni* prefix in their name (see Table 4.6).
- Custom CJK fonts with *unicode* encoding.

In addition to *unicode* encoding PDFlib supports several other methods for selecting Unicode characters.

Unicode code pages for PostScript and TrueType fonts. PDFlib supports Unicode addressing for characters within the Adobe Glyph List (AGL). This kind of Unicode support is available for Unicode-based TrueType fonts and PostScript fonts with glyph names in the AGL.

This feature can be activating by using any of PDFlib's internal code pages, or supplying a suitable custom encoding or code page file (see Section 4.4.1, »8-Bit Encodings«, page 85).

8-Bit strings for addressing Unicode segments. PDFlib supports an abbreviated format which can be used to address up to 256 consecutive Unicode characters starting at an arbitrary offset between U+0000 and U+FFFF. This can be used to easily access a small range of Unicode characters while still working with 8-bit characters.

This feature can be activated by using the string *U+XXXX* as the *encoding* parameter for *PDF_load_font( )*, where *XXXX* denotes a hexadecimal offset. The 8-bit character value will be added to the supplied offset. For example, using the encoding

```
U+0400
```

will select the Cyrillic Unicode section, and 8-bit strings supplied to the text functions will select the Unicode characters U+0400, U+0401, etc.

Proper Unicode values for cut-and-paste and find operations. PDFlib will include additional information (a *ToUnicode CMap)* in the PDF output which helps Acrobat in assigning proper Unicode values for exporting text (e.g., via the clipboard) and searching for text. By default ToUnicode CMaps will be generated for all supported font types, but they can only be included if Unicode information is available for a given font/encoding

1. See http://www.unicode.org

combination. While this is case for most font/encoding combinations, user-defined Type 3 fonts, for example, may be missing Unicode information. In this case PDFlib will not be able to generate a ToUnicode CMap, and text export or searching will not work in Acrobat.

Generation of a ToUnicode CMap can be globally disabled with the *unicodemap* parameter, or on a per-font basis with the *PDF_load_font( )* option of the same name. The default of this parameter/option is true. Setting it to false will decrease the output file size while potentially disabling proper cut-and-paste support in Acrobat.

4.5.2 Unicode Text Formats

The Unicode standard supports several transformation formats for storing the actual byte values which comprise a Unicode string. These vary in the number of bytes per character and the ordering of bytes within a character. Unicode strings in PDFlib can be supplied in UTF-8 or UTF-16 formats with any byte ordering. This can be controlled with the *textformat* parameter for all text on page descriptions, and the *hypertextformat* parameter for all hypertext elements. The following values are supported for both of these parameters:

- *bytes:* one byte in the string corresponds to one character. This is mainly useful for 8-bit encodings.
- *utf8:* strings are expected in UTF-8 format.
- *utf16:* strings are expected in UTF-16 format. A Unicode Byte Order Mark (BOM) at the start of the string will be evaluated and then removed. If no BOM is present the string is expected in the machine's native byte ordering (on Intel x86 architectures, for example, the native byte order is little-endian, while on Sparc and PowerPC systems it is big-endian).
- *utf16be:* Strings are expected in UTF-16 format in big-endian byte ordering. There is no special treatment for Byte Order Marks.
- *utf16le:* Strings are expected in UTF-16 format in little-endian byte ordering. There is no special treatment for Byte Order Marks.
- *auto:* equivalent to *bytes* for 8-bit encodings, and *utf16* for wide-character addressing *(unicode, glyphid,* or a UCS2 CMap). This setting will provide proper text interpretation in most environments which do not use Unicode natively.

The default setting for the *textformat* parameter is *utf16* for Unicode-capable language bindings, and *auto* otherwise.

Although the *textformat* setting is in effect for all encodings, it will be most useful for *unicode* encoding. Table 4.3 details the interpretation of text strings for various combinations of font encodings and *textformat* settings.

Table 4.3 Relationship of font encodings and text format

font encoding	*textformat = bytes*	*textformat = utf8, utf16, utf16be, or utf16le*
8-bit, or builtin encoding for TTF/OTF	*8-bit codes*	*convert Unicode values to 8-bit codes according to the chosen encoding*[1]
builtin encoding for PostScript	*8-bit codes*	*only in Unicode-capable language bindings. PDFlib will throw an exception otherwise*
U+XXXX	*8-bit codes will be added to the offset XXXX to address Unicode values*	*convert Unicode values to 8-bit codes according to the chosen Unicode offset*
glyphid	*8-bit codes address glyph ids from 0 to 255*	*Unicode values will be interpreted as glyph ids*[2]

Table 4.3 Relationship of font encodings and text format

font encoding	textformat = bytes	textformat = utf8, utf16, utf16be, or utf16le
unicode and UCS2-based CMaps	8-bit codes address Unicode values from U+0000 to U+00FF	any Unicode value, encoded according to the chosen text format[1]
any other CMap (not UCS2-based)	any single- or multibyte codes according to the chosen CMap	only in Unicode-capable language bindings. PDFlib will throw an exception otherwise

1. If the Unicode character is not available in the font PDFlib will issue a warning and replace it with the space character. (this can be controlled via the glyphwarning parameter).
2. If the glyph id is not available in the font PDFlib will issue a warning and replace it with glyph id 0.

4.5.3 Unicode for Hypertext Elements

Unicode can be supplied for various hypertext elements, such as bookmarks, contents and title of note annotations (see Figure 4.1), standard and user-defined document information field contents, description and author of file attachments. For details on Unicode-enable hypertext items please review the respective function descriptions in Section 7.9, »Hypertext Functions«, page 204.

Note The usability of Unicode in hypertext elements heavily depends on the Unicode support available on the target system. Unfortunately, most systems today are far from being fully Unicode-enabled in their default configurations. Although Windows NT/2000/XP and Mac OS support Unicode internally, availability of appropriate Unicode fonts is still an issue.

Hypertext encoding. PDF supports only two encoding schemes for hypertext elements:

- Unicode in big-ending UTF-16 format.
- PDFDocEncoding, (see Figure 4.2), which is a superset of ISO 8859-1 (Latin 1). Although PDFDocEncoding and the Windows code page 1252 are quite similar, they differ substantially in the character range 128-160 (0x80–0xA0).

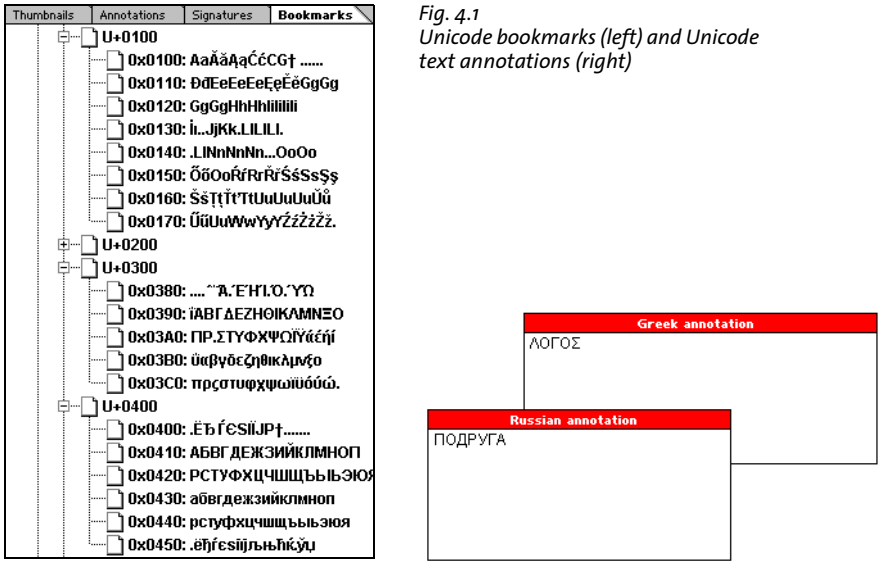


Fig. 4.1
Unicode bookmarks (left) and Unicode text annotations (right)

While PDF allows only Unicode and PDFDocEncoding, PDFlib supports all 8-Bit and Unicode-based encodings which are allowed for *PDF_load_font( )*, and will automatically apply any required conversions.

The *hypertextencoding* parameter works analogous to the *encoding* parameter of *PDF_load_font( )*, and controls the 8-bit encoding of hypertext strings. It can attain any name of an 8-bit encoding known to PDFlib, including *auto* (see Section 4.4, »Encoding Details«, page 85). Note that *glyphid, builtin,* and CMap names are not allowed for this parameter. The default is *auto.*

Hypertext format. Similar to the *textformat* parameter, the format of hypertext strings can be controlled with the *hypertextformat* parameter. However, interpretation of the allowed values is somewhat different for the *hypertextformat* parameter. While *utf8, utf16, utf16be,* and *utf16le* have the same meaning as for the *textformat* parameter, the behavior of *bytes* and *auto* is slightly different:

- *auto:* UTF-16 strings with big-endian BOM will be detected (in C such strings must be terminated with a double-null), and Unicode output will be generated. If the string does not start with a big-endian BOM it will be interpreted as an 8-bit encoded string according to the *hypertextencoding* parameter (see above). If it contains at least one character which is not contained in PDFDocEncoding, the complete string will be converted to a big-endian UTF-16 string, and written to the PDF output as Unicode. Otherwise it will be written to the PDF output as 8-bit encoded PDFDocEncoding text.

Fig. 4.2
The PDFDocEncoding character set with hexadecimal and octal codes.

- *bytes:* one byte in the string corresponds to one character, and the string will be output without any interpretation. This is mainly useful for 8-bit encodings. In addition, UTF-16 strings with big-endian BOM will automatically be detected. In C, such strings must be terminated with a double-null unless the length in bytes is explicitly supplied in the respective function call.

The default setting for the *hypertextformat* parameter is *auto*.

4.5.4 Unicode Support in PDFlib Language Bindings

Unicode in the C language binding. Clients of the C language binding must take care not to use the standard text *(PDF_show( )*, *PDF_show_xy( )*, and *PDF_continue_text( ))* or hypertext functions *(PDF_add_bookmark( )*, etc.) when the text may contain embedded null characters. In such cases the alternate functions *PDF_show2( )* etc. must be used, and the length of the string must be supplied separately. This is not a concern for all other language bindings since the PDFlib language wrappers internally call *PDF_show2( )* etc. in the first place.

Unicode in the C++ language binding. C++ users must be aware of a pitfall related to the compiler automatically converting literal strings to the C++ string type which is expected by the PDFlib API functions: this conversion supports embedded null characters only if an explicit length parameter is supplied. For example, the following will not work since the string will be truncated at the first null character:

```
p.show("\x00\x41\x96\x7B\x8C\xEA");                    // Wrong!
```

To fix this problem apply the string constructor with an explicit length parameter:

```
p.show(string("\x00\x41\x96\x7B\x8C\xEA", 6));         // Correct
```

Unicode-aware language bindings. The following PDFlib language bindings are Unicode-capable:

- COM
- .NET
- Java
- Tcl (requires Tcl 8.2 or above)

These language wrappers will correctly deal with Unicode strings provided by the client, and automatically set certain PDFlib parameters. This has the following consequences:

- Since the language wrapper automatically sets the *textformat* and *hypertextformat* parameters to *utf16*, these are no longer accessible by the client, and must not be used.
- Using *unicode* encoding for page descriptions is the easiest way to deal with encodings.
- Non-Unicode CMaps for standard CJK fonts on page descriptions must be avoided since the wrapper will always supply Unicode to the PDFlib core; only UCS2 CMaps can be used.

The overall effect is basically that clients can provide plain Unicode strings to PDFlib functions without any additional configuration or parameter settings required.

4.6 Text Metrics, Text Variations, and Box Formatting

4.6.1 Font and Character Metrics

Text position. PDFlib maintains the text position independently from the current point for drawing graphics. While the former can be queried via the *textx/texty* parameters, the latter can be queried via *currentx/currenty*.

Character metrics. PDFlib uses the character and font metrics system used by PostScript and PDF which shall be briefly discussed here.

The font size which must be specified by PDFlib users is the minimum distance between adjacent text lines which is required to avoid overlapping character parts. The font size is generally larger than individual characters in a font, since it spans ascender and descender, plus possibly additional space between lines.

The *leading* (line spacing) specifies the vertical distance between the baselines of adjacent lines of text. By default it is set to the value of the font size. The *capheight* is the height of capital letters such as *T* or *H* in most Latin fonts. The *ascender* is the height of lowercase letters such as *f* or *d* in most Latin fonts. The *descender* is the distance from the baseline to the bottom of lowercase letters such as *j* or *p* in most Latin fonts. The descender is usually negative. The values of capheight, ascender, and descender are measured as a fraction of the font size, and must be multiplied with the required font size before being used.

The values of capheight, ascender, and descender for a specific font can be queried from PDFlib as follows:

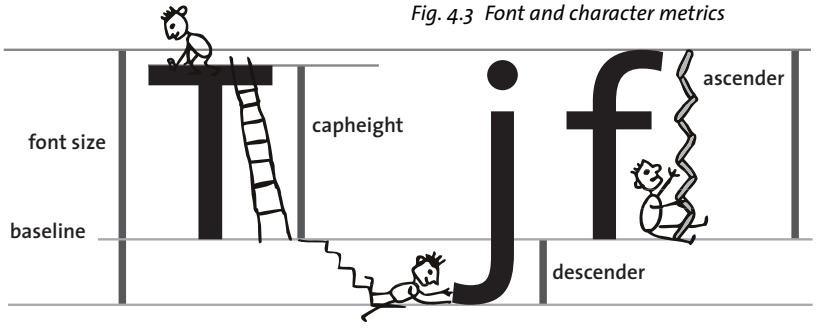
```
float capheight, ascender, descender, fontsize;
...
font = PDF_load_font(p, "Times-Roman", 0, "winansi", "");
PDF_setfont(p, font, fontsize);

capheight = PDF_get_value(p, "capheight", font) * fontsize;
ascender = PDF_get_value(p, "ascender", font) * fontsize;
descender = PDF_get_value(p, "descender", font) * fontsize;
```

Note The position and size of superscript and subscript cannot be queried from PDFlib.

CPI calculations. While most fonts have varying character widths, so-called monospaced fonts use the same widths for all characters. In order to relate PDF font metrics to the characters per inch (CPI) measurements often used in high-speed print environ-

Fig. 4.3 Font and character metrics



ments, some calculation examples for the mono-spaced Courier font may be helpful. In Courier, all characters have a width of 600 units with respect to the full character cell of 1000 units per point (this value can be retrieved from the corresponding AFM metrics file). For example, with 12 point text all characters will have an absolute width of

```
12 points * 600/1000 = 7.2 points
```

with an optimal line spacing of 12 points. Since there are 72 points to an inch, exactly 10 characters of Courier 12 point will fit in an inch. In other words, 12 point Courier is a 10 cpi font. For 10 point text, the character width is 6 points, resulting in a 72/6 = 12 cpi font. Similarly, 8 point Courier results in 15 cpi.

4.6.2 Kerning

Some character combinations can lead to unpleasant appearance. For example, two Vs next to each other can look like a W, and the distance between T and e must be reduced in order to avoid ugly white space. This compensation is referred to as kerning. Many fonts contain comprehensive kerning tables which contain spacing adjustment values for certain critical letter pairs.

PDFlib supports kerning for PostScript, TrueType and OpenType fonts, but not for PostScript host fonts on the Mac (fonts fetched from the operating system). There are two PDFlib controls for the kerning behavior:

- By default, kerning information in a font is not read when loading a font. If kerning is desired the *kerning* option must be set in the respective call to *PDF_load_font( )*. This instructs PDFlib to read the font's kerning data (if available).
- When a font for which kerning data has been read is used with any text output function, the positional corrections provided by the kerning data will be applied. However, kerning can also be disabled by setting the *kerning* parameter to *false*:

```
PDF_set_parameter(p, "kerning", "false");  /* disable kerning */
```

Fig. 4.4 Kerning

Temporarily disabling kerning may be useful, for example, for tabular figures when the kerning data contains pairs of figures, since kerned figures wouldn't line up in a table.

Kerning is applied in addition to any character spacing, word spacing, and horizontal scaling which may be activated. Note, however, that the combination of horizontal spacing and kerning only works correctly in Acrobat 4.05 and above, but not any older versions.

PDFlib does not have any limit for the number of kerning pairs in a font.

4.6.3 Text Variations

Artificial font styles. Bold and italic variations of a font should normally be created by choosing an appropriate font. In addition, PDFlib also supports artificial font styles: based on a regular font Acrobat will simulate bold, italic, or bolditalic styles by emboldening or slanting the base font. The aesthetic quality of artificial font styles does not match those of real bold or italic fonts which have been fine-tuned by the font designer. However, in situations where a particular font style is not available directly, artificial styles can be used as a workaround. In particular, artificial font styles are useful for the standard CJK fonts which support only normal fonts, but not any bold or italic variants.

Note Using the fontstyle feature for fonts other than the standard CJK fonts is not recommended.

Due to restrictions in Adobe Acrobat, artificial font styles work only if all of the following conditions are met:

- The base font is a TrueType or OpenType font, including standard and custom CJK fonts. The base font must not be one of the PDF core fonts (see Section 4.3.2, »Font Embedding«, page 81).
- The encoding is *winansi, macroman,* or one of the predefined CJK CMaps listed in Table 4.6 (since otherwise PDFlib will force font embedding).
- The *embedding* option must be set to *false*.
- The base font must be installed on the target system where the PDF will be viewed.

While PDFlib will check the first three conditions, it is the user's responsibility to ensure the last one.

Artificial font styles can be requested by using one of the *normal* (no change of the base font)*, bold, italic,* or *bolditalic* keywords for the *fontstyle* option of *PDF_load_font( ):*

```
PDF_load_font(p, "HeiseiKakuGo-W5", 0, "UniJIS-UCS2-H", "fontstyle bold");
```

Underline, overline, and strikeout text. PDFlib can be instructed to put lines below, above, or in the middle of text. The stroke width of the bar and its distance from the baseline are calculated based on the font's metrics information. In addition, the current values of the horizontal scaling factor and the text matrix are taken into account when calculating the width of the bar. *PDF_set_parameter( )* can be used to switch the underline, overline, and strikeout feature on or off as follows:

```
PDF_set_parameter(p, "underline", "true");      /* enable underlines */
```

The current stroke color is used for drawing the bars. The current linecap and dash parameters are ignored, however. Aesthetics alert: in most fonts underlining will touch descenders, and overlining will touch diacritical marks atop ascenders.

Note The underline, overline, and strikeout features are not supported for standard CJK fonts unless a UCS2 CMap is used.

Text rendering modes. PDFlib supports several rendering modes which affect the appearance of text. This includes outline text and the ability to use text as a clipping path. Text can also be rendered invisibly which may be useful for placing text on scanned images in order to make the text accessible to searching and indexing, while at the same time assuring it will not be visible directly. The rendering modes are described in Table 4.4. They can be set with *PDF_set_value( )* and the *textrendering* parameter.

```
PDF_set_value(p, "textrendering", 1);   /* set stroked text rendering (outline text) */
```

When stroking text, graphics state parameters such as linewidth and color will be applied to the glyph outline. The rendering mode has no effect on text displayed using a Type 3 font.

Table 4.4 Values for the text rendering mode

value	explanation	value	explanation
0	fill text	4	fill text and add it to the clipping path
1	stroke text (outline)	5	stroke text and add it to the clipping path
2	fill and stroke text	6	fill and stroke text and add it to the clipping path
3	invisible text	7	add text to the clipping path

Text color. Text will usually be display in the current fill color, which can be set using *PDF_setcolor( )*. However, if a rendering mode other than 0 has been selected, both stroke and fill color may affect the text depending on the selected rendering mode.

4.6.4 Box Formatting

While PDFlib offers the *PDF_stringwidth( )* function for performing text width calculations, many clients need easy access to text box formatting and justifying, e.g. to fit a certain amount of text into a given column. Although PDFlib offers such features, you shouldn't think of PDFlib as a full-featured text and graphics layout engine. The *PDF_show_boxed( )* function is an easy-to-use method for text box formatting with a number of formatting options. Text may be laid out in a rectangular box either left-aligned, right-aligned, centered, or fully justified. The first line of text starts at a baseline with a vertical position which equals the top edge of the supplied box minus the leading. The bottom edge of the box serves as the last baseline used. For this reason, descenders of the last text line may appear outside the specified box (see Figure 4.5).

This function justifies by adjusting the inter-word spacing (the last line will be left-aligned only). Obviously, this requires that the text contains spaces (PDFlib will not insert spaces if the text doesn't contain any). Advanced text processing features such as hyphenation are not available – PDFlib simply breaks text lines at existing whitespace characters. Text is never clipped at the boundaries of the box.

Supplying a *feature* parameter of *blind* can be useful to determine whether a string fits in a given box, without actually producing any output.

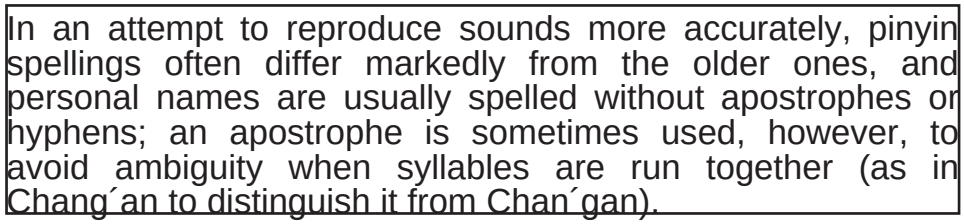


Fig. 4.5 Text box formatting: the bottom edge will serve as the last baseline, not as a clipping border.

ASCII newline characters (in C: *oxoA)* in the supplied text are recognized, and force a new paragraph. CR/NL combinations are treated like a single newline character. Other formatting characters (especially tab characters) are not supported.

The following is a small example of using *PDF_show_boxed( )*. It uses *PDF_rect( )* to draw an additional border around the box which may be helpful in debugging:

```
text = "In an attempt to reproduce sounds more accurately, pinyin spellings often ... ";
fontsize = 13;

font = PDF_load_font(p, "Helvetica", 0, "auto", "");
PDF_setfont(p, font, fontsize);

x = 50;
y = 650;
w = 357;
h = 6 * fontsize;

c = PDF_show_boxed(p, text, x, y, w, h, "justify", "");
if (c > 0 ) {
        /* Not all characters could be placed in the box; act appropriately here */
        ...
}
PDF_rect(p, x, y, w, h);
PDF_stroke(p);
```

The following requirements and restrictions of *PDF_show_boxed( )* shall be noted:

- The function supports only the setting *bytes* for the *textformat* parameter, or the setting *auto* in combination with an 8-bit encoding.
- Contiguous blanks in the text should be avoided.
- Due to restrictions in PDF's word spacing support, the *space* character must be available at code position 0x20 in the encoding. Although this is the case for most common encodings, it implies that justification will not work with EBCDIC encoding.
- The simplistic formatting algorithm may fail for unsuitable combinations of long words and narrow columns since there is no hyphenation support.
- Since the bottom part of the box is used as a baseline, descenders in the last line may extend beyond the box area.
- It's currently not possible to feed the text in multiple portions into the box formatting routine. However, you can retrieve the text position after calling *PDF_show_boxed( )* with the *textx* and *texty* parameters.
- The font within the text box can't be changed.
- Text box formatting is only supported for 8-bit encodings.

4.7 Chinese, Japanese, and Korean Text

4.7.1 CJK support in Acrobat and PDF

Acrobat/PDF supports a set of standard CJK fonts without font embedding, as well as custom embedded CJK fonts. While embedded CJK fonts will work in all versions of Acrobat without further ado, using any of the standard CJK fonts in Acrobat requires the user to do one of the following[1]:

- Use a localized CJK version of Acrobat.
- If you use any non-CJK version of the full Acrobat product, select the Acrobat installer's option »Asian Language Support« (Windows) or »Language Kit« (Mac). The required support files (fonts and CMaps) will be installed from the Acrobat product CD-ROM.
- If you use Acrobat Reader, install one of the Asian Font Packs which are available on the Acrobat product CD-ROM, or on the Web.[2]

Printing PDF documents with CJK text. Printing CJK documents gives rise to a number of issues which are outside the scope of this manual. However, we will supply some useful hints for the convenience of PDFlib users. If you have trouble printing CJK documents (especially those using the standard fonts) with Acrobat, consider the following:

- Acrobat's CJK support is based on CID fonts. Printing CID fonts does not work on all PostScript printers. Native CID font support has only been integrated in PostScript version 2015, i.e., PostScript Level 1 and early Level 2 printers do not natively support CID fonts. However, for early Level 2 devices the printer driver is supposed to take care of this by downloading an appropriate set of compatibility routines to pre-2015 Level 2 printers.
- Due to the large number of characters CID fonts consume very much printer memory unless font subsetting has been applied. Disk files for full CJK fonts typically are 5 to 10 MB in size. Not all printers have enough memory for printing such fonts. For example, in our testing we found that we had to upgrade a Level 3 laser printer from 16 MB to 48 MB RAM in order to reliably print PDF documents with CID fonts.
- Non-Japanese PostScript printers do not have any Japanese fonts installed. For this reason, you must check *Download Asian Fonts* in Acrobat's print dialog.
- If you can't successfully print using downloaded fonts, check *Print as image* in Acrobat's print dialog. This instructs Acrobat to send a bitmapped version of the page to the printer (300 dpi, though).

4.7.2 Standard CJK Fonts and CMaps

Historically, a wide variety of CJK encoding schemes has been developed by diverse standards bodies, companies, and other organizations. Fortunately, all prevalent encodings are supported by Acrobat and PDF by default. Since the concept of an encoding is much more complicated for CJK text than for Latin text, simple 8-bit encodings no longer suffice. Instead, PostScript and PDF use the concept of character collections and character maps *(CMaps)* for organizing the characters in a font.

1. This is a good opportunity to praise Ken Lunde's seminal tome »CJKV information processing – Chinese, Japanese, Korean & Vietnamese Computing« (O'Reilly 1999, ISBN 1-56592-224-7), as well as his work at Adobe since he's one of the driving forces behind CJK support in PostScript and PDF.
2. See http://www.adobe.com/products/acrobat/acrrasianfontpack.html

Acrobat supports a number of standard fonts for CJK text. These fonts are supplied with the Acrobat installation (or the Asian FontPack), and therefore don't have to be embedded in the PDF file. These fonts contain all characters required for common encodings, and support both horizontal and vertical writing modes. The standard fonts and CMaps are documented in Table 4.5. The Acrobat 4 fonts can also be used with Acrobat 5, but the corresponding Acrobat 5 fonts will be used for display and printing if a required font is not installed on the system.

Note Acrobat's standard CJK fonts do not support bold and italic variations. However, these can be simulated with the artificial font style feature (see Section 4.6.3, »Text Variations«, page 98).

As can be seen from the table, the default CMaps support most CJK encodings used on Mac, Windows, and Unix systems, as well as several other vendor-specific encodings. In particular, the major Japanese encoding schemes Shift-JIS, EUC, ISO 2022, and Unicode (UCS-2) are supported. Tables with all supported characters are available from Adobe[1]; CMap descriptions can be found in Table 4.6.

Note Unicode-capable language bindings must only use UCS2-compatible CMaps. Other CMaps are not supported.

Table 4.5 Acrobat's standard fonts and CMaps (encodings) for Japanese, Chinese, and Korean text

locale	font name	sample	supported CMaps (encodings)
Simplified Chinese	STSong-Light[1] STSongStd-Light-Acro[2]	国际 国际	GB-EUC-H, GB-EUC-V, GBpc-EUC-H, GBpc-EUC-V, GBK-EUC-H, GBK-EUC-V, GBKp-EUC-H, GBKp-EUC-V, GBK2K-H, GBK2K-V, UniGB-UCS2-H, UniGB-UCS2-V
Traditional Chinese	MHei-Medium[1] MSung-Light[1] MSungStd-Light-Acro[2]	中文 中文 中文	B5pc-H, B5pc-V, HKscs-B5-H, HKscs-B5-V, ETen-B5-H, ETen-B5-V, ETenms-B5-H, ETenms-B5-V, CNS-EUC-H, CNS-EUC-V, UniCNS-UCS2-H, UniCNS-UCS2-V
Japanese	HeiseiKakuGo-W5[1] HeiseiMin-W3[1] KozMinPro-Regular-Acro[2]	日本語 日本語 日本語	83pv-RKSJ-H, 90ms-RKSJ-H, 90ms-RKSJ-V, 90msp-RKSJ-H, 90msp-RKSJ-V, 90pv-RKSJ-H, Add-RKSJ-H, Add-RKSJ-V, EUC-H, EUC-V, Ext-RKSJ-H, Ext-RKSJ-V, H, V, UniJIS-UCS2-H, UniJIS-UCS2-V, UniJIS-UCS2-HW-H, UniJIS-UCS2-HW-V
Korean	HYGoThic-Medium[1] HYSMyeongJo-Medium[1] HYSMyeongJoStd-Medium-Acro[2]	한국 한국 한국	KSC-EUC-H, KSC-EUC-V, KSCms-UHC-H, KSCms-UHC-V, KSCms-UHC-HW-H, KSCms-UHC-HW-V, KSCpc-EUC-H, UniKS-UCS2-H, UniKS-UCS2-V

1. Available in Acrobat 4; Acrobat 5 will substitute these with different fonts.
2. Available in Acrobat 5 only.

Horizontal and vertical writing mode. PDFlib supports both horizontal and vertical writing modes for standard CJK fonts and CMaps. The mode is selected along with the encoding by choosing the appropriate CMap name. CMaps with names ending in *-H* select horizontal writing mode, while the *-V* suffix selects vertical writing mode.

1. See http://partners.adobe.com/asn/developer/typeforum/cidfonts.html for a wealth of resources related to CID fonts, including tables with all supported glyphs (search for »character collection«).

Table 4.6 Predefined CMaps for Japanese, Chinese, and Korean text (from the PDF Reference)

locale	supported CMaps	description
Simplified Chinese	UniGB-UCS2-H UniGB-UCS2-V	Unicode (UCS-2) encoding for the Adobe-GB1 character collection
	GB-EUC-H GB-EUC-V	Microsoft Code Page 936 (charset 134), GB 2312-80 character set, EUC-CN encoding
	GBpc-EUC-H GBpc-EUC-V	Macintosh, GB 2312-80 character set, EUC-CN encoding, Script Manager code 2
	GBK-EUC-H, -V	Microsoft Code Page 936 (charset 134), GBK character set, GBK encoding
	GBKp-EUC-H[1] GBKp-EUC-V[1]	Same as GBK-EUC-H, but replaces half-width Latin characters with proportional forms and maps code 0x24 to dollar ($) instead of yuan (¥).
	GBK2K-H[1], -V[1]	GB 18030-2000 character set, mixed 1-, 2-, and 4-byte encoding
Traditional Chinese	UniCNS-UCS2-H UniCNS-UCS2-V	Unicode (UCS-2) encoding for the Adobe-CNS1 character collection
	B5pc-H B5pc-V	Macintosh, Big Five character set, Big Five encoding, Script Manager code 2
	HKscs-B5-H[1] HKscs-B5-V[1]	Hong Kong SCS (Supplementary Character Set), an extension to the Big Five character set and encoding
	ETen-B5-H ETen-B5-V	Microsoft Code Page 950 (charset 136), Big Five character set with ETen extensions
	ETenms-B5-H ETenms-B5-V	Same as ETen-B5-H, but replaces half-width Latin characters with proportional forms
	CNS-EUC-H, -V	CNS 11643-1992 character set, EUC-TW encoding
Japanese	UniJIS-UCS2-H, -V	Unicode (UCS-2) encoding for the Adobe-Japan1 character collection
	UniJIS-UCS2-HW-H UniJIS-UCS2-HW-V	Same as UniJIS-UCS2-H, but replaces proportional Latin characters with half-width forms
	83pv-RKSJ-H	Macintosh, JIS X 0208 character set with KanjiTalk6 extensions, Shift-JIS encoding, Script Manager code 1
	90ms-RKSJ-H 90ms-RKSJ-V	Microsoft Code Page 932 (charset 128), JIS X 0208 character set with NEC and IBM extensions
	90msp-RKSJ-H 90msp-RKSJ-V	Same as 90ms-RKSJ-H, but replaces half-width Latin characters with proportional forms
	90pv-RKSJ-H	Macintosh, JIS X 0208 character set with KanjiTalk7 extensions, Shift-JIS encoding, Script Manager code 1
	Add-RKSJ-H, -V	JIS X 0208 character set with Fujitsu FMR extensions, Shift-JIS encoding
	EUC-H, -V	JIS X 0208 character set, EUC-JP encoding
	Ext-RKSJ-H, -V	JIS C 6226 (JIS78) character set with NEC extensions, Shift-JIS encoding
	H, V	JIS X 0208 character set, ISO-2022-JP encoding
Korean	UniKS-UCS2-H -V	Unicode (UCS-2) encoding for the Adobe-Korea1 character collection
	KSC-EUC-H, -V	KS X 1001:1992 character set, EUC-KR encoding
	KSCms-UHC-H KSCms-UHC-V	Microsoft Code Page 949 (charset 129), KS X 1001:1992 character set plus 8822 additional hangul, Unified Hangul Code (UHC) encoding
	KSCms-UHC-HW-H KSCms-UHC-HW-V	Same as KSCms-UHC-H, but replaces proportional Latin characters with half-width forms
	KSCpc-EUC-H	Macintosh, KS X 1001:1992 character set with Mac OS KH extensions, Script Manager Code 3

1. Only available for PDF 1.4 / Acrobat 5 and above

Note Some PDFlib functions change their semantics according to the writing mode. For example, PDF_continue_text( ) should not be used in vertical writing mode, and the character spacing must be negative in order to spread characters apart in vertical writing mode.

CJK text encoding for standard CMaps. The client is responsible for supplying text encoded such that it matches the requested CMap. PDFlib does not check whether the supplied text conforms to the requested CMap.

For multi-byte encodings, the high-order byte of a character must appear first. Alternatively, the byte ordering and text format can be selected with the *textformat* parameter (see Section 4.5.1, »Unicode for Page Descriptions«, page 91) provided a UCS2-based CMap is used.

Since several of the supported encodings may contain null characters in the text strings, C developers must take care not to use the *PDF_show( )* etc. functions, but instead *PDF_show2( )* etc. which allow for arbitrary binary strings along with a length parameter. For all other language bindings, the text functions support binary strings, and *PDF_show2( )* etc. are not required.

Restrictions for standard CJK fonts and CMaps. The following features are not supported for standard CJK fonts in combination with CMaps other than UCS2:

- calculating the extent of text with *PDF_stringwidth( )* (but see Section 4.7.4, »Forcing monospaced Fonts«, page 106)
- box formatting with *PDF_show_boxed( )* (doesn't work with any standard CMap, even UCS2)
- activating underline/overline/strikeout mode
- retrieving the *textx/texty* position

These restrictions hold for standard CJK fonts. Note that although the widths of CJK text cannot be queried in these cases, the width will nevertheless be generated correctly in the PDF output. Also note the above features are well supported for custom CJK fonts.

Note The UniJIS-UCS2-HW-H/V CMaps are incorrectly treated as monospaced. This will be fixed in a future release.

Note These restrictions will be lifted in a future release. We intend to offer extended support for other major CJK encodings including SJIS and EUC.

Standard CJK font example. Standard CJK fonts can be selected with the *PDF_load_font( )* interface, supplying the CMap name as the *encoding* parameter. However, you must take into account that a given CJK font supports only a certain set of CMaps (see Table 4.5), and that Unicode-aware language bindings support only UCS2-compatible CMaps. The *KozMinPro-Regular-Acro* sample in Table 4.5 can been generated with the following code:

```
font = PDF_load_font(p, "KozMinPro-Regular-Acro", 0, "UniJIS-UCS2-H", "");
PDF_setfont(p, font, 24);
PDF_set_text_pos(p, 50, 500);
/* We use UTF-16 format with little-endian (LE) byte ordering */
PDF_set_parameter(p, "textformat", "utf16le");
PDF_show(p, "\xE5\x65\x2C\x67\x9E\x8A");
```

These statements locate one of the Japanese standard fonts, choosing a Shift-JIS-compatible CMap *(Ext-RKSJ)* and horizontal writing mode *(H)*. The *fontname* parameter must

be the exact name of the font without any encoding or writing mode suffixes. The *encoding* parameter is the name of one of the supported CMaps (the choice depends on the font) and will also indicate the writing mode (see above). PDFlib supports all of Acrobat's default CMaps, and will complain when it detects a mismatch between the requested font and the CMap. For example, PDFlib will reject a request to use a Korean font with a Japanese encoding.

4.7.3 Custom CJK Fonts

In addition to Acrobat's standard CJK fonts PDFlib supports custom CJK fonts (fonts outside the list in Table 4.5) in the TrueType (including TrueType Collections, TTC) and OpenType formats. A custom CJK font will be processed as follows:

- The font will be converted to a CID font and embedded in the PDF output regardless of the embedding setting provided by the client. Since PDFlib respects font embedding restrictions which may be defined in a font, fonts which do not allow embedding can not be used as custom CJK fonts.
- By default, font subsetting will be applied to all embedded custom CJK fonts; this can be controlled with various parameters, see Section 4.3, »Font Embedding and Subsetting«, page 80.
- Proportional Latin characters and half-width characters are fully supported for custom CJK fonts.
- Japanese host font names can be supplied to *PDF_load_font( )* as UTF-8 with initial BOM, or UCS-2.

Note Original Composite Fonts (OCF) and raw PostScript CID fonts are not supported. Windows EUDC fonts are supported.

Supported encodings for custom CJK fonts. Custom CJK fonts can be used with the following encodings:

- *unicode* encoding.
- 8-bit encodings (although these are unlikely to be useful for CJK text)
- *glyphid* addressing (see Section 4.4.3, »Glyph ID Addressing for TrueType and OpenType Fonts«, page 89)

The *textformat* parameter will be evaluated for custom CJK fonts.

Restrictions for custom CJK fonts. The following features are currently not supported for custom CJK fonts:

- Encodings other than those listed above can not be used. In particular, the CMaps listed in Table 4.6 can not be used with custom CJK fonts, but only with the standard CJK fonts.
- Vertical writing mode is not implemented.

Custom CJK font example. The following example uses the ArialUnicodeMS font to display some Chinese text. The font must either be installed on the system or must be configured according to Section 4.3.1, »How PDFlib Searches for Fonts«, page 80):

```
/* This is not required if the font is installed on the system */
PDF_set_parameter(p, "FontOutline", "Arial Unicode MS=ARIALUNI.TTF");
font = PDF_load_font(p, "Arial Unicode MS", 0, "unicode", "");

PDF_setfont(p, font, 24);
```

```
PDF_set_text_pos(p, x, y);

/* We use UTF-16 format with big-endian (BE) byte ordering */
PDF_set_parameter(p, "textformat", "utf16be");
PDF_show2(p, "\x4e\x00\x50\x0b\x4e\xba", 6);
```

4.7.4 Forcing monospaced Fonts

Some applications are not prepared to deal with proportional CJK fonts, and calculate the extent of text based on a constant glyph width and the number of glyphs. PDFlib can be instructed to force monospaced glyphs even for fonts that usually have glyphs with varying widths. Use the *monospace* option of *PDF_load_font()* to specify the desired width for all glyphs. For standard CJK fonts the value 1000 will result in pleasing results:

```
font = PDF_load_font(p, "KozMinPro-Regular-Acro", 0, "UniJIS-UCS2-H", "monospace 1000");
```

The *monospace* option is only recommended for standard CJK fonts.

4.8 Placing and Fitting Text

The function *PDF_fit_textline( )* for placing a single line of text on a page offers a wealth of formatting options. The most important options will be discussed in this section using some common application examples. A complete description of these options can be found in Table 7.10. Most options for *PDF_fit_textline( )* are identical to those of *PDF_fit_image( )* . Therefore we will only use text-related examples here; it is recommended to take a look at the examples in Section 5.3, »Placing Images and Imported PDF Pages«, page 121, for an introduction.

The examples below demonstrate only the relevant call of the function *PDF_fit_textline( )* , assuming that the required font has already been loaded and set in the desired font size.

PDF_fit_textline( ) uses the so-called text box to determine the positioning of the text: the width of the text box is identical to the width of the text, and the box height is identical to the height of capital letters in the font. The text box can be extended to the left and right or top and bottom using the *margin* option. The margin will be scaled along with the text line.

4.8.1 Simple Text Placement

Placing text in the bottom center. We place text at the reference point such that the text box will be positioned with the center of its bottom line at the reference point (see Figure 4.6):

```
PDF_fit_textline(p, text, 297, 0, "position {50 0}");
```

This code fragment places the text box with the bottom center *(position {50 0})* at the reference point *(297, 0)*.

Placing text in the top right corner. Now we place the text at the reference point such that the text box will be placed with the upper right corner at the reference point (see Figure 4.7):

```
PDF_fit_textline(p, text, 595, 842, "position 100");
```

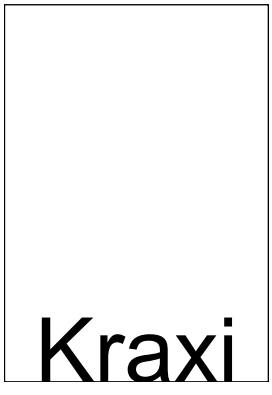


Fig. 4.6
Placing text in the bottom center

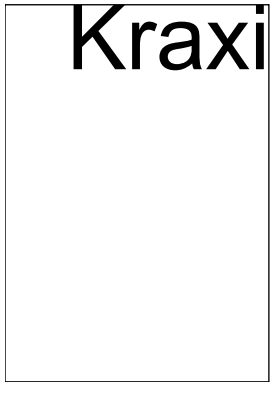


Fig. 4.7
Placing text in the upper right corner

This code fragment places the text box with the upper right corner *(position 100)* at the reference point *(595, 842)*.

Placing text with a margin. To extend the previous example we can add a horizontal margin to the text to achieve a certain distance to the right. This may be useful for placing text in table columns:

```
PDF_fit_textline(p, text, 595, 842, "position 100 margin {20 0}");
```

4.8.2 Placing Text in a Box

Placing centered text in a box. We define a box and place the text centered within the box (see Figure 4.8):

```
PDF_fit_textline(p, text, 10, 200, "boxsize {500 220} position 50");
```

This code fragment places the text centered *(position 50)* in a box with the lower left corner at *(10, 200)*, 500 units wide and 220 units high (*boxsize {500 220}*).

Proportionally fitting text to a box. We extend the previous example and fit the text into the box completely (see Figure 4.9):

```
PDF_fit_textline(p, text, 10, 200, "boxsize {500 220} position 50 fitmethod meet");
```

Note that the font size will be changed when text is fit into the box with *fitmethod meet*. In order to prevent the text from being scaled up use *auto* instead of *meet*.

Completely fitting text to a Box. We can further modify the previous example such that the text will not be fit into the box proportionally, but completely covers the box. However, this combination will only rarely be used since the text may be distorted (see Figure 4.10):

```
PDF_fit_textline(p, text, 10, 200, "boxsize {500 220} position 50 fitmethod entire");
```

Fig. 4.8
Placing centered text in a box

Fig. 4.9
Proportionally fitting text to a box

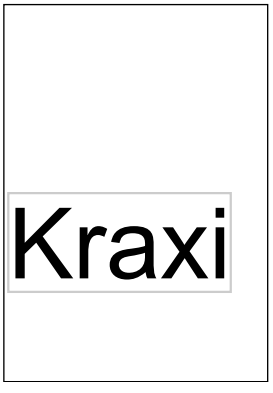
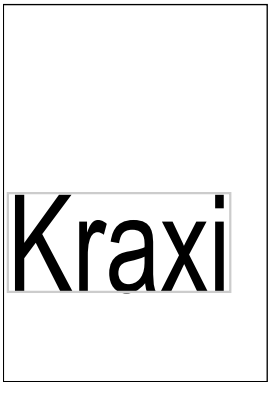


Fig. 4.10
Completely fitting text to a box



4.8.3 Aligning Text

Simple alignment. Our next goal is to rotate text such that its original lower left corner will be placed at a given reference point (see Figure 4.11). This may be useful, for example, for placing a rotated column heading in a table header:

```
PDF_fit_textline(p, text, 5, 5, "orientate west");
```

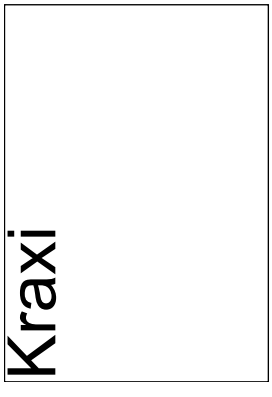
This code fragment orientates the text to the west (90° counterclockwise) and then translates it the lower left corner of the rotated text to the reference point *(5, 5)*.

Aligning text at a vertical line. Positioning text along a vertical line (i.e., a box with zero width) is a somewhat extreme case which may be useful nevertheless (see Figure 4.12):

```
PDF_fit_textline(p, text, 0, 0, "boxsize {0 600} position {0 50} orientate west");
```

This code fragment rotates the text, and places it at the center of the line from *(0, 0)* to *(0, 600)*.

Fig. 4.11
Simple Aligning



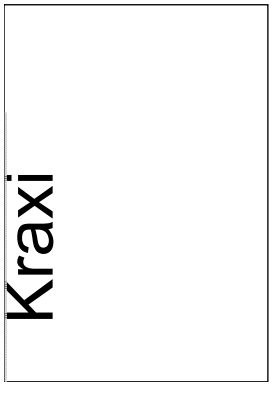


Fig. 4.12
Aligning text at a vertical line

5 Importing and Placing Objects

PDFlib offers a variety of features for importing raster images and pages from existing PDF documents, and placing them on the page. This chapter covers the details of dealing with raster images and importing pages from existing PDF documents. It also presents samples which demonstrate how to place images and PDF pages on an output page.

5.1 Importing Raster Images

5.1.1 Basic Image Handling

Embedding raster images with PDFlib is easy to accomplish. First, the image file has to be opened with a PDFlib function which does a brief analysis of the image parameters. The *PDF_load_image( )* function returns a handle which serves as an image descriptor. This handle can be used in a call to *PDF_fit_image( )*, along with positioning and scaling parameters:

```
if ((image = PDF_load_image(p, "jpeg", "image.jpg", 0, "")) == -1) {
        fprintf(stderr,"Error: Couldn't read image file.\n");
} else {
        PDF_fit_image(p, image, 0.0, 0.0, "");
        PDF_close_image(p, image);
}
```

The last argument to *PDF_fit_image( )* is an option list which supports a variety of options for positioning, scaling, and rotating the image. Details regarding these options are discussed in Section 5.3, »Placing Images and Imported PDF Pages«, page 121.

Re-using image data. PDFlib supports an important PDF optimization technique for using repeated raster images. Consider a layout with a constant logo or background on multiple pages. In this situation it is possible to include the actual image data only once in the PDF, and generate only a reference on each of the pages where the image is used. Simply load the image file once, and call *PDF_fit_image( )* every time you want to place the logo or background on a particular page. You can place the image on multiple pages, or use different scaling factors for different occurrences of the same image (as long as the image hasn't been closed). Depending on the image's size and the number of occurrences, this technique can result in enormous space savings.

Inline images. As opposed to reusable images, which are written to the PDF output as image XObjects, inline images are written directly into the respective content stream (page, pattern, template, or glyph description) . This results in some space savings, but should only be used for small amounts of image data (up to 4 KB) per a recommendation in the PDF reference. The primary use of inline images is for bitmap glyph descriptions in Type 3 fonts.

Inline images can be generated with the *PDF_load_image( )* interface by supplying the *inline* option. Inline images cannot be reused, i.e., the corresponding handle must not be supplied to any call which accepts image handles. For this reason if the *inline* option has been provided *PDF_load_image( )* internally performs the equivalent of

```
PDF_fit_image(p, image, 0, 0, "");
PDF_close_image(p, image);
```

Scaling and dpi calculations. PDFlib never changes the number of pixels in an imported image. Scaling either blows up or shrinks image pixels, but doesn't do any downsampling (the number of pixels in an image will always remain the same). A scaling factor of 1 results in a pixel size of 1 unit in user coordinates. In other words, the image will be imported at 72 dpi if the user coordinate system hasn't been scaled (since there are 72 default units to an inch).

5.1.2 Supported Image File Formats

PDFlib deals with the image file formats described below. If possible, PDFlib passes the compressed image data unchanged to the PDF output since PDF internally supports most compression schemes used in common image file formats. This technique (called *pass-through mode* in the descriptions below) results in very fast image import, since decompressing the image data and subsequent recompression are not necessary. However, PDFlib cannot check the integrity of the compressed image data in this mode. Incomplete or corrupt image data may result in error or warning messages when using the PDF document in Acrobat (e.g., *Read less image data than expected)*.

If an image file can't be imported successfully *PDF_load_image( )* will return an error code by default. However, if you need to know more details about the failure set the *imagewarning* option in *PDF_load_image( )* to *true* (see Section 7.6, »Image and Template Functions«, page 188). Alternatively, the *imagewarning* parameter can be set on a global basis:

```
PDF_set_parameter(p, "imagewarning", "true");          /* enable image warnings */
```

This will cause *PDF_load_image( )* to raise an exception with more details about the failure in the corresponding exception message.

PNG images. PDFlib supports all flavors of PNG images (Portable Network Graphics). PNG images are handled in pass-through mode in most cases. PNG images which make use of interlacing, contain an alpha channel (which will be lost anyway, see below), or have 16 bit color depth will have to be uncompressed, which takes significantly longer than pass-through mode. If a PNG image contains transparency information, the transparency is retained in the generated PDF (see Section 5.1.3, »Image Masks and Transparency«, page 114). However, alpha channels are not supported by PDFlib.

JPEG images. JPEG images are always handled in pass-through mode. PDFlib supports the following flavors of JPEG image compression:

- Baseline JPEG compression which accounts for the vast majority of JPEG images.
- Progressive JPEG compression.

JPEG images can be packaged in several different file formats. PDFlib supports all common JPEG file formats, and will read resolution information from the following flavors:

- JFIF, which is generated by a wide variety of imaging applications.
- JPEG files written by Adobe Photoshop and other Adobe applications. PDFlib applies a workaround which is necessary to correctly process Photoshop-generated CMYK JPEG files.

Note PDFlib does not interpret resolution information from JPEG images in the SPIFF file format, nor color space information from JPEG images in the EXIF file format. JPEG images with multiple scans (this is a very rare flavor) are not supported due to restrictions in Acrobat.

GIF images. GIF images are always handled in pass-through mode (PDFlib does not implement LZW decompression). PDFlib supports the following flavors of GIF images:

- Due to restrictions in the compression schemes supported by the PDF file format, the entry in the GIF file called »LZW minimum code size« must have a value of 8 bits. Unfortunately, there is no easy way to determine this value for a certain GIF file. An image which contains more than 128 distinct color values will always qualify (e.g., a full 8-bit color palette with 256 entries). Images with a smaller number of distinct colors may also work, but it is difficult to tell in advance because graphics programs may use 8 bits or less as LZW minimum code size in this case, and PDFlib may therefore reject the image. The following trick which works in Adobe Photoshop and similar image processing software is known to result in GIF images which are accepted by PDFlib: load the GIF image, and change the image color mode from »indexed« to »RGB«. Now change the image color mode back to »indexed«, and choose a color palette with more than 128 entries, for example the Mac or Windows system palette, or the Web palette.
- The image must not be interlaced.

For other GIF image flavors conversion to the PNG graphics format is recommended.

Note In a particular test case PDFlib converted a GIF image to a PDF file which displays just fine, but results in a PostScript error when printed to a PostScript Level 2 or 3 printer. Since the problem does not occur with Ghostscript, we consider this a bug in the PostScript interpreter. You can work around the problem by selecting PostScript Level 1 output in Acrobat's print dialog.

TIFF images. PDFlib will handle most TIFF images in pass-through mode. PDFlib supports the following flavors of TIFF images:

- compression schemes: uncompressed, CCITT (group 3, group 4, and RLE), ZIP (=Flate), LZW (with restrictions), and PackBits (=RunLength) are handled in pass-through mode; other compression schemes are handled by uncompressing.
- color: black and white, grayscale, RGB, and CMYK images; any alpha channel or mask which may be present in the file will be ignored.
- TIFF files containing more than one image (see Section 5.1.5, »Multi-Page Image Files«, page 117)
- Color depth must be 1, 2, 4, or 8 bits per color sample (this is a requirement of PDF).

Multi-strip TIFF images are converted to multiple images in the PDF file which will visually exactly represent the original image, but can be individually selected with Acrobat's TouchUp object tool. Multi-strip TIFF images can be converted to single-strip images with the *tiffcp* command line tool which is part of the TIFFlib package.[1] The ImageMagick[2] tool always writes single-strip TIFF images.

Some TIFF features (e.g., CIEL*a*b* color space, JPEG compression) and certain combinations of features (e.g., LZW compression and alpha channel or mask, LZW compression and tiling) are not supported.

1. See http://www.libtiff.org
2. See http://www.imagemagick.org

BMP images. BMP images cannot be handled in pass-through mode. PDFlib supports the following flavors of BMP images:

- BMP versions 2 and 3;
- color depth 1, 4, and 8 bits per component, including 3 x 8 = 24 bit TrueColor;
- black and white or RGB color (indexed and direct);
- uncompressed as well as 4-bit and 8-bit RLE compression;
- PDFlib will not mirror images if the pixels are stored in bottom-up order (this is a rarely used feature in BMP which is interpreted differently in applications).

CCITT images. Group 3 or Group 4 fax compressed image data are always handled in pass-through mode. Note that this format actually means raw CCITT-compressed image data, *not* TIFF files using CCITT compression. Raw CCITT compressed image files are usually not supported in end-user applications, but can only be generated with fax-related software. Since PDFlib is unable to analyze CCITT images, all relevant image parameters have to be passed to *PDF_load_image( )* by the client.

Raw data. Uncompressed (raw) image data may be useful for some special applications. The nature of the image is deduced from the number of color components: 1 component implies a grayscale image, 3 components an RGB image, and 4 components a CMYK image.

5.1.3 Image Masks and Transparency

Transparency in PDF. PDF supports various transparency features, all of which are implemented in PDFlib:

- Masking by position: an image may carry the intrinsic information »print the foreground or the background«. This is realized by a 1-bit mask image, and is often used in catalog images.
- Masking by color value: pixels of a certain color are not painted, but the previously painted part of the page shines through instead (»ignore all blue pixels in the image«). In TV and video technology this is also known as bluescreening, and is most often used for combining the weather man and the map into one image.
- PDF 1.4 introduced alpha channels or soft masks. These can be used to create a smooth transition between foreground and background, or to create semi-transparent objects (»blend the image with the background«). Soft masks are represented by 1-component images with 1-8 bits per pixel.

PDFlib supports three kinds of transparency information in images: implicit transparency, explicit transparency, and image masks.

Implicit transparency. In the implicit case, the transparency information from an external image file is respected, provided the image file format supports transparency or an alpha channel (this is not the case for all image file formats). Transparency information is detected in the following image file formats:

- GIF image files may contain a single transparent color value which is respected by PDFlib.
- PNG image files may contain several flavors of transparency information, or a full alpha channel. PDFlib will retain single transparent color values; if multiple color val-

ues with an attached alpha value are given, only the first one with an alpha value below 50 percent is used. A full alpha channel is ignored.

Explicit transparency. The explicit case requires two steps, both of which involve image operations. First, an image must be prepared for later use as a transparency mask. This is accomplished by opening the image with the *mask* option. In PDF 1.3, which supports only 1-bit masks, using this option is required; in PDF 1.4 it is optional. The following kinds of images can be used for constructing a mask:

- PNG images
- TIFF images (only single-strip)
- raw image data

Pixel values of 0 in the mask will result in the corresponding area of the masked image being painted, while high pixel values result in the background shining through. If the pixel has more than 1 bit per pixel, intermediate values will blend the foreground image against the background, providing for a transparency effect. In the second step the mask is applied to another image which itself is acquired through one of the image functions:

```
mask = PDF_load_image(p, "png", maskfilename, 0, "mask");
if (mask == -1)
        return;
sprintf(optlist, "masked %d", mask);
image = PDF_load_image(p, type, filename, optlist)
if (image == -1)
        return;
PDF_fit_image(p, image, x, y, "");
```

Note the different use of the option list for *PDF_load_image()*: *mask* for defining a mask, and *masked* for applying a mask to another image.

The image and the mask may have different pixel dimensions; the mask will automatically be scaled to the image's size.

Note PDFlib converts multi-strip TIFF images to multiple PDF images, which would be masked individually. Since this is usually not intended, this kind of images will be rejected both as a mask as well as a masked target image. Also, it is important to not mix the implicit and explicit cases, i.e., don't use images with transparent color values as mask.

Image masks. Image masks are images with a bit depth of 1 (bitmaps) in which 0-bits are treated as transparent: whatever contents already exist on the page will shine through the transparent parts of the image. 1-bit pixels are colorized with the current fill color. The following kinds of images can be used as image masks:

- PNG images
- TIFF images (single- or multi-strip)
- JPEG images (only as soft mask, see below)
- BMP; note that BMP images are oriented differently than other image types. For this reason BMP images must be reflected along the *x* axis before they can be used as a mask.
- raw image data

Image masks are simply opened with the *mask* option, and placed on the page after the desired fill color has been set:

```
mask = PDF_load_image(p, "tiff", maskfilename, 0, "mask");
PDF_setcolor(p, "fill", "rgb", (float) 1, (float) 0, (float) 0, (float) 0);
if (mask != -1) {
        PDF_fit_image(p, mask, x, y, "");
}
```

If you want to apply a color to an image without the 0-bit pixels being transparent you must use the *colorize* option (see Section 5.1.4, »Colorizing Images«, page 116).

Soft masks. Soft masks generalize the concept of image masks to masks with more than 1 bit. They have been introduced in PDF 1.4 and blend the image against some existing background. PDFlib accepts all kinds of single-channel (grayscale) images as soft mask. They can be used the same way as image masks, provided the PDF output compatibility is at least PDF 1.4.

Ignoring transparency. Sometimes it is desirable to ignore any transparency information which may be contained in an image file. For example, Acrobat's anti-aliasing feature (also known as »smoothing«) isn't used for 1-bit images which contain black and transparent as their only colors. For this reason imported images with fine detail (e.g., rasterized text) may look ugly when the transparency information is retained in the generated PDF. In order to deal with this situation, PDFlib's automatic transparency support can be disabled with the *ignoremask* option when opening the file:

```
image = PDF_load_image(p, "gif", filename, 0, "ignoremask");
```

5.1.4 Colorizing Images

Similarly to image masks, where a color is applied to the non-transparent parts of an image, PDFlib supports colorizing an image with a spot color. This feature works for black and white or grayscale images in the following formats:

- BMP
- PNG
- JPEG
- TIFF (single- or multi-strip)
- GIF

For images with an RGB palette, colorizing is only reasonable when the palette contains only gray values, and the palette index is identical to the gray value. PDFlib does not check this condition, however.

In order to colorize an image with a spot color you must supply the *colorize* option when opening the image, and supply the respective spot color handle which must have been retrieved with *PDF_makespotcolor( )*:

```
PDF_setcolor(p, "both", "cmyk", 1, .79, 0, 0);
spot = PDF_makespotcolor(p, "PANTONE Reflex Blue CV", 0);
sprintf(optlist, "colorize %d", spot);
image = PDF_load_image(p, "tiff", "image.tif", 0, optlist)
if (image != -1) {
        PDF_fit_image(p, image, x, y, "");
}
```

5.1.5 Multi-Page Image Files

PDFlib supports TIFF files which contain more than one image, also known as multi-page files. In order to use multi-page TIFFs, additional string and numerical parameters are used in the call to *PDF_load_image():*

```
image = PDF_load_image(p, "tiff", filename, 0 "page 2");
```

The *page* option indicates that a multi-image file is to be used. The last parameter specifies the number of the image to use. The first image is numbered 1. This option may be increased until *PDF_load_image()* returns -1, signalling that no more images are available in the file.

A code fragment similar to the following can be used to convert all images in a multi-image TIFF file to a multi-page PDF file:

```
for (frame = 1; /* */ ; frame++) {
        sprintf(optlist, "page %d", frame);
        image = PDF_load_image(p, "tiff", filename, 0, optlist);
        if (image == -1)
            break;
        PDF_begin_page(p, width, height);
        PDF_fit_image(p, image, 0.0, 0.0, "");
        PDF_close_image(p, image);
        PDF_end_page(p);
}
```

5.2 Importing PDF Pages with PDI (PDF Import Library)

Note All functions described in this section require PDFlib+PDI. The PDF import library (PDI) is not contained in PDFlib or PDFlib Lite. Although PDI is integrated in all precompiled editions of PDFlib, a license key for PDI (or PPS, which includes PDI) is required.

5.2.1 PDI Features and Applications

When the optional PDI (PDF import) library is attached to PDFlib, pages from existing PDF documents can be imported. PDI contains a parser for the PDF file format, and prepares pages from existing PDF documents for easy use with PDFlib. Conceptually, imported PDF pages are treated similarly to imported raster images such as TIFF or PNG: you open a PDF document, choose a page to import, and place it on an output page, applying any of PDFlib's transformation functions for translating, scaling, rotating, or skewing the imported page. Imported pages can easily be combined with new content by using any of PDFlib's text or graphics functions after placing the imported PDF page on the output page (think of the imported page as the background for new content). Using PDFlib and PDI you can easily accomplish the following tasks:

- overlay two or more pages from multiple PDF documents (e.g., add stationary to existing documents in order to simulate preprinted paper stock);
- place PDF ads in existing documents;
- clip the visible area of a PDF page in order to get rid of unwanted elements (e.g., crop marks), or scale pages;
- impose multiple pages on a single sheet for printing;
- process multiple PDF/X-conforming documents to create a new PDF/X file;
- add some text (e.g., headers, footers, stamps, page numbers) or images (e.g., company logo) to existing PDF pages;
- copy all pages from an input document to the output document, and place barcodes on the pages.

In order to place a PDF background page and populate it with dynamic data (e.g., mail merge, personalized PDF documents on the Web, form filling) we recommend using PDI along with PDFlib blocks (see Chapter 6).

5.2.2 Using PDI Functions with PDFlib

General considerations. It is important to understand that PDI will only import the actual page contents, but not any hypertext features (such as sound, movies, embedded files, hypertext links, form fields, JavaScript, bookmarks, thumbnails, and notes) which may be present in the imported PDF document. These hypertext features can be generated with the corresponding PDFlib functions. PDFlib blocks will also be ignored when importing a page.

You can not re-use individual elements of imported pages with other PDFlib functions. For example, re-using fonts from imported documents for some other content is not possible. Instead, all required fonts must be configured in PDFlib. If multiple imported documents contain embedded font data for the same font, PDI will not remove any duplicate font data. On the other hand, if fonts are missing from some imported PDF, they will also be missing from the generated PDF output file. As an optimization you should keep the imported document open as long as possible in order to avoid the same fonts to be embedded multiple times in the output document.

PDI does not change the color of imported PDF documents in any way. For example, if a PDF contains ICC color profiles these will be retained in the output document.

PDFlib uses the template feature for placing imported PDF pages on the output page. Since some third-party PDF software does not correctly support the templates, restrictions in certain environments other than Acrobat may apply (see Section 3.2.4, »Templates«, page 57).

PDFlib-generated output which contains imported pages from other PDF documents can be processed with PDFlib+PDI again. However, due to restrictions in PostScript printing the nesting level should not exceed 10.

Code fragments for importing PDF pages. Dealing with pages from existing PDF documents is possible with a very simple code structure. The following code snippet opens a page from an existing document, and copies the page contents to a new page in the output PDF document (which must have been opened before):

```
int     doc, page, pageno = 1;
char    *filename = "input.pdf";

...

doc = PDF_open_pdi(p, filename, "", 0);
if (doc == -1) {
        printf("Couldn't open PDF input file '%s'\n", filename);
        exit(1);
}
page = PDF_open_pdi_page(p, doc, pageno, "");
if (page == -1) {
        printf("Couldn't open page %d of PDF file '%s'\n", pageno, filename);
        exit(2);
}

/* dummy page size, will be modified by the adjustpage option */
PDF_begin_page(p, 20, 20);
PDF_fit_pdi_page(p, page, 0, 0, "adjustpage");
PDF_close_pdi_page(p, page);
...add more content to the page using PDFlib functions...
PDF_end_page(p);
```

The last argument to *PDF_fit_pdi_page( )* is an option list which supports a variety of options for positioning, scaling, and rotating the imported page. Details regarding these options are discussed in Section 5.3, »Placing Images and Imported PDF Pages«, page 121.

Dimensions of imported PDF pages. Imported PDF pages are regarded similarly to imported raster images, and can be placed on the output page using *PDF_fit_pdi_page( )*. By default, PDI will import the page exactly as it is displayed in Acrobat, in particular:

- cropping will be retained (in technical terms: if a CropBox is present, PDI favors the CropBox over the MediaBox; see Section 3.2.2, »Page Sizes and Coordinate Limits«, page 55);
- rotation which has been applied to the page will be retained.

Alternatively, you can use the *pdiusebox* option to explicitly instruct PDI to use any of the MediaBox, CropBox, BleedBox, TrimBox or ArtBox entries of a page (if present) for determining the size of the imported page (see Table 7.29 for details).

Many important properties, such as width and height of an imported PDF page, all of the Box entries, and the number of pages in a document, can be queried via PDFlib's parameter mechanism. The relevant parameters are listed in Table 7.28 and Table 7.29. These properties can be useful in making decisions about the placement of imported PDF pages on the output page.

Imported PDF pages with layers. Acrobat 6 (PDF 1.5) introduced the layer functionality (technically known as *optional content)*. PDI will ignore any layer information which may be present in a file. All layers in the imported page, including invisible layers, will be visible in the generated output.

5.2.3 Acceptable PDF Documents

Generally, PDI will happily process all kinds of PDF documents which can be opened with Acrobat, regardless of PDF version number or features used within the file. In order to import pages from encrypted documents (i.e., files with permission settings or password) the corresponding master password must be supplied.

However, in rare cases a PDF document or a particular page of a document may be rejected by PDI.

If a PDF document or page can't be imported successfully *PDF_open_pdi( )* and *PDF_open_pdi_page( )* will return an error code by default. However, if you need to know more details about the failure set the *pdiwarning* option in the option lists of these functions to *true*. Alternatively, the *pdiwarning* parameter can be set on a global basis:

```
PDF_set_parameter(p, "pdiwarning", "true");            /* enable PDI warnings */
```

This will cause *PDF_open_pdi( )* and *PDF_open_pdi_page( )* to raise an exception with more details about the failure in the corresponding exception message. The following kinds of PDF documents can not be imported with PDI:

- PDF documents which use a higher PDF version number than the PDF output document that is currently being generated. The reason is that PDFlib can no longer make sure that the output will actually conform to the requested PDF version after a PDF with a higher version number has been imported. Solution: set the version of the output PDF to the required level using the *compatibility* parameter.
- Files with a damaged cross-reference table. You can identify such files by Acrobat's warning message *File is damaged but is being repaired*. Solution: open and resave the file with Acrobat.

5.3 Placing Images and Imported PDF Pages

The *PDF_fit_image( )* function for placing raster image and templates, as well as *PDF_fit_pdi_page( )* for placing imported PDF pages offer a wealth of options for controlling the placement on the page. This section demonstrates the most important options by looking at some common application tasks. A complete list and descriptions of all options can be found in Table 7.24.

Embedding raster images is easy to accomplish with PDFlib. The image file must first be loaded with *PDF_load_image( )*. This function returns an image handle which can be used along with positioning and scaling options in *PDF_fit_image( )*.

Embedding imported PDF pages works along the same line. The PDF page must be opened with *PDF_open_pdi_page( )* to retrieve a page handle for use in *PDF_fit_pdi_page( )*. The same positioning and scaling options can be used as for raster images.

All samples in this section work the same for raster images, templates, and imported PDF pages. Although code samples are only presented for raster images we talk about placing objects in general. Note that before calling any of the *fit* functions a call to *PDF_load_image( )* or *PDF_open_pdi( )* and *PDF_open_pdi_page( )* must be issued. For the sake of simplicity these calls are not reproduced here.

5.3.1 Scaling, Orientation, and Rotation

Simple Placing. Let's start with the simplest case (see Figure 5.1): an object will be placed at a certain position it its original size:

```
PDF_fit_image(p, image, 80, 100, "");
```

In this code fragment the object will be placed with its lower left corner at the point *(80, 100)* in the user coordinate system. This point is called the reference point. The option list (the last function parameter) is empty. This means the object will be place in its original size at the provided reference point.

Placing with Scaling. The following variation is also very easy to use (see Figure 5.2) We place the object as in the previous example, but will modify the object's scaling:

```
PDF_fit_image(p, image, 80, 100, "scale 0.5");
```

Fig. 5.1
Simple placing

Fig. 5.2
Placing with scaling

This code fragment places the object with its lower left corner at the point *(80, 100)* in the user coordinate system. In addition, the object will be scaled in *x* and *y* direction by a scaling factor of 0.5, which makes it appear at 50 percent of its original size.

Placing with orientation. In the next code fragment we will orientate the object in direction west (see Figure 5.3):

```
PDF_fit_image(p, image, 80, 100, "scale 0.5 orientate west");
```

This code fragment orientates the object towards western direction (90 degrees counterclockwise), and then translates the object's lower left corner (after applying the *orientate* option) to the reference point *(x, y)*. The object will be rotated in itself.

Placing with rotation. Rotating an object (see Figure 5.4) works similarly to orientation. However, it not only affects the placed object but the whole coordinate system. Before placing the object the coordinate system will be rotated at the reference point *(x, y)* by 90 degrees counterclockwise. The rotated object's lower right corner (which is the unrotated object's lower left corner) will end up at the reference point. The function call to achieve this looks as follows:

```
PDF_fit_image(p, image, 80, 100, "scale 0.5 rotate 90");
```

Since there is no translation in this case the object will be partially moved outside the page.

Comparing orientation and rotation. Orientation and rotation are quite similar concepts, but are different nevertheless, and you should be aware of these differences. Figure 5.5 and Figure 5.6 demonstrate the principal difference between the *orientate* and *rotate* options:

- The *orientate* option rotates the object at the reference point *(x, y)* and subsequently translates it. This option supports the direction keywords *north*, *east*, *west*, and *south*.
- The *rotate* option rotates the object at the reference point *(x, y)* without any translation. This option supports arbitrary rotation angles. These have to be specified numerically in degrees (a full circle has 360 degrees).

Fig. 5.3
Placing with orientation

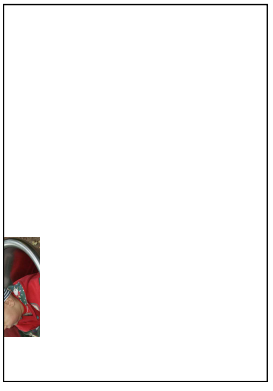


Fig. 5.4
Placing with rotation

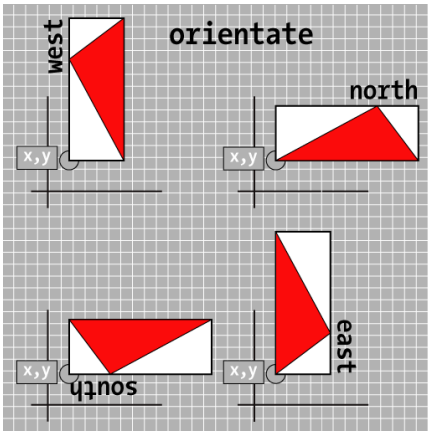


Fig. 5.5
The orientate option

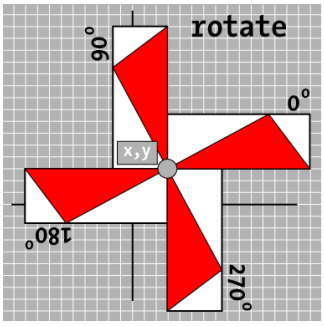


Fig. 5.6
The rotate option

5.3.2 Adjusting the Page Size

In the next example (see Figure 5.7) we will automatically adjust the page size to the object's size. This can be useful, for example, for archiving images in the PDF format. The reference point *(x, y)* can be used to specify whether the page will be exactly the object's size, or somewhat larger or smaller. When enlarging the page size (see Figure 5.7) some border will be kept around the image; when the page size is smaller than the image some parts of the image will be clipped. Let's start with exactly matching the page size to the object's size:

```
PDF_fit_image(p, image, 0, 0, "adjustpage");
```

The next code fragment makes the page size larger by 40 units in *x* and *y* direction than the object, resulting in some border around the object:

```
PDF_fit_image(p, image, 40, 40, "adjustpage");
```

The next code fragment makes the page size smaller by 40 units in *x* and *y* direction than the object. The object will be clipped at the page borders, and some area within the object (with a width of 40 units) will be invisible:

```
PDF_fit_image(p, image, -40, -40, "adjustpage");
```

Fig. 5.7
Adjusting the page size. Left to right: exact, enlarge, shrink

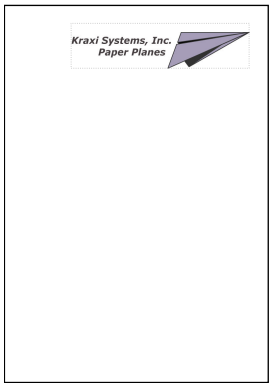


Fig. 5.8
Fitting an object to the box

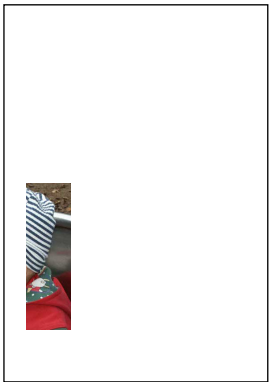


Fig. 5.9
Clipping an object when fitting it to the box

In addition to placing by means of x and y coordinates (which specify the object's distance from the page edges, or the coordinate axes in the general case) you can also specify a target box. This is a rectangular area in which the object will be placed subject to various formatting rules. These can be controlled with the *boxsize*, *fitmethod* and *position* options.

Fitting an object to a box. First, let's place a company logo in the upper right area of the page (see Figure 5.8). The size of the target rectangle where the logo is to appear is fixed. However, we don't know how to scale the logo so that it fits into the box while avoiding any distortion (the ratio of width and height must not be changed). The following statement does the job:

```
PDF_fit_image(p, image, 350, 750, "boxsize {200 100} position 0 fitmethod meet");
```

This code fragment places the lower left corner of a box which is 200 units wide and 100 units high *(boxsize {200 100})* at the point *(350, 750)*. The object's lower left corner will be placed at the box's lower left corner *(position 0)*. The object will be scaled without any distortion to make its height and/or width exactly fit into the box *(fitmethod meet)*.

This concept offers a broad range of variations. For example, the *position* option can be used to specify which point within the object is to be used as the reference point (specified as a percentage of width and height). The *position* option will also specify the reference point within the target box. If both width and height position percentages are identical it is sufficient to specify a single value. For example, *position 50* can be used to select the object's and box's midpoint as reference point for placing the object.

Clipping an object when fitting it to the box. Using another flavor of the *fitmethod* option we can clip the object such that it exactly fits into the target box (see Figure 5.9). In this case the object won't be scaled:

```
PDF_fit_image(p, image, 50, 80, "boxsize {100 400} position 50 fitmethod clip");
```

This code fragment places a box of width 100 and heigth 400 *(boxsize {100 400})* at the coordinates (50, 80). The object will be placed in its original size in the middle of the box *(position 50)*, and will be cropped if it exceeds the box *(fitmethod clip)*.

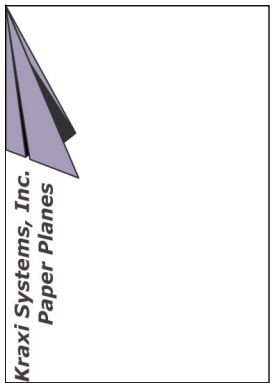


Fig. 5.10
Fitting a logo to the page

Adjusting an object to the page. Adjusting an object to a given page size can easily be accomplished by choosing the page as target box for placing the object. The following statement uses an A4-sized page with dimensions 595 x 842:

```
PDF_fit_image(p, image, 0, 0, "boxsize {595 842} position 0 fitmethod slice");
```

In this code fragment a box is placed at the lower left corner of the page. The size of the box equals the size of an A4 page. The object is placed in the lower left corner of the box and scaled proportionally until it fully covers the box and therefore the page. If the object exceeds the box it will be cropped. Note that *fitmethod slice* results in the object being scaled (as opposed to *fitmethod clip* which doesn't scale the object). Of course the *position* and *fitmethod* options could also be varied in this example.

Fitting a logo to the page. How can we achieve the rotated company logo in Figure 5.10? It is rotated by 90 degrees counterclockwise, starts in the lower left corner, and covers the full height of the page:

```
PDF_fit_image(p, image, 0, 0, "boxsize {595 842} orientate west fitmethod meet");
```

The reference point is (*0, 0*) and orientation is specified as *orientate west*. In order to make the logo cover the full page height we choose the box height to be equal to the page height *(842),* and choose a large enough value for the box's width *(595)*. The logo's proportions should not be changed, therefore we choose *fitmethod meet*.

6 Variable Data and Blocks

PDFlib supports a template-driven PDF workflow for variable data processing. Using the concept of blocks, imported pages can be populated with variable amounts of text, images, or PDF graphics which can be pulled from an external source. This can be used to easily implement applications which require customized PDF documents, for example:

- mail merge
- flexible direct mailings
- transactional and statement processing
- business card personalization

Note The block processing features discussed in this chapter require the PDFlib Personalization Server (PPS). Although PPS is integrated in all precompiled editions of PDFlib, you must purchase a license key for PPS; a PDFlib or PDFlib+PDI license key is not sufficient. The PDFlib Block plugin for Adobe Acrobat is required for creating blocks in PDF templates.

6.1 Overview of the PDFlib Block Concept

6.1.1 Complete Separation of Document Design and Program Code

PDFlib data blocks make it easy to place variable text over imported pages. In contrast to simple PDF pages, pages containing data blocks intrinsically carry information about the required processing which will be performed later on the server side. The PDFlib block concept completely separates the following tasks:

- A designer creates the page layout, and specifies the location of variable text and image elements along with relevant properties such as font size, color, or image scaling. After creating the layout as a PDF document, the designer uses the PDFlib Block plugin for Acrobat to specify variable data blocks and their associated properties.
- A programmer writes code to connect the information contained in PDFlib blocks on imported PDF pages with some dynamic information, e.g., database fields. The programmer doesn't need to know any details about a block (whether it contains a name or a ZIP code, the exact location on the page, its formatting, etc.) and is therefore independent from any layout changes. PDFlib will take care of all block-related details based on the block properties found in the file.

In other words, the code written by the programmer is »data-blind«, i.e., it is generic and does not depend on the particulars of any block. For example, the designer may decide to use the first name of the addressee in a mailing instead of the last name. The generic block handling code doesn't need to be changed, and will generate correct output once the designer changed the block properties with the Acrobat plugin to use the first name instead of the last name.

Example: adding variable text to a template. Adding dynamic text to a PDF template is a very common task. The following code fragment will open a page in an input PDF document (the template), place it on the output page, and fill some variable text into a text block called *firstname:*

```
doc = PDF_open_pdi(p, filename, "", 0);
if (doc == -1) {
        printf("Couldn't open PDF template '%s'\n", filename);
```

```
        return (1);
}
page = PDF_open_pdi_page(p, doc, pageno, "");
if (page == -1) {
        printf("Couldn't open page %d of PDF template '%s'\n", pageno, filename);
        return (2);
}

PDF_begin_page(p, width, height);
PDF_fit_pdi_page(p, page, 0.0, 0.0, "");
PDF_fill_textblock(p, page, "firstname", "Serge", 0, "encoding winansi");
PDF_close_pdi_page(p, page);
PDF_end_page(p);
```

6.1.2 Block Properties

The behavior of blocks can be controlled with block properties. The properties are assigned to a block with the PDFlib Block plugin for Acrobat.

Standard block properties. PDFlib blocks are defined as rectangles on the page which are assigned a name, a type, and an open set of properties which will later be processed on the server side. The name is an arbitrary string which identifies the block, such as *firstname*, *lastname*, or *zipcode*. PDFlib supports the following kinds of blocks:

- Type *Text* means that the block will hold some textual data.
- Type *Image* means that the block will hold a raster image. This is similar to importing a TIFF or JPEG file in a DTP application.
- Type *PDF* means that the block will hold arbitrary PDF graphics imported from a page in another PDF document. This is similar to importing an EPS graphic in a DTP application.

A block may carry a number of standard properties depending on its type. For example, a text block may specify the font and size of the text, an image or PDF block may specify the scaling factor or rotation. For each type of block the PDFlib API offers a dedicated function for processing the block. These functions search an imported PDF page for a block by its name, analyze its properties, and place some client-supplied data (text, raster image, or PDF page) on the new page according to the corresponding block properties.

Custom block properties. Standard block properties make it possible to quickly implement variable data processing applications, but these are limited to the set of properties which are internally known to PDFlib and can automatically be processed. In order to provide more flexibility, the designer may also assign custom properties to a block. These can be used to extend the block concept in order to match the requirements of the most demanding variable data processing applications.

There are no rules for custom properties since PDFlib will not process custom properties in any way, except making them available to the client. The client code can examine the custom properties and act in whatever way it deems appropriate. Based on some custom property of a block the code may make layout-related or data-gathering decisions. For example, a custom property for a scientific application could specify the number of digits for numerical output, or a database field name may be defined as a custom block property for retrieving the data corresponding to this block.

Overriding block properties. In certain situations the programmer would like to use only some of the properties provided in a block definition, but override some other properties with custom values. This can be useful in various situations:

- The scaling factor for an image or PDF page will be calculated instead of taken from the block definition.
- Change the block coordinates programmatically, for example when generating an invoice with a variable number of data items.
- Individual spot color names could be supplied in order to match the requirements of individual customers in a print shop application.

Property overrides can be achieved by supplying property names and the corresponding values in the option list of all *PDF_fill_*block( )* functions as follows:

```
PDF_fill_textblock(p, page, "firstname", "Serge", 0, "fontsize 12");
```

This will override the block's internal *fontsize* property with the supplied value 12. Almost all names of general properties can be used as options, as well as those specific to a particular block type. For example, the *underline* option is only allowed for *PDF_fill_textblock( )*, while the *scale* option is allowed for both *PDF_fill_imageblock( )* and *PDF_fill_pdfblock( )* since *scale* is a valid property for both image and PDF blocks.

Property overrides apply only to the respective function calls; they will not be stored in the block definition.

Coordinate systems. The coordinates describing a block reference the PDF default coordinate system. When the page containing the block is placed on the output page, several positioning and scaling options may be supplied to *PDF_fit_pdi_page( )*. These parameters are taken into account when the block is being processed. This makes it possible to place a template page on the output page multiply, every time filling its blocks with data. For example, a business card template may be placed four times on an imposition sheet. The block functions will take care of the coordinate system transformations, and correctly place the text for all blocks in all invocations of the page. The only requirement is that the client must place the page and then process all blocks on the placed page. Then the page can be placed again at a different location on the output page, followed by more block processing operations referring to the new position, and so on.

Note The Block plugin will display the block coordinates differently from what is stored in the PDF file. The plugin uses Acrobat's convention which has the coordinate origin in the top left corner of the page, while the internal coordinates (those stored in the block) use PDF's convention of having the origin in the bottom left corner of the page.

6.1.3 Why not use PDF Form Fields?

Experienced Acrobat users may ask why we implemented a new block concept for PDFlib, instead of relying on the established form field scheme available in PDF. The primary distinction is that PDF form fields are optimized for interactive filling, and PDFlib blocks are targeted at automated filling. Applications which need both interactive and automated filling can easily achieve this by using a feature which automatically converts form fields to blocks (see Section 6.2.4, »Converting PDF Form Fields to PDFlib Blocks«, page 135).

Although there are many parallels between both concepts, PDFlib blocks offer several advantages over PDF form fields as detailed in Table 6.1.

Table 6.1 Comparison of PDF form fields and PDFlib blocks

Feature	***PDF form fields***	***PDFlib blocks***
design objective	*for interactive use*	*for automated filling*
typographic features (beyond choice of font and font size)	*–*	*kerning, word and character spacing, underline/ overline/strikeout*
font control	*font embedding*	*font embedding and subsetting, encoding*
merged result is integral part of PDF page description	*no*	*yes*
users can edit merged field contents	*yes*	*no*
extensible set of properties	*no*	*yes (custom block properties)*
color support	*RGB*	*grayscale, RGB, CMYK, spot color, Lab*
PDF/X compatible	*no*	*yes (both template with blocks and merged results)*
graphics and text properties can be overridden upon filling	*no*	*yes*

6.2 Creating PDFlib Blocks

6.2.1 Installing the PDFlib Block Plugin

The Block plugin and its sibling, the PDF form field conversion plugin (see Section 6.2.4, »Converting PDF Form Fields to PDFlib Blocks«, page 135), work only with the full version of Acrobat 5, Acrobat 6 Standard and Acrobat 6 Professional. The plugins don't work with Acrobat 6 Elements or any version of Acrobat Reader/Adobe Reader.

Note *If the PDFlib Block plugin doesn't seem to work make sure that in Edit, Preferences, [General...], Startup (Acrobat 6) or Options (Acrobat 5) the »Use only certified plug-ins« box is unchecked.*

Installing the PDFlib Block plugins for Acrobat on Windows. To install the PDFlib Block plugin and the PDF form field conversion plugin in Acrobat 5 or 6 the plugin files must be copied to a subdirectory in the Acrobat plugin folder. This is done automatically by the plugin installer, but can also be done manually. On Windows the plugin files are *Block.api* and *AcroFormConversion.api*, and a typical location of the PDFlib plugin folder looks as follows:

```
C:\Program Files\Adobe\Acrobat 6.0\Acrobat\plug_ins\PDFlib
```

Installing the PDFlib Block plugins for Acrobat on the Mac. With Acrobat 6 the plugin folder will no be visible in the finder. Instead of dragging the plugin files to the plugin folder use the following steps:

- Extract the plugin files by double-clicking the disk image.
- Make sure Acrobat is not running.
- Locate the Acrobat application icon in the finder. It is usually located in a folder which has a name similar to the following:

  ```
  /Applications/Adobe Acrobat 6.0 Professional
  ```

- Single-click on the icon and select *File, Get Info*.
- In the window that pops up click the triangle next to Plug-ins.
- Click *Add...* and select the *PDFlib* folder from the folder which has been created in the first step. Note that the *PDFlib* folder will not immediately show up in the list of plug-ins, but only when you open up the info window next time.

To install the plugins for Acrobat 5, start by double-clicking the disk image. Drag the PDFlib folder to the Acrobat 5 plugin folder. A typical plugin folder name is as follows:

```
/Applications/Adobe Acrobat 5.0/Plug-Ins
```

6.2.2 Creating Blocks interactively with the PDFlib Block Plugin

Using the PDFlib Block tool. The PDFlib Block plugin to create PDFlib blocks is similar to the form tool in Acrobat. All blocks on the page will be visible when the block tool is active. When another Acrobat tool is selected the blocks will be hidden, although they are still present. You can activate the block tool in several ways:

- by clicking the block icon  in Acrobat's *Advanced Editing* toolbar (in Acrobat 5: *Editing* toolbar);
- via the menu item *PDFlib Blocks, PDFlib Block Tool;*

- by using the keyboard shortcut *P* (in Acrobat 6 make sure to enable *Edit, Preferences, [General...], General, Use single key accelerators to access tools*, which is disabled by default).

Creating and modifying blocks. Once you selected the block tool you can simply drag the cross-hair pointer to create a block at the desired position on the page and the desired size. Blocks will always be rectangular with edges parallel to the page edges. When you create a new block the block properties dialog appears where you can edit various properties of the block (see Section 6.2.3, »Editing Block Properties«, page 134). The block tool will automatically create a block name which can be changed in the properties dialog. Block names must be unique within a page. You can change the block type in the General tab to one of Text, Image, or PDF. The General and Custom tabs will always be available, while only one of the Text, Image, and PDF tabs will be active at a time depending on the chosen block type.

To delete one or more blocks, select it with the block tool and press the *Delete* key.

Note After you added blocks or made changes to existing blocks in a PDF, use Acrobat's »Save as...« Command (as opposed to »Save«) to achieve smaller file sizes.

Note When using the Acrobat plugin Enfocus PitStop to edit documents which contain PDFlib blocks you may see the message »This document contains PieceInfo from PDFlib. Press OK to continue editing or Cancel to abort.« This message can be ignored; it is safe to click OK in this situation.

Fine-tuning block size and position. Using the block tool you can select an existing block by clicking on it, and move it to a different position. Hold down the Shift key while dragging a block to restrain the positioning to horizontal and vertical movements. This may be useful for exactly aligning blocks. When the pointer is located near a block corner, the pointer will change to an arrow and you can resize the block. To adjust the position or size of multiple blocks, select two or more blocks and use the *Align, Center, Distribute,* or *Size* commands from the *PDFlib Blocks* menu. The position of one or more blocks can also be changed by using the arrow keys.

Copying one or more blocks. You can create copies of one or blocks by selecting the blocks (use the shift key to select multiple blocks), and dragging them to a new location while pressing the Ctrl key (on Windows) or Alt key (on the Mac). The mouse cursor will change while the key is pressed. A copied block will have the same properties as the original block, with the exception of its name which will automatically be changed.

Creating blocks by selecting an image or graphic. As an alternative to manually dragging block rectangles you can use some page content to define the block size. First, make sure that the menu item *PDFlib Blocks, Click Object to define Block* is active. Now you can use the block tool to click on an image on the page in order to create a block with the size of the image. You can also click on other graphical objects, and the block tool will try to select the surrounding graphic (e.g., a logo). The *Click Object* feature is intended as an aid for defining blocks. If you want to reposition or resize the block you can do so afterwards without any restriction. The block will not be locked to the image or graphics object which was used as a positioning and sizing aid.

The *Click Object* feature will try to recognize which vector graphics and images form a logical element on the page. When some page content is clicked its bounding box (the surrounding rectangle) will be selected unless the object is white or very large. In the

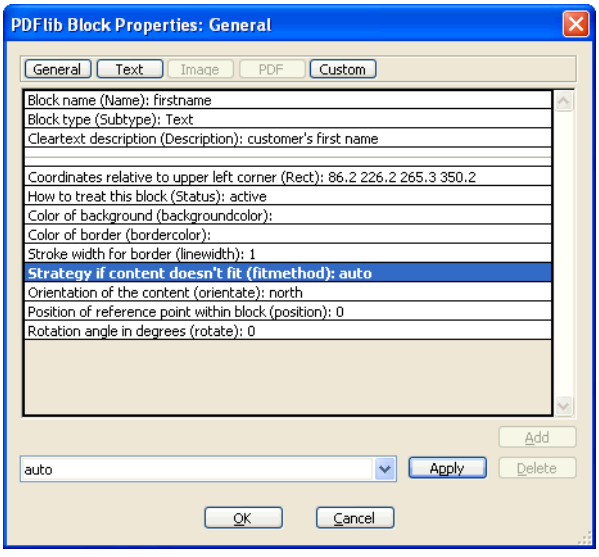


Fig. 6.1
Editing block properties

next step other objects which are partially contained in the detected rectangle will be added to the selected area, and so on. The final area will be used as the basis for the generated block rectangle. The end result is that the *Click Object* feature will try to select complete graphics, and not only individual lines.

The *Click Object* feature isn't perfect: it will not always select what you want depending on the nature of the page content. Keep in mind that this feature is only intended as a positioning aid for quickly placing rectangles.

Automatically detecting font properties. The PDFlib Block plugin can analyze the underlying font which is present on the location where a block is positioned, and can automatically fill in the corresponding properties of the block:

```
fontname, fontsize, fillcolor, charspacing, horizscaling, wordspacing,
textrendering, textrise
```

Since automatic detection of font properties can result in undesired behavior when the background shall be ignored it can be activated or deactivated using *PDFlib Blocks, Detect underlying font*. By default this feature is turned off.

Importing and exporting blocks. Using the export and import features for blocks it is possible to share block definitions for a particular page among multiple PDF files. This is useful for updating the page contents while maintaining existing block definitions. To export block definitions to a separate file proceed as follows:

- Select the block tool.
- Select the blocks you want to export. Press Ctrl-A to select all blocks.
- Choose *PDFlib Blocks, Import and Export, Export...*, and enter a file name.

You can import block definitions via *PDFlib Blocks, Import and Export, Import...* . Upon importing blocks you can choose whether to apply the imported blocks to all pages in the document, or only to one or more selected pages. If more than one page is selected the block definitions will be copied unmodified to the selected pages.

As an alternative to importing/exporting blocks you can also replace the underlying page(s) while keeping the blocks. Use *Document, Pages, Replace* (Acrobat 6) or *Document, Replace Pages...* (Acrobat 5).

Locking blocks. Blocks can be locked to protect them against accidental moving, resizing, or deleting. With the block tool active, select the block, activate its context menu (using the right mouse button on Windows, or ctrl-click on the Mac), and select *Lock*. While a block is locked you cannot move, resize, or delete it, nor display its properties dialog.

Using Blocks with PDF/X. Unlike PDF form fields, PDFlib blocks are PDF/X-compatible. Both the input document containing blocks, as well as the generated output PDF can be made PDF/X conforming. However, in preparing block files for a PDF/X workflow you may run into the following problem:

- PDF/X-1, PDF/X-1a, and PDF/X-3 are based on PDF 1.3, and do not support Acrobat 5 files;
- The PDFlib Block plugin requires Acrobat 5 or above.

How to work around this potential deadlock depends on your Acrobat version:

- Acrobat 6: You can save the file as PDF 1.3 directly from Acrobat using *File, Reduce File Size...*, and choosing *Acrobat 4.0 and later*.
- Acrobat 5: For saving the generated PDF with blocks in the PDF/X-conforming PDF version 1.3 use an additional plugin by callas software called *pdfSaveAsPDF1.3*. Fully functional demo versions are available on the callas web site[1].

6.2.3 Editing Block Properties

When you create a new block or double-click an existing one the properties dialog will appear where you can edit a block's properties (see Figure 6.1). As detailed in Section 6.3, »Standard Properties for automated Processing«, page 137, there are several types of properties:

- Properties in the *General* tab apply to all blocks.
- Properties in the *Text, Image,* and *PDF* tabs apply only to the respective block type. Only the tab corresponding to the current block's type will be active, while the other two tabs are inactive.
- Properties in the *Custom* tab can be defined by the user, and apply to any block type.

To change a property's value, select it in the property list, change its value in the lower part of the dialog, and click *Apply*. For some properties you can also click the »...« symbol in order to define the value using a dedicated dialog box. When you are done editing properties click OK to close the properties dialog. The properties just defined will be stored in the PDF file as part of the block definition.

Default properties. In order to save some amount of typing and clicking the block tool will remember the property values which have been entered into the previous block's properties dialog. These values will be reused when you create a new block. Of course you can override these values with different ones at any time.

1. See http://www.callassoftware.com

Shared properties. By holding the shift key and using the block tool to select several blocks you can select an arbitrary number of blocks. The *Enter* key will display the properties dialog which now applies to all selected blocks. However, since not all properties can be shared among multiple blocks, only a subset of all properties will be available for editing. Section 6.3, »Standard Properties for automated Processing«, page 137, details which properties can be shared among multiple blocks. Custom properties cannot be shared.

6.2.4 Converting PDF Form Fields to PDFlib Blocks

As an alternative to creating PDFlib blocks manually you can automatically convert PDF form fields to blocks. This is especially convenient if you have complicated PDF forms which you want to fill automatically with PDFlib, or need to convert a large number of existing PDF forms for automated filling. In order to convert all form fields on a page to PDFlib blocks choose *PDFlib Blocks, Convert Form Fields, Current Page*. To convert all form fields in a document choose *All Pages* instead. Finally, you can convert only selected form fields (choose Acrobat's Form Tool or the Select Object Tool to select form fields) with *Selected Form Fields*.

Form field conversion details. Automatic form field conversion will convert form fields of the types selected in the *PDFlib Blocks, Convert Form Fields, Conversion Options...* dialog to blocks of type *Text*. By default all form field types will be converted. Attributes of the converted fields will be transformed to the corresponding block properties according to Table 6.2.

Table 6.2 Conversion of PDF form fields to PDFlib blocks

PDF form field attribute...	***...will be converted to the PDFlib block property***
all fields	
Position	*General, Rect*
Name	*General, Name*
Short Description	*General, Description*
Appearance, Text, Font	*Text, fontname*
Appearance, Text, Size	*Text, fontsize; »auto« font size will be converted to a fixed font size of 2/3 of the block height, and the fitmethod will be set to »auto«*
Appearance, Text, Text Color	*Text, textcolor and Text, fillcolor*
Appearance, Border, Border Color	*General, bordercolor*
Appearance, Border, Background Color	*General, backgroundcolor*
Appearance, Border, Width	*General, linewidth: Thin=1, Medium=2, Thick=3*
Appearance, Common Properties, Form Field is...	*General, Status: Visible=active, Hidden=ignore, Visible but doesn't print=ignore, Hidden but printable=active*
Appearance, Common Prop., Orientation	*General, orientate: 0=north, 90=west, 180=south, 270=east*
text fields	
Options, Default	*Text, defaulttext*
Options, Alignment	*General, position: Left={0 50}, Center={50 50}, Right={100, 50}*
radio buttons and check boxes	
If »Default is Checked« is selected: Options, Radio Style and Options, Check Style	*Text, defaulttext: Check=4, Circle=l, Cross=8, Diamond=u, Square=n, Star=H (these characters represent the respective symbols in the ZapfDingbats font)*

Table 6.2 Conversion of PDF form fields to PDFlib blocks

PDF form field attribute...	***...will be converted to the PDFlib block property***
List boxes and combo boxes	
Options, Selected (default) item	*Text, defaulttext*
buttons	
Options, Button Face Attributes, Text	*Text, defaulttext*

Binding blocks to the corresponding form fields. In order to keep PDF form fields and the generated PDFlib blocks synchronized, the generated blocks can be bound to the corresponding form fields. This means that the block tool will internally maintain the relationship of form fields and blocks. When the conversion process is activated again, bound blocks will be updated to reflect the attributes of the corresponding PDF form fields. Bound blocks are useful to avoid duplicate work: when a form is updated for interactive use, the corresponding blocks can automatically be updated, too.

If you do not want to keep the converted form fields after blocks have been generated you can choose the option *Delete converted Form Fields* in the *PDFlib Blocks, Convert Form Fields, Conversion Options...* dialog. This option will permanently remove the form fields after the conversion process. Any actions (e.g., JavaScript) associated with the affected fields will also be removed from the document.

Batch conversion. If you have many PDF documents with form fields that you want to convert to PDFlib blocks you can automatically process an arbitrary number of documents using the batch conversion feature. The batch processing dialog is available via *PDFlib Blocks, Convert Form Fields, Batch conversion...:*

- The input files can be selected individually; alternatively the full contents of a folder can be processed.
- The output files can be written to the same folder where the input files are, or to a different folder. The output files can receive a prefix to their name in order to distinguish them from the input files.
- When processing a large number of documents it is recommended to specify a log file. After the conversion it will contain a full list of processed files as well as details regarding the result of each conversion along with possible error messages.

During the conversion process the converted PDF documents will be visible in Acrobat, but you cannot use any of Acrobat's menu functions or tools.

6.3 Standard Properties for automated Processing

PDFlib supports general properties which can be assigned to any type of block. In addition there are properties which are specific to the block types *Text*, *Image*, and *PDF*. Some properties are *shared*, which means that they can be assigned to multiple blocks at once using the Block plugin.

Properties support the same data types as option lists (see Section 3.1.4, »Option Lists«, page 44) except handles.

Many block properties have the same name as options for *PDF_fit_image( )* (e.g., *fitmethod)* and other functions, or as PDFlib parameters (e.g., *charspacing)*. In these cases the behavior is exactly the same as the one documented for the respective option or parameter.

Property processing in PDFlib. The PDFlib Block functions *PDF_fill_*block( )* will process block properties in the following order:

- If the *backgroundcolor* property is present and contains a color space keyword different from *None,* the block rectangle will be filled with the specified color.
- All other properties except *bordercolor* and *linewidth* will be processed.
- If the *bordercolor* property is present and contains a color space keyword different from *None,* the block rectangle will be stroked with the specified color and linewidth.

There will be no clipping; if you want to make sure that the block contents do not exceed the block rectangle choose *fitmethod clip*.

If a separation color is used in a block property the specified spot color name must either be known to PDFlib internally (see Section 3.3.3, »Spot Colors«, page 60), or must have been specified earlier in the PDFlib client program using *PDF_makespotcolor( )*. Otherwise the block functions will fail.

General properties. General properties apply to all kinds of blocks *(Text, Image, PDF)*. They are required for block administration, describe the appearance of the block rectangle itself, and manage how the contents will be placed within block. Required entries will automatically be generated by the PDFlib Block Plugin. Table 6.3 lists the general properties.

Table 6.3 General block properties

keyword	*type*	*possible values and explanation*
block administration		
Name	*string*	*(Required) Name of the block; maximum length is 127 bytes. Block names must be unique within a page, but not within a document. The three characters [ ] / are not allowed in block names.*
Description	*string*	*Human-readable description of the block's function, coded in PDFDocEncoding or Unicode (in the latter case starting with a BOM). This property is for user information only, and will be ignored when processing the block.*
Locked	*boolean*	*(Shareable) If true, the block and its properties can not be edited with the Block plugin. This property will be ignored when processing the block. Default: false.*
Rect	*float list*	*(Required) Four coordinates of the block (lower left and upper right corner). The origin of the coordinate system is in the lower left corner of the page. However, the Block plugin will display the coordinates in Acrobat's notation, i.e., with the origin in the upper left corner of the page.*

Table 6.3 General block properties

keyword	type	possible values and explanation
Status	keyword	Keyword describing how this block will be processed (Default: active): active The block will be fully processed according to its properties. ignore The block will be ignored. static No variable contents will be placed; instead, the block's default text, image, or PDF contents will be used if available.
Subtype	keyword	(Required) Depending on the block type, one of Text, Image, or PDF.
Type	keyword	(Required) Always Block
block appearance		
backgroundcolor	color	(Shareable) If this property is present and contains a color space keyword different from None, a rectangle will be drawn and filled with the supplied color. This may be useful to cover existing page contents. Default: None.
bordercolor	color	(Shareable) If this property is present and contains a color space keyword different from None, a rectangle will be drawn and stroked with the supplied color. Default: None
linewidth	float	(Shareable) Stroke width of the line used to draw the block rectangle; only used if strokecolor is set. Default: 1
content placing		
fitmethod	keyword	(Shareable) Strategy to use if the supplied content doesn't fit into the box. Possible values are auto, nofit, clip, meet, slice, and entire (see Table 7.24). Default: nofit
orientate	keyword	(Shareable) Specifies the desired orientation of the content when it is placed (see Table 7.24). Possible values are north, east, south, west. Default: north
position	float list	(Shareable) One or two values specifying the position of the reference point within the content (see Table 7.24). Default: 0
rotate	float	(Shareable) Rotation angle in degrees by which the block will be rotated counter-clockwise before processing begins. The center of the rotation is the reference point. Default: 0

Text-related properties. Text-related properties apply to blocks of type *Text* (in addition to general properties). All text-related properties can be shared. The encoding for the text must be specified as an option for *PDF_fill_textblock( )* when filling the block. Table 6.4 lists the text-related properties.

Table 6.4 Text-related block properties

keyword	type	possible values and explanation
charspacing	float	The character spacing (see Table 7.9). Default: 0
defaulttext	string	Text which will be used if no substitution text is supplied by the client[1]
fillcolor	color	Fill color of the text. Default: gray 0 (=black)
fontname[2]	string	Name of the font as required by PDF_load_font( )
fontsize[2]	float	Size of the font in points
fontstyle	keyword	Font style, must be one of normal, bold, italic, or bolditalic (see Table 7.7)
horizscaling	float	The horizontal text scaling (see Table 7.9). Default: 100
kerning	boolean	Kerning behavior (see Table 7.9). Default: false
margin	float list	One or two float values describing additional horizontal and vertical extensions of the text box (see Table 7.10). Default: 0
overline	boolean	Overline mode (see Table 7.9). Default: false

Table 6.4 Text-related block properties

keyword	type	possible values and explanation
strikeout	boolean	Strikeout mode (see Table 7.9). Default: false
strokecolor	color	Stroke color of the text. Default: gray 0 (=black)
textrendering	int	The text rendering mode (see Table 4.4). Default: 0
textrise	float	The text rise parameter (see Table 7.9). Default: 0
underline	boolean	Underline mode (see Table 7.9). Default: false
wordspacing	float	The word spacing (see Table 7.9). Default: 0

1. The text will be interpreted in winansi encoding or Unicode.
2. This property is required in a text block; it will automatically be enforced by the PDFlib Block plugin.

Image-related properties. Image-related properties apply to blocks of type *Image* (in addition to general properties). All image-related properties can be shared. Table 6.5 lists the image-related properties.

Table 6.5 Image-related block properties

keyword	type	possible values and explanation
defaultimage	string	Path name of an image which will be used if no substitution image is supplied by the client. It is recommended to use file names without absolute paths, and use the SearchPath feature (see Section 3.1.6, »Resource Configuration and File Searching«, page 47) in the PDFlib client application. This will make block processing independent from platform and file system details. When the block will be processed the image must be supplied in a format which is supported by PDF_load_image( ).
dpi	float list	One or two values specifying the desired image resolution in pixels per inch in horizontal and vertical direction. With the value 0 the image's internal resolution will be used if available, or 72 dpi otherwise. This property will be ignored if the fitmethod property has been supplied with one of the keywords meet, slice, or entire. Default: 0.
scale	float list	One or two values specifying the desired scaling factor(s) in horizontal and vertical direction. This option will be ignored if the fitmethod property has been supplied with one of the keywords meet, slice, or entire. Default: 1

PDF-related properties. PDF-related properties apply to blocks of type *PDF* (in addition to general properties). All PDF-related properties can be shared. Table 6.6 lists the PDF-related properties.

Table 6.6 PDF-related block properties

keyword	type	possible values and explanation
defaultpdf	string	Path name of a PDF document which will be used if no substitution PDF is supplied by the client. It is recommended to use file names without absolute paths, and use the SearchPath feature (see Section 3.1.6, »Resource Configuration and File Searching«, page 47) in the PDFlib client application. This will make block processing independent from platform and file system details.
defaultpdfpage	float	Page number of the page in the default PDF document. Default: 1
scale	float list	One or two values specifying the desired scaling factor(s) in horizontal and vertical direction. This option will be ignored if the fitmethod property has been supplied with one of the keywords meet, slice, or entire. Default: 1

Custom properties. Custom properties apply to blocks of any type of block (in addition to general and type-specific properties). Custom properties are optional, and can not be shared. Table 6.7 lists the PDF-related properties.

Table 6.7 Custom block properties

keyword	*type*	*possible values and explanation*
any name not containing the three characters [ ] /	*string, name, float, float list*	*The interpretation of the values corresponding to custom properties is completely up to the client application.*

6.4 Querying Block Names and Properties

In addition to automatic block processing PDFlib supports some features which can be used to enumerate block names and query standard and custom properties.

Finding the number and names of Blocks. The client code must not even know the names or number of the blocks on an imported page since these can also be queried. The following statement returns the number of blocks on the page:

```
blockcount = PDF_get_pdi_value(p, "vdp/blockcount", doc, page, 0);
```

The following statement returns the name of block number *5* on the page (block counting starts at 0), or an empty string if no such block exists (however, an exception will be thrown if the *pdiwarning* parameter is set to *true):*

```
blockname = PDF_get_pdi_parameter(p, "vdp/Blocks[5]/Name", doc, page, 0, &len);
```

The returned block name can further be used to automatically fill the blocks using PDFlib's fill functions, or query block properties, or populate the block with text, image, or PDF content.

In the path syntax for addressing block properties the following expressions are equivalent, assuming that the block with the sequential <*number*> has its *Name* property set to <*blockname*>*:*

```
Blocks[<number>]/
Blocks/<blockname>/
```

Finding Block coordinates. The two coordinate pairs *(llx, lly)* and *(urx, ury)* describing the lower left and upper right corner of a block named *foo* can be queried as follows:

```
llx = PDF_get_pdi_value(p, "vdp/Blocks/foo/Rect[0]", doc, page, 0);
lly = PDF_get_pdi_value(p, "vdp/Blocks/foo/Rect[1]", doc, page, 0);
urx = PDF_get_pdi_value(p, "vdp/Blocks/foo/Rect[2]", doc, page, 0);
ury = PDF_get_pdi_value(p, "vdp/Blocks/foo/Rect[3]", doc, page, 0);
```

Note that these coordinates are provided in the default user coordinate system (with the origin in the bottom left corner, possibly modified by the page's CropBox), while the Block plugin displays the coordinates according to Acrobat user interface coordinate system with an origin in the top left corner of the page. Also note that the *topdown* parameter is not taken into account when querying block coordinates.

Querying custom properties. Custom properties can be queried as in the following example, where the property *zipcode* is queried from a block named *b1:*

```
zip = PDF_get_pdi_parameter(p, "vdp/Blocks/b1/Custom/zipcode", doc, page, 0, &len);
```

Name space for custom properties. In order to avoid confusion when PDF documents from different sources are exchanged, it is recommended to use an Internet domain name as a company-specific prefix in all custom property names, followed by a colon ':' and the actual property name. For example, ACME corporation would use the following property names:

```
acme.com:digits
acme.com:refnumber
```

Since standard and custom properties are stored differently in the block, standard PDFlib property names (as defined in Section 6.3, »Standard Properties for automated Processing«, page 137) will never conflict with custom property names.

6.5 PDFlib Block Specification

The PDFlib Block syntax is fully compliant with the PDF Reference, which specifies an extension mechanism which allows applications to store private data attached to the data structures comprising a PDF page. A detailed description of the PDFlib block syntax is provided here for the benefit of users who wish to create PDFlib blocks by other means than the PDFlib block plugin. Plugin users can safely skip this section.

6.5.1 PDF Object Structure for PDFlib Blocks

The page dictionary contains a */PieceInfo* entry, which has a dictionary as value. This dictionary contains the key */PDFlib* with an application data dictionary as value. The application data dictionary contains two standard keys listed in Table 6.8.

Table 6.8 Entries in a PDFlib application data dictionary

Key	type	value
LastModified	*date string*	*(Required) The date and time when the blocks on the page were created or most recently modified.*
Private	*dictionary*	*(Required) A block list (see Table 6.9)*

A Block list is a dictionary containing general information about block processing, plus a list of all blocks on the page. Table 6.9 lists the keys in a block list dictionary.

Table 6.9 Entries in a block list dictionary

Key	type	value
Version	*number*	*(Required) The version number of the block specification to which the file complies. This document describes version x of the block specification.*
Blocks	*dictionary*	*(Required) Each key is a name object containing the name of a block; the corresponding value is the block dictionary for this block (see Table 6.11). The /Name key in the block dictionary must be identical to the block's name in this dictionary.*
PluginVersion	*string*	*(Required unless the pdfmark key is present[1]) A string containing a version identification of the PDFlib Block plugin which has been used to create the blocks.*
pdfmark	*boolean*	*(Required unless the PluginVersion key is present[1]) Must be true if the block list has been generated by use of pdfmarks.*

1. Exactly one of the PluginVersion and pdfmark keys must be present.

Data types for block properties. Properties support the same data types as option lists (see Section 3.1.4, »Option Lists«, page 44) except handles. Table 6.10 details how these types are mapped to PDF data types.

Table 6.10 Data types for block properties

block type	PDF type	remarks
boolean	*boolean*	
string	*string*	
keyword	*name*	*It is an error to provide keywords outside the list of keywords supported by a particular property.*

Table 6.10 Data types for block properties

block type	PDF type	remarks
float, integer	*number*	*While option lists support both point and comma as decimal separators, PDF numbers support only point.*
color	*array with two elements*	*The first element in the array specifies a color space, and the second element specifies a color value as follows. The following entries are supported for the first element in the array:* */DeviceGray* — *The second element is a single gray value.* */DeviceRGB* — *The second element is an array of three RGB values.* */DeviceCMYK* — *The second element is an array of four CMYK values.* *[/Separation/spotname]* — *The first element is an array containing the keyword /Separation and a color name. The second element is a tint value.* *[/Lab]* — *The first element is an array containing the keyword /Lab. The second element is an array of three Lab values.* *To specify the absence of color the respective property must be omitted.*

Block dictionary keys. Block dictionaries may contain the keys in Table 6.11. Only keys from one of the Text, Image or PDF groups may be present depending on the */Subtype* key in the General group.

Table 6.11 Entries in a block dictionary

Key	type	value
General block properties		*(Some keys are required) General block properties according to Table 6.3*
text-related properties		*(Optional) Text-related properties according to Table 6.4*
image-related properties		*(Optional) Image-related properties according to Table 6.5*
PDF-related properties		*(Optional) PDF-related properties according to Table 6.6*
Custom	*dict*	*(Optional) A dictionary containing key/value pairs for custom properties according to Table 6.7.*
Internal	*dict*	*(Optional) This key is reserved for private use, and applications should not depend on its presence or specific behavior. Currently it is used for maintaining the relationship between converted form fields and corresponding blocks.*

Example. The following fragment shows the PDF code for two blocks, a text block called *job_title* and an image block called *logo*. The text block contains a custom property called *format:*

```
<<
        /Contents 12 0 R
        /Type /Page
        /Parent 1 0 R
        /MediaBox [ 0 0 595 842 ]
        /PieceInfo << /PDFlib 13 0 R >>
>>

13 0 obj
<<
        /Private <<
            /Blocks <<
                /job_title 14 0 R
```

```
                /logo 15 0 R
            >>
            /Version 2
            /PluginVersion (1.1.0)
        >>
        /LastModified (D:20030913200730)
>>
endobj

14 0 obj
<<
        /Type /Block
        /Rect [ 70 740 200 800 ]
        /Name /job_title
        /Subtype /Text
        /fitmethod /auto
        /fontname (Helvetica)
        /fontsize 12
        /Custom << /format 5 >>
>>
endobj

15 0 obj
<<
        /Type /Block
        /Rect [ 250 700 400 800 ]
        /Name /logo
        /Subtype /Image
        /fitmethod /auto
>>
```

6.5.2 Generating PDFlib Blocks with pdfmarks

As an alternative to creating PDFlib blocks with the plugin, blocks can be created by inserting appropriate pdfmark commands into a PostScript stream, and distilling it to PDF. Details of the pdfmark operator are discussed in the Acrobat documentation. The following fragment shows pdfmark operators which can be used to generate the block definition in the preceding section:

```
% ---------- Setup for the blocks on a page ----------
[/_objdef {B1} /type /dict /OBJ pdfmark         % Blocks dict

[{ThisPage} <<
        /PieceInfo <<
            /PDFlib <<
                /LastModified (D:20030913200730)
                /Private <<
                    /Version 2
                    /pdfmark true
                    /Blocks {B1}
                >>
            >>
        >>
>> /PUT pdfmark

% ---------- text block ----------
[{B1} <<
        /job_title <<
```

```
            /Type /Block
            /Name /job_title
            /Subtype /Text
            /Rect [ 70 740 200 800 ]
            /fitmethod /auto
            /fontsize 12
            /fontname (Helvetica)
            /Custom << /format 5 >>
        >>
>> /PUT pdfmark

% ---------- image block ----------
[{B1} <<
        /logo <<
            /Type /Block
            /Name /logo
            /Subtype /Image
            /Rect [ 250 700 400 800 ]
            /fitmethod /auto
        >>
>> /PUT pdfmark
```

7 API Reference for PDFlib, PDI, and PPS

The API reference documents all supported functions of PDFlib, PDI (PDF Import) and PPS (PDFlib Personalization Server).

7.1 Data Types and Naming Conventions

PDFlib data types. The exact syntax to be used for a particular language binding may actually vary slightly from the C syntax shown here in the reference. This especially holds true for the PDF document parameter *(PDF *)* in the API reference) which has to be supplied as the first argument to almost all PDFlib functions in the C binding, but not those bindings which hide the PDF document parameter in an object created by the language wrapper.

Table 7.1 details the use of the PDF document type and the string type in all language bindings. The data types *integer*, *long*, and *float* are not mentioned since there is an obvious mapping of these types in all bindings. Please refer to the respective language section and the examples in Chapter 2 for more language-specific details.

Table 7.1 Data types in the language bindings

language binding	p parameter?	PDF_ prefix?	string data type	binary data type
C (also used in this API reference)	yes	yes	const char * [1]	const char *
C++	no	no	string[2]	char *
Cobol[3]	yes	no[4]	STRING	STRING
Java	no	no	String	byte[ ]
Perl	yes	yes	string	string
PHP	yes	yes	string	string
Python	yes	yes	string	string
RPG	yes	yes	string, but must add x'00'	data
Tcl	yes	yes	string	byte array

1. C language NULL string values and empty strings are considered equivalent.
2. NULL string values must not be used in the C++ binding.
3. See Section 2.2.1, »Special Considerations for Cobol«, page 18 for more information on Cobol data types.
4. Cobol programs must use abbreviated names for the PDFlib functions.

Naming conventions for PDFlib functions. In the C binding, all PDFlib functions live in a global namespace and carry the common *PDF_* prefix in their name in order to minimize namespace pollution. In contrast, several language bindings hide the PDF document parameter in an object created by the language wrapper. For these bindings, the function name given in this API reference must be changed by omitting the *PDF_* prefix and the *PDF** parameter used as first argument. For example, the C-like API description

```
PDF *p;
PDF_open_file(PDF *p, const char *filename);
```

translates to the following when the function is used from Java:

```
pdflib p;
p.open_file(String filename);
```

7.2 General Functions

7.2.1 Setup

Table 7.2 lists relevant parameters and values for this section.

Table 7.2 Parameters and values for the setup functions

function	key	explanation
set_parameter	compatibility	Set PDFlib's compatibility mode to one of the strings »1.3«, »1.4«, or »1.5« for Acrobat 4, 5, or 6. See Section 1.4, »Acrobat Versions and PDFlib Features«, page 16 for details. This parameter must be set before the first call to PDF_open_*( ). It will be ignored if the pdfx parameter is used. Scope: object. Default: »1.4«.
set_parameter	pdfx	Set the PDF/X conformance level to one of »PDF/X-1:2001« , »PDF/X-1a:2001«, »PDF/X-3:2002«, or »none« (see Section 3.4, »PDF/X Support«, page 67). Scope: object. Default: none
set_parameter	flush	Set PDFlib's flushing strategy to none, page, content, or heavy. See Section 3.1.7, »Generating PDF Documents in Memory«, page 50 for details. This parameter is only available in the C and C++ language bindings. Scope: any. Default: page
set_parameter	SearchPath	(Not supported on MVS) Relative or absolute path name of a directory containing files to be read. The SearchPath can be set multiply; the entries will be accumulated and used in least-recently-set order (see Section 3.1.6, »Resource Configuration and File Searching«, page 47). Scope: any.
set_parameter	prefix	(Deprecated) Resource file name prefix as used in a UPR file. The prefix can be set multiply. It contains a slash character plus a path name, which in turn may start with a slash. Scope: any.
set_parameter	resourcefile	Relative or absolute file name of the PDFlib UPR resource file. The resource file will be loaded immediately. Existing resources will be kept; their values will be overridden by new ones if they are set again. Scope: any.
set_parameter	asciifile	(Only supported on iSeries and zSeries). Expect text files (PFA, AFM, UPR, encodings) in ASCII encoding. Default: true on iSeries; false on zSeries. Scope: any.
set_parameter	license	Set the license key for PDFlib, PDFlib+PDI, or PPS.The license key can only be set once before the first call to PDF_begin_page( ). Scope: object.
set_parameter	licensefile	Set the name of a file containing the license keyThe license file can only be set once before the first call to PDF_begin_page( ). Scope: object.
set_value	compress	Set the compression parameter to a value from 0–9. This parameter does not affect image data handled in pass-through mode. Scope: page, document. 0 no compression 1 best speed 6 default value 9 best compression
get_value	major, minor revision	Return the major, minor, or revision number of PDFlib, respectively. Scope: any, null[1].
get_parameter	version	Return the full PDFlib version string in the format <major>.<minor>.<revision>, possibly suffixed with additional qualifiers such as beta, rc, etc. Scope: any, null[1].
get_parameter	scope	Return the name of the current scope (see Table 3.1). Scope: any.
set_parameter	trace	If true, all API function calls will be logged to a trace file. The contents of the trace file may be useful for debugging purposes, or may be requested by PDFlib support. Scope: any. Default: false.
set_parameter	tracefile	Set trace file name. Scope: any, but before enabling tracing. Default: PDFlib.trace.
set_parameter	tracemsg	If tracing is enabled, the supplied message text will be written to the trace file in addition to API calls. This may be useful for debugging client code. Scope: any.

*1. May be called with a PDF * argument of NULL or 0.*

void PDF_boot(void)
void PDF_shutdown(void)

Boot and shut down PDFlib, respectively.

Scope *null*

Bindings C: Recommended for the C language binding, although currently not required.

Other bindings: For all other language bindings booting and shutting down is accomplished automatically by the wrapper code, and these functions are not available.

PDF *PDF_new(void)

Create a new PDFlib object with default settings.

Details This function creates a new PDFlib object, using PDFlib's internal default error handling and memory allocation routines.

Returns A handle to a PDFlib object which is to be used in subsequent PDFlib calls. If this function doesn't succeed due to unavailable memory it will return NULL (in C) or throw an exception.

Scope *null;* this function start object scope, and must always be paired with a matching *PDF_delete( )* call.

Bindings The data type used for the opaque PDFlib object handle varies among language bindings. This doesn't really affect PDFlib clients, since all they have to do is pass the PDF handle as the first argument to all functions.

C: In order to load the PDFlib DLL dynamically at runtime use *PDF_new_dl( )* instead (see Section 2.4.3, »Using PDFlib as a DLL loaded at Runtime«, page 22). *PDF_new_dl( )* will return a pointer to a *PDFlib_api* structure filled with pointers to all PDFlib API functions. If the DLL cannot be loaded, or a mismatch of major or minor version number is detected, NULL will be returned.
C++: this function is not available since it is hidden in the PDF constructor.
COM, Java: this function is automatically called by the wrapper code, and therefore not available.

PDF *PDF_new2(void (*errorhandler)(PDF *p, int errortype, const char *msg),
void* (*allocproc)(PDF *p, size_t size, const char *caller),
void* (*reallocproc)(PDF *p, void *mem, size_t size, const char *caller),
void (*freeproc)(PDF *p, void *mem),
void *opaque)

Create a new PDFlib object with client-supplied error handling and memory allocation routines.

errorhandler Pointer to a user-supplied error-handling function. The error handler will be ignored in *PDF_TRY/PDF_CATCH* blocks.

allocproc Pointer to a user-supplied memory allocation function.

reallocproc Pointer to a user-supplied memory reallocation function.

freeproc Pointer to a user-supplied free function.

opaque Pointer to some user data which may be retrieved later with *PDF_get_opaque( )*.

Returns A handle to a PDFlib object which is to be used in subsequent PDFlib calls. If this function doesn't succeed due to unavailable memory it will return NULL (in C) or throw an exception.

Details This function creates a new PDFlib object with client-supplied error handling and memory allocation routines. Unlike *PDF_new( )*, the caller may optionally supply own procedures for error handling and memory allocation. The function pointers for the error handler, the memory procedures, or both may be NULL. PDFlib will use default routines in these cases. Either all three memory routines must be provided, or none.

Scope *null;* this function starts *object* scope, and must always be paired with a matching *PDF_delete( )* call. No other PDFlib function with the same PDFlib object must be called after calling this function.

Bindings C++: this function is indirectly available via the PDF constructor. Not all function arguments must be given since default values of NULL are supplied. All supplied functions must be »C« style functions, not C++ methods.

---

void PDF_delete(PDF *p)

Delete a PDFlib object and free all internal resources.

Details This function deletes a PDF object and frees all document-related PDFlib-internal resources. Although not necessarily required for single-document generation, deleting the PDF object is heavily recommended for all server applications when they are done producing PDF. This function must only be called once for a given PDF object. *PDF_delete( )* should also be called for cleanup when an exception occurred. *PDF_delete( )* itself is guaranteed to not throw any exception. If more than one PDF document will be generated it is not necessary to call *PDF_delete( )* after each document, but only when the complete sequence of PDF documents is done.

Scope *any*; this function starts *null* scope, i.e., no more API function calls are allowed.

Bindings C: If the PDFlib DLL has been loaded dynamically at runtime with *PDF_new_dl( ),* use *PDF_delete_dl( )* to delete the PDFlib object (see Section 2.4.3, »Using PDFlib as a DLL loaded at Runtime«, page 22).

C++: this function is indirectly available via the PDF destructor.
Java: this function is automatically called by the wrapper code. However, it can explicitly be called from client code in order to overcome shortcomings in Java's finalizer system.

7.2.2 Document and Page

Table 7.3 lists relevant parameters and values for this section. Section 7.9.1, »Document Open Action and Open Mode«, page 204 presents additional relevant parameters.

Table 7.3 Parameters and values for the document and page functions

function	*key*	*explanation*
set_parameter	*openwarning*	*Enable or suppress warnings related to opening the PDF output file. Possible values are true and false. Scope: any. Default: false.*
set_value	*pagewidth pageheight*	*Change the page size of the current page. Scope: page, path.*
set_parameter	*topdown*	*If true, the origin of the coordinate system at the beginning of a page, pattern, or template will be assumed in the top left corner of the page, and y coordinates will increase downwards; otherwise the default coordinate system will be used (see Section 3.2.1, »Coordinate Systems«, page 53). Scope: document. Default: false.*
get_value	*pagewidth pageheight*	*Get the page size of the current page. Scope: page, path.*
set_value	*CropBox, BleedBox, ArtBox, TrimBox*	*Change one of the box parameters of the current page. The parameter name must be followed by a slash '/' character and one of llx, lly, urx, ury, for example: CropBox/llx (see Section 3.2.2, »Page Sizes and Coordinate Limits«, page 55 for details). Scope: page.*
set_parameter	*userpassword*	*Set the user password to the supplied string. Scope: object. Default: none.*
set_parameter	*master-password*	*Set the master password to the supplied string. Scope: object. Default: none.*
set_parameter	*permissions*	*Set the document permissions to a combination of the keywords given in Table 3.12. Scope: object. Default: none.*

int PDF_open_file(PDF *p, const char *filename)

Create a new PDF file using the supplied file name.

filename Absolute or relative name of the PDF output file to be generated. Only 8-bit characters are supported in the file name; Unicode or embedded null characters are not supported.

If *filename* is empty, the PDF document will be generated in memory instead of on file, and the generated PDF data must be fetched by the client with the *PDF_get_buffer( )* function. *PDF_open_file( )* will always succeed in this case, and never return the -1 (in PHP: 0) error value.

The special file name »–« can be used for generating PDF on the *stdout* channel (this obviously does not apply to environments which don't support the notion of a *stdout* channel, such as Mac OS 9). On Windows it is OK to use UNC paths or mapped network drives.

Returns -1 (in PHP: 0) on error, and 1 otherwise. If the *openwarning* parameter is set to true this function will throw an exception if the PDF output file cannot be opened for writing.

Details This function creates a new PDF file using the supplied *filename*. PDFlib will attempt to open a file with the given name, and close the file when the PDF document is finished.

Scope *object;* this function starts document scope if the file could successfully be opened, and must always be paired with a matching *PDF_close( )* call.

Params *openwarning*

Bindings C, C++, Java, JScript: take care of properly escaping the backslash path separator. For example, the following denotes a file on a network drive: *malik**rp**foo.pdf*.

void PDF_open_mem(PDF *p, size_t (*writeproc)(PDF *p, void *data, size_t size))

Open a new PDF in memory, and install a callback for fetching the data.

writeproc Callback function which will be called by PDFlib in order to submit (portions of) the generated PDF data.

Details This function opens a new PDF document in memory, without writing to a disk file. The callback function must return the number of bytes written. If the return value doesn't match the *size* argument supplied by PDFlib, an exception will be thrown, and PDF generation stops. The frequency of *writeproc* calls is configurable with the *flush* parameter. The default value of the *flush* parameter is *page* (see Section 3.1.7, »Generating PDF Documents in Memory«, page 50 for details).

Scope *object;* this function starts document scope, and must always be paired with a matching *PDF_close( )* call.

Bindings This function is only available in the C and C++ bindings.
C++: *writeproc* must be a C-style function, not a C++ method.
Other bindings: use *PDF_open_file( )* with an empty file name in order to create PDF documents in memory.

const char * PDF_get_buffer(PDF *p, long *size)

Get the contents of the PDF output buffer. The result must be used by the client before calling any other PDFlib function.

size C-style Pointer to a memory location where the length of the returned data in bytes will be stored.

Returns A buffer full of binary PDF data for consumption by the client. It returns a language-specific data type for binary data according to Table 7.1. The returned buffer can be used until the end of the surrounding *object* scope or until the next call to *PDF_get_buffer( )*, depending on what happens first.

Details Fetch the full or partial buffer containing the generated PDF data. If this function is called between page descriptions, it will return the PDF data generated so far. If it is called after *PDF_close( )* it returns the remainder of the PDF document. If there is only a single call to this function which happens after *PDF_close( )* the returned buffer is guaranteed to contain the complete PDF document in a contiguous buffer.

Since PDF output contains binary characters, client software must be prepared to accept non-printable characters including null values.

Scope *object, document* (in other words: after *PDF_end_page( )* and before *PDF_begin_page( ),* or after *PDF_close( )* and before *PDF_delete( )*. This function can only be used if an empty filename has been supplied to *PDF_open_file( )*.

Bindings C and C++: the *size* parameter is only used for C and C++ clients.

Other bindings: an object of appropriate length will be returned, and the *size* parameter must be omitted.

void PDF_close(PDF *p)

Close the generated PDF file, and release all document-related resources.

Details This function finishes the generated PDF document, free all document-related resources, and close the output file if the PDF document has been opened with *PDF_open_file( )*. This function must be called when the client is done generating pages, regardless of the method used to open the PDF document.

When the document was generated in memory (as opposed to on file), the document buffer will still be kept after this function is called (so that it can be fetched with *PDF_get_buffer( )*), and will be freed in the next call to *PDF_open( )*, or when the PDFlib object goes out of scope in *PDF_delete( )*.

Scope *document;* this function terminates document scope, and must always be paired with a matching call to one of the *PDF_open_*( )* functions.

void PDF_begin_page(PDF *p, float width, float height)

Add a new page to the document.

width, height The *width* and *height* parameters are the dimensions of the new page in points. Acrobat's page size limits are documented in Section 3.2.1, »Coordinate Systems«, page 53. A list of commonly used page formats can be found in Table 3.5. The page size can be changed after calling *PDF_begin_page( )* with the *pagewidth* and *pageheight* parameters. In order to produce landscape pages use *width* > *height*. PDFlib uses *width* and *height* to construct the page's MediaBox. You can use several parameters to set other box entries in the PDF (see Table 7.2).

Scope *document;* this function starts page scope, and must always be paired with a matching *PDF_end_page( )* call.

Params *pagewidth, pageheight, CropBox, BleedBox, ArtBox, TrimBox*

void PDF_end_page(PDF *p)

Finish the page.

Details This function must be used to finish a page description.

Scope *page;* this function terminates page scope, and must always be paired with a matching *PDF_begin_page( )* call.

7.2.3 Parameter Handling

PDFlib maintains a number of internal parameters which are used for controlling PDFlib's operation and the appearance of the PDF output. Four functions are available for setting and retrieving both numerical and string parameters. All parameters (both keys and values) are case-sensitive. The descriptions of available parameters can be found in the respective sections in this chapter.

float PDF_get_value(PDF *p, const char *key, float modifier)

Get the value of some PDFlib parameter with numerical type.

key The name of the parameter to be queried.

modifier An optional modifier to be applied to the parameter. Whether a modifier is required and what it relates to is explained in the various parameter tables. If the modifier is unused it must be 0.

Returns The numerical value of the parameter.

Scope Depends on *key*.

See also *PDF_get_pdi_value( )*

void PDF_set_value(PDF *p, const char *key, float value)

Set the value of some PDFlib parameter with numerical type.

key The name of the parameter to be set.

value The new value of the parameter to be set.

Scope Depends on *key*.

const char * PDF_get_parameter(PDF *p, const char *key, float modifier)

Get the contents of some PDFlib parameter with string type.

key The name of the parameter to be queried.

modifier An optional modifier to be applied to the parameter. Whether a modifier is required and what it relates to is explained in the various parameter tables. If the modifier is unused it must be 0.

Returns The string value of the parameter. The returned string can be used until the end of the surrounding *document* scope. If no information is available an empty string will be returned.

Scope Depends on *key*.

Bindings C and C++: C and C++ clients must not free the returned string. PDFlib manages all string resources internally.

See also *PDF_get_pdi_parameter( )*

void PDF_set_parameter(PDF *p, const char *key, const char *value)

Set some PDFlib parameter with string type.

key The name of the parameter to be set.

value The new value of the parameter to be set.

Scope Depends on *key*.

7.2.4 PDFlib Virtual File System (PVF) Functions

void PDF_create_pvf(PDF *p,
const char *filename, int reserved, const void *data, size_t size, const char *optlist)

Create a named virtual read-only file from data provided in memory.

filename The name of the virtual file. This is an arbitrary string which can later be used to refer to the virtual file in other PDFlib calls.

reserved (C language binding only.) Reserved, must be 0.

data A pointer to a memory buffer containing the data comprising the virtual file.

size (C language binding only) The length in bytes of the memory block containing the data.

optlist An option list (see Section 3.1.4, »Option Lists«, page 44) according to Table 7.4.

Details The virtual file name can be supplied to any API function which use files. Some of these functions may set a lock on the virtual file until the data is no longer needed. Virtual files will be kept in memory until they are deleted explicitly with *PDF_delete_pvf( )*, or automatically in *PDF_delete( )*.

If *filename* refers to an existing virtual file an exception will be thrown. This function does not check whether *filename* is already in use for a regular disk file.

Unless the *copy* option has been supplied, the caller must not modify or free (delete) the supplied data before a corresponding successful call to *PDF_delete_pvf( )*. Not obeying to this rule will most likely result in a crash.

Scope *any*

Table 7.4 Options for PDF_create_pvf( ).

option	*type*	*description*
copy	*boolean*	*PDFlib will immediately create an internal copy of the supplied data. In this case the caller may dispose of the supplied data immediately after this call. The copy option will automatically be set to true in the COM, .NET, and Java bindings (default for other bindings: false). In other language bindings the data will not be copied unless the copy option is supplied.*

int PDF_delete_pvf(PDF *p, const char *filename, int reserved)

Delete a named virtual file and free the associated data structures (but not the actual contents).

filename The name of the virtual file as supplied to *PDF_create_pvf( )*.

reserved (C language binding only.) Reserved, must be 0.

Returns -1 (in PHP: 0) if the corresponding virtual file exists but is locked, and 1 otherwise.

Details If the file isn't locked, PDFlib will immediately delete the data structures associated with *filename*. If *filename* does not refer to a valid virtual file this function will silently do nothing. After successfully calling this function *filename* may be reused. All virtual files will automatically be deleted in *PDF_delete( )*.

The detailed semantics depend on whether or not the *copy* option has been supplied to the corresponding call to *PDF_create_pvf( ):* If the *copy* option has been supplied, both the administrative data structures for the file and the actual file contents (data) will be freed; otherwise, the contents will not be freed, since the client is supposed to do so.

Scope *any*

7.2.5 Exception Handling

Table 7.5 lists relevant parameters and values for this section.

Table 7.5 Parameters and values for exception handling

function	*key*	*explanation*
set_parameter	*warning*	*Enable or suppress warnings (nonfatal exceptions). Possible values are true and false. Scope: any. Default: true*

int PDF_get_errnum(PDF *p)

Get the number of the last thrown exception, or the reason of a failed function call.

Returns The number of an exception, or the reason code of the the most recently called function which failed with an error code.

Scope Between an exception thrown by PDFlib and *PDF_delete( ).* Alternatively, this function may be called after a function returned a -1 (in PHP: 0) error code, but before calling any other function except those listed in Section 7.2.5, »Exception Handling«, page 156.

Bindings In the Java and C++ bindings this method is also available as *get_errnum( )* in the *PDFlibException* object.

const char *PDF_get_errmsg(PDF *p)

Get the descriptive text of the last thrown exception, or the reason of a failed function call.

Returns Text containing the description of the last exception thrown, or the reason why the most recently called function failed with an error code.

Scope Between an exception thrown by PDFlib and *PDF_delete( ).* Alternatively, this function may be called after a function returned a -1 (in PHP: 0) error code, but before calling any other function except those listed in Section 7.2.5, »Exception Handling«, page 156.

Bindings In the Java and C++ bindings this method is also available as *getMessage( )* in the *PDFlibException* object.

const char *PDF_get_apiname(PDF *p)

Get the name of the API function which threw the last exception or failed.

Returns The name of the function which threw an exception, or the name of the most recently called function which failed with an error code.

Scope Between an exception thrown by PDFlib and *PDF_delete( )*. Alternatively, this function may be called after a function returned a -1 (in PHP: 0) error code, but before calling any other function except those listed in Section 7.2.5, »Exception Handling«, page 156.

Bindings In the Java and C++ bindings this method is also available as *get_apiname( )* in the *PDFlibException* object.

void *PDF_get_opaque(PDF *p)

Fetch the opaque application pointer stored in PDFlib.

Details This function returns the opaque application pointer stored in PDFlib which has been supplied in the call to *PDF_new2( )*. PDFlib never touches the opaque pointer, but supplies it unchanged to the client. This may be used in multi-threaded applications for storing private thread-specific data within the PDFlib object. It is especially useful for thread-specific exception handling.

Scope *any*

Bindings Only available in the C and C++ bindings.

7.3 Text Functions

7.3.1 Font Handling

Table 7.6 lists relevant parameters and values for this section.

Table 7.6 Parameters and values for the font functions (see Section 7.2.3, »Parameter Handling«, page 153)

function	key	explanation
set_parameter	FontAFM FontPFM FontOutline Encoding HostFont	The corresponding resource file line as it would appear for the respective category in a UPR file (see Section 3.1.6, »Resource Configuration and File Searching«, page 47). Multiple calls add new entries to the internal list. (See also prefix and resourcefile in Table 7.2). Scope: any.
get_value	font	Return the identifier of the current font which must have been set with PDF_setfont( ). Scope: page, pattern, template, glyph.
get_value	fontmaxcode	Return the number of valid glyph ids for the font identified by the modifier. Scope: any.
get_parameter	fontname	The name of the current font which must have been previously set with PDF_setfont( ). Scope: page, pattern, template, glyph.
get_parameter	fontencoding	The name of the encoding or CMap used with the current font. A font must have been previously set with PDF_setfont( ). Scope: page, pattern, template, glyph.
get_value	fontsize	Return the size of the current font which must have been previously set with PDF_setfont( ). Scope: page, pattern, template, glyph.
get_parameter	fontstyle	The style of the current font, which resembles the fontstyle option (normal, bold, italic, or bolditalic). Scope: page, pattern, template, glyph.
get_value	capheight ascender descender	Return metrics information for the font identified by the modifier. See Section 4.6, »Text Metrics, Text Variations, and Box Formatting«, page 96 for more details. The values are measured in fractions of the font size, and must therefore be multiplied by the desired font size. Scope: any.
set_parameter	fontwarning	If set to false, PDF_load_font( ) returns -1 (in PHP: 0) if the font/encoding combination cannot be loaded (instead of throwing an exception). Default: true. Scope: any.
get_value	monospace	Returns the value of the monospace option for the current font if it has been set, and 0 otherwise. Scope: page, pattern, template, glyph.
set_value	subsetlimit	Disables font subsetting if the document uses more than the given percentage of characters in the font. Default value: 100 percent. Scope: any.
set_value	subsetminsize	Subsetting will only be applied to fonts above this size in Kilobyte (see Section 4.3, »Font Embedding and Subsetting«, page 80). Default: 100 KB. Scope: any.
set_parameter	autosubsetting	Controls the automatic activation of font subsetting for TrueType and OpenType fonts (see Section 4.3, »Font Embedding and Subsetting«, page 80). Default: true. Scope: any.
set_parameter	autocidfont	Controls the automatic conversion of TrueType fonts with encodings other than macroman and winansi to CID fonts (see Section 4.3, »Font Embedding and Subsetting«, page 80). Default: true. Scope: any.
set_parameter	unicodemap	Controls the generation of ToUnicode CMaps (see Section 4.5.1, »Unicode for Page Descriptions«, page 91). Default: true. Scope: any.

int PDF_load_font(PDF *p,
const char *fontname, int len, const char *encoding, const char *optlist)

Search for a font, and prepare it for later use.

fontname The real or alias name of the font. It will be used to find font data according to the description in Section 4.3.1, »How PDFlib Searches for Fonts«, page 80. The font name can be encoded in ASCII, UTF-16, or UTF-8 with initial BOM. The latter two are useful for localized host font names. Case is significant. In C *fontname* must not contain null characters unless the *len* parameter is different from 0.

len (C language binding only) Length of *fontname* in bytes for strings which may contain null characters. If *len = 0* a null-terminated string must be provided.

encoding The encoding to be used with the font, which must be one of the following:

- one of the predefined 8-bit encodings *winansi, macroman, macroman_euro, ebcdic, pdfdoc, iso8859-X, cpXXXX*, or *U+XXXX*;
- *host* or *auto* for an automatically selected encoding;
- the name of a user-defined encoding loaded from file or defined via *PDF_encoding_set_char( );*
- *unicode* for Unicode-based addressing;
- *glyphid for* glyph id addressing;
- *builtin* to select the font's internal encoding;
- the name of a standard CMap. In this case *fontname* must be the name of a standard CJK font; custom CJK fonts are only supported with *unicode* encoding (see Section 4.7, »Chinese, Japanese, and Korean Text«, page 101);
- an encoding name known to the operating system (not available on all platforms).

Care must be taken to choose an encoding which is compatible with the font, and matches the available input and desired output. Review Section 4.4, »Encoding Details«, page 85, for more details. Case is significant for *encoding*.

optlist An option list (see Section 3.1.4, »Option Lists«, page 44) according to Table 7.7.

Returns A font handle for later use with *PDF_setfont( )*. The behavior of this function changes when the *fontwarning* parameter or option is set to *false*. In this case the function returns an error code of -1 (in PHP: 0) if the requested font/encoding combination cannot be loaded, and does not throw an exception. However, exceptions will still be thrown when bad parameters are supplied.

The returned number – the font handle – doesn't have any significance to the user other than serving as an argument to *PDF_setfont( )* and related functions. In particular, requesting the same font/encoding combination in different documents may result in different font handles.

When calling this function again with the same font name the same font handle as in the first call will be returned unless a different *encoding* parameter or *fontstyle* option has been supplied.

Details This function prepares a font for later use with *PDF_setfont( )*. The metrics will be loaded from memory or from a (virtual or disk-based) metrics file. If the requested font/encoding combination cannot be used due to configuration problem (e.g., a font, metrics, or encoding file could not be found, or a mismatch was detected), an exception will be

thrown unless the *fontwarning* parameter is set to false. Otherwise, the value returned by this function can be used as font argument to other font-related functions.

Scope *document, page, pattern, template, glyph*

Params See Table 7.6.

PDF/X If PDF/X-1, PDF/X-1a, or PDF/X-3 output is generated the *embedding* option must be true.

Table 7.7 Options for PDF_load_font( ). If the same font is loaded multiply with different options, only the first set of options will be processed. Conflicting options (except fontwarning) in subsequent calls will be ignored.

option	type	description
auto-subsetting	*boolean*	*Dynamically decide whether or not the font will be subset, subject to the subsetlimit and subsetminsize parameters and the actual usage of glyphs. This option will be ignored when the subsetting option has been supplied. Default: the value of the global autosubsetting parameter.*
autocidfont	*boolean*	*If true, TrueType fonts with 8-bit encoding except winansi, macroman, builtin and OpenType fonts without glyph names will automatically be stored as CID fonts. This avoids problems with certain non-accessible glyphs outside winansi encoding. Default: the value of the global autocidfont parameter.*
embedding	*boolean*	*Controls whether or not the font will be embedded. This does not have any effect on Type 3 fonts. If a font is to be embedded, the font outline file must be available in addition to the metrics information (this is irrelevant for TrueType and OpenType fonts), and the actual font outline definition will be included in the PDF output. If a font is not embedded, only general information about the font is included in the PDF output. Default: false.*
fontstyle	*keyword*	*Controls the creation of artificial font styles. These work only for TrueType and OpenType fonts which are not embedded (see Section 4.6.3, »Text Variations«, page 98). Possible keywords are normal, bold, italic, bolditalic. Default: normal.*
fontwarning	*boolean*	*If true, an exception will be thrown when the requested font/encoding combination cannot be loaded; If false an error code will be returned. (The encoding search is under control of the fontwarning parameter, but not under control of the fontwarning option.) Default: the value of the global fontwarning parameter.*
kerning	*boolean*	*Controls whether or not kerning values will be read from the font (see Section 4.6, »Text Metrics, Text Variations, and Box Formatting«, page 96). Default: false.*
monospace	*integer 1...2048*	*Forces all glyphs in the font to use the specified width (in the font coordinate system: 1000 units equal the font size). For Type 3 fonts all glyph widths which are different from 0 will be modified. This option is only recommended for standard CJK fonts, and not supported for core fonts; it will be ignored if the font is embedded. Default: absent (metrics from the font will be used)*
subsetlimit	*float*	*Font subsetting will be disabled if the percentage of glyphs used in the document related to the total number of glyphs in the font exceeds the provided percentage. Default: the value of the global subsetlimit parameter.*
subsetminsize	*float*	*Font subsetting will be disabled if the size of the original font file is less than the provided value in KB. Default: the value of the global subsetminsize parameter.*
subsetting	*boolean*	*Controls whether or not the font will be subset, subject to the total number of glyphs used in the document and the values of the subsetlimit and subsetminsize options (see Section 4.3, »Font Embedding and Subsetting«, page 80). Default: false.*
unicodemap	*boolean*	*Controls the generation of ToUnicode CMaps (see Section 4.5.1, »Unicode for Page Descriptions«, page 91). Default: true.*

int PDF_findfont(PDF *p, const char *fontname, const char *encoding, int embed)

Search for a font, and prepare it for later use. *PDF_load_font( )* is recommended.

fontname See *PDF_load_font( )*.

encoding See *PDF_load_font( )*.

embed See *embedding* option of *PDF_load_font( ):* 1 is equivalent to *embedding = true*.

Returns See *PDF_load_font( )*.

Details See *PDF_load_font( )*.

Scope *document, page, pattern, template, glyph*

Params See Table 7.6.

PDF/X If PDF/X-1, PDF/X-1a, or PDF/X-3 output is generated, *embed* must be 1.

void PDF_setfont(PDF *p, int font, float fontsize)

Set the current font in the given size.

font A font handle returned by *PDF_findfont( ) or PDF_load_font( )*.

fontsize Size of the font, measured in units of the current user coordinate system. The font size must not be 0; negative font size will result in mirrored text relative to the current transformation matrix.

Details The font must be set on each page before drawing any text. Font settings will not be retained across pages. The current font can be changed an arbitrary number of times per page.

Scope *page, pattern, template, glyph*

Params See Table 7.6. This function automatically sets the *leading* parameter to *fontsize*.

7.3.2 User-defined (Type 3) Fonts

void PDF_begin_font(PDF *p, char *fontname, int reserved,
float a, float b, float c, float d, float e, float f, const char *optlist)

Start a type 3 font definition.

fontname The name under which the font will be registered, and can later be used with *PDF_load_font( )*.

reserved (C language binding only.) Reserved, must be 0.

a, b, c, d, e, f The elements of the font matrix. This matrix defines the coordinate system in which the glyphs will be drawn. The six floating point values make up a matrix in the same way as in PostScript and PDF (see references). In order to avoid degenerate transformations, *a*d* must not be equal to *b*c*.

A typical font matrix for a 1000 *x* 1000 coordinate system is *[0.001, 0, 0, 0.001, 0, 0]*.

optlist An option list (see Section 3.1.4, »Option Lists«, page 44) according to Table 7.8.

Details The font may contain an arbitrary number of glyphs, but only 256 glyphs can be accessed via an encoding. The defined font can be used until the end of the current *document* scope.

Scope *document;* this function starts *font* scope, and must always be paired with a matching *PDF_end_font( )* call.

Table 7.8 Options for PDF_begin_font( )

option	*type*	*description*
colorized	*boolean*	*If true, the font may explicitly specify the color of individual characters. If false, all characters will be drawn with the current color, and the glyph definitions must not contain any color operators or images other than masks. Default: false.*

void PDF_end_font(PDF *p)

Terminate a type 3 font definition.

Scope *font;* this function terminates *font* scope, and must always be paired with a matching *PDF_begin_font( )* call.

void PDF_begin_glyph(PDF *p,
char *glyphname, float wx, float llx, float lly, float urx, float ury)

Start a type 3 glyph definition.

glyphname The name of the glyph. This name must be used in any encoding which will be used with the font. Glyph names within a font must be unique.

wx The width of the glyph in the glyph coordinate system, as specified by the font's matrix.

llx, lly, urx, ury If the font's *colorized* option is false (which is default), the coordinates of the lower left and upper right corners of the glyph's bounding box. The bounding box values must be correct in order to avoid problems with PostScript printing. If the font's *colorized* option is *true*, all four values must be 0.

Details The glyphs in a font can be defined using text, graphics, and image functions. Images, however, can only be used if the font's *colorized* option is true, or the image has been opened with the *mask* option. It is strongly suggested to use the inline image feature (see Section 5.1.1, »Basic Image Handling«, page 111) for defining bitmaps in Type 3 fonts.

Since the complete graphics state will be inherited when the character is drawn, the glyph definition should explicitly set any aspect of the graphics state which is relevant for the glyph definition (e.g., linewidth).

Scope *font;* this function starts *glyph* scope, and must always be paired with a matching *PDF_end_glyph( )* call.

void PDF_end_glyph(PDF *p)

Terminate a type 3 glyph definition.

Scope *glyph;* this function terminates *glyph* scope, and must always be paired with a matching *PDF_begin_glyph( )* call.

7.3.3 Encoding Definition

void PDF_encoding_set_char(PDF *p,
const char *encoding, int slot, const char *glyphname, int uv)

Add a glyph name to a custom encoding.

encoding The name of the encoding. This is the name which must be used with *PDF_load_font( )*. The encoding name must be different from any built-in encoding and all previously used encodings.

slot The position of the character in the encoding to be defined, with *0 <= slot <= 255*. A particular slot must only be filled once within a given encoding.

glyphname The character's name.

uv The character's Unicode value.

Details This function can be called multiply to define up to 256 character slots in an encoding. More characters may be added to a particular encoding until it has been used for the first time; otherwise an exception will be raised. Not all code points must be specified; undefined slots will be filled with *.notdef*.

There are three possible combinations of glyph name and Unicode value:

- *glyphname* supplied, *uv = 0:* this parallels an encoding file without Unicode values;
- *uv* supplied, but no *glyphname* supplied*:* this parallels a codepage file;
- *glyphname* and *uv* supplied: this parallels an encoding file with Unicode values;

The defined encoding can be used until the end of the current *object* scope.

Scope *object, document, page, pattern, template, path, font, glyph*

7.3.4 Text Output

Note *All text supplied to the functions in this section must match the encoding selected with PDF_load_font( ). This applies to 8-bit text as well as Unicode or other encodings selected via a CMap. Due to restrictions in Acrobat, text strings must not exceed 32 KB in length.*

Table 7.9 lists relevant parameters and values for this section.

void PDF_set_text_pos(PDF *p, float x, float y)

Set the current text position.

x, y The current text position to be set.

Table 7.9 Parameters and values for the text functions (see Section 7.2.3, »Parameter Handling«, page 153)

function	key	explanation
set_parameter get_parameter	textformat	Specifies the format in which the text output functions will expect the client-supplied strings. Possible values are bytes, utf8, utf16, utf16le, utf16be, and auto (see Section 4.5.2, »Unicode Text Formats«, page 92). Default: auto. Scope: any.
set_value get_value	leading	Set or get the leading, which is the distance between baselines of adjacent lines of text. The leading is used for PDF_continue_text( ) and PDF_show_boxed( ). It is set to the value of the font size when a new font is selected using PDF_setfont( ). Setting the leading equal to the font size results in dense line spacing (leading = 0 will result in overprinting lines). However, ascenders and descenders of adjacent lines will generally not overlap. Scope: page, pattern, template, glyph.
set_value get_value	textrise	Set or get the text rise parameter. The text rise specifies the distance between the desired text position and the default baseline. Positive values of text rise move the text up. The text rise always relates to the vertical coordinate. This may be useful for superscripts and subscripts. The text rise is set to the default value of 0 at the beginning of each page. Scope: page, pattern, template, glyph.
set_value get_value	textrendering	Set or get the current text rendering mode. It can have one of the values given in Table 4.4. The text rendering parameter is set to the default of 0 (= solid fill) at the beginning of each page. Scope: page, pattern, template, glyph.
set_parameter	glyphwarning	If true, an exception will be thrown when a glyph cannot be shown because the font does not contain the corresponding glyph description. If false, glyphs missing from a font will be replaced with a space character or glyph ID 0. Default: false. Scope: any.
set_value get_value	horizscaling	Set or get the horizontal text scaling to the given percentage. Text scaling shrinks or expands the text by a given percentage. The text scaling is set to the default of 100 at the beginning and end of each page. Text scaling always relates to the horizontal coordinate. Scope: page, pattern, template, glyph, document.
set_value get_value	charspacing	Set or get the character spacing, i.e., the shift of the current point after placing the individual characters in a string. The spacing is given in units of the current user coordinate system. It is reset to the default of 0 at the beginning and end of each page. In order to spread the characters apart use positive values for horizontal writing mode, and negative values for vertical writing mode. Scope: page, pattern, template, glyph, document.
set_value get_value	wordspacing	Set or get the word spacing, i.e., the shift of the current point after placing individual words in a text line. In other words, the current point is moved horizontally after each ASCII space character (0x20). Since only 8-bit encodings can include an ASCII space character, word spacing does not work with wide-character addressing, such as unicode, glyphid, or encodings which contain non-AGL glyph names. Also, it does not work when font subsetting is active. The spacing value is given in text space units. It is reset to the default of 0 at the beginning and end of each page. Scope: page, pattern, template, glyph, document.
get_value	textx texty	Get the x or y coordinate, respectively, of the current text position. Scope: page, pattern, template, glyph.
set_parameter get_parameter	underline overline strikeout	Set or get the current underline, overline, and strikeout modes, which are retained until they are explicitly changed, or a new page is started. Theses modes can be set independently from each other, and are reset to false at the beginning of each page (see Section 4.6, »Text Metrics, Text Variations, and Box Formatting«, page 96). Scope: page, pattern, template, glyph. true underline/overline/strikeout text false do not underline/overline/strikeout text
set_parameter	kerning	If true, enable kerning for fonts which have been opened with the kerning option; disable if false. Default value is true (see Section 4.6, »Text Metrics, Text Variations, and Box Formatting«, page 96). Scope: any.

Details The text position is set to the default value of *(0, 0)* at the beginning of each page. The current point for graphics output and the current text position are maintained separately.

Scope *page, pattern, template, glyph*

Params See Table 7.9.

---

void PDF_show(PDF *p, const char *text)
void PDF_show2(PDF *p, const char *text, int len)

Print text in the current font and size at the current text position.

text The text to be printed. In C *text* must not contain null characters when using *PDF_show( )*, since it is assumed to be null-terminated; use *PDF_show2( )* for strings which may contain null characters.

len (Only for *PDF_show2( ).)* Length of *text* (in bytes) for strings which may contain null characters. If *len = 0* a null-terminated string must be provided.

Details The font must have been set before with *PDF_setfont( )*. The current text position is moved to the end of the printed text.

Scope *page, pattern, template, glyph*

Params See Table 7.9.

Bindings *PDF_show2( )* is only available in C since in all other bindings arbitrary string contents can be supplied with *PDF_show( )*.

---

void PDF_show_xy(PDF *p, const char *text, float x, float y)
void PDF_show_xy2(PDF *p, const char *text, int len, float x, float y)

Print *text* in the current font at position *(x, y)*.

text The text to be printed. In C *text* must not contain null characters when using *PDF_show_xy( )*, since it is assumed to be null-terminated; use *PDF_show_xy2( )* for strings which may contain null characters.

x,y The position in the user coordinate system where the text will be printed.

len (Only for *PDF_show_xy2( ).)* Length of *text* (in bytes) for strings which may contain null characters. If *len = 0* a null-terminated string must be provided.

Details The font must have been set before with *PDF_setfont( )*. The current text position is moved to the end of the printed text.

Scope *page, pattern, template, glyph*

Params See Table 7.9.

Bindings *PDF_show_xy2( )* is only available in C since in all other bindings arbitrary string contents can be supplied with *PDF_show_xy( )*.

void PDF_continue_text(PDF *p, const char *text)
void PDF_continue_text2(PDF *p, const char *text, int len)

Print text at the next line.

text The text to be printed. If this is an empty string, the text position will be moved to the next line anyway. In C *text* must not contain null characters when using *PDF_continue_text( )*, since it is assumed to be null-terminated; use *PDF_continue_text2( )* for strings which may contain null characters.

len (Only for *PDF_continue_text2( ).)* Length of *text* (in bytes) for strings which may contain null characters. If *len = 0* a null-terminated string must be provided as in *PDF_continue_text( )*.

Details The positioning of text *(x* and *y* position) and the spacing between lines is determined by the *leading* parameter and the most recent call to *PDF_show_xy( )* or *PDF_set_text_pos( )*. This function can also be used after *PDF_show_boxed( )* if that function has been called with *mode = left* or *justify*. The current point will be moved to the end of the printed text; the *x* position for subsequent calls of this function will not be changed.

Scope *page, pattern, template, glyph;* this function should not be used in vertical writing mode.

Params See Table 7.9.

Bindings *PDF_continue_text2( )* is only available in C since in all other bindings arbitrary string contents can be supplied with *PDF_continue_text( )*.

void PDF_fit_textline(PDF *p, const char *text, int len, float x, float y, const char *optlist)

Place a single text line at *(x, y)* with various options.

text The text to be printed.

len (C binding only) Length of *text* (in bytes) for strings which may contain null characters. If *len = 0* a null-terminated string must be provided.

x, y The coordinates of the reference point in the user coordinate system where the text will be placed, subject to various options.

optlist An option list (see Section 3.1.4, »Option Lists«, page 44) specifying placement details according to Table 7.10.

Details The current graphics state will not be modified by this function. In particular, the current text position and the current font will be unaffected.

Scope *page, pattern, template, glyph;* this function should not be used in vertical writing mode.

Params See Table 7.9.

Table 7.10 Options for PDF_fit_textline( )

key	type	explanation
boxsize	float list	Two values specifying the width and height of a box, relative to which the text will be placed and possibly scaled. The lower left corner of the box coincides with the reference point (x, y). Placing the text and fitting it into the box is controlled by the position and fitmethod options. If width = 0, only the height is considered; If height = 0, only the width is considered. In these cases the text will be placed relative to the vertical line from (x, y) to (x, y+height), or the horizontal line from (x, y) to (x+width, y), respectively. Default: {0 0}.
charspacing	float	The character spacing (see Table 7.9). Default: the current value of the global charspacing parameter.
fitmethod	keyword	Specifies the method used to fit the text into the specified box. This option will be ignored if no box has been specified. Default: nofit. nofit — Position the text only, without any scaling or clipping. clip — Position the text, and clip it at the edges of the box. meet — Position the text according to the position option, and scale it such that it entirely fits into the box while preserving its aspect ratio. Generally at least two edges of the text will meet the corresponding edges of the box. The dpi and scale options are ignored. auto — This method tries to fit the text into the box automatically. In detail: Same as nofit if the text fits into the box. Otherwise a scaling factor is calculated such that the text fits into the box. If this factor is larger than 0.75 the text is distorted to fit intothe box, otherwise the meet method is applied. slice — Position the text according to the position option, and scale it such that it entirely covers the box, while preserving the aspect ratio and making sure that at least one dimension of the text is fully contained in the box. Generally parts of the text's other dimension will extend beyond the box, and will therefore be clipped. entire — Position the text according to the position option, and scale it such that it entirely covers the box. Generally this method will distort the text. The scale option will be ignored.
font	font handle	A font handle returned by PDF_load_font( ). Default: the current font.
fontsize	float	(Required if the font option is provided) Size of the font, measured in units of the current user coordinate system. Default: the current font size.
horizscaling	float	The horizontal text scaling (see Table 7.9). Default: the current value of the global horizscaling parameter.
kerning	boolean	Kerning behavior (see Table 7.9). Default: the current value of the global kerning parameter.
margin	float list	One or two float values describing additional horizontal and vertical extensions of the text box. Default: 0.
orientate	keyword	Specifies the desired orientation of the text when it is placed. Default: north. north — upright east — pointing to the right south — upside down west — pointing to the left
overline	boolean	Overline mode (see Table 7.9). Default: the current value of the global overline parameter.

Table 7.10 Options for PDF_fit_textline( )

key	type	explanation
position	float list	(Alignment control) One or two values specifying the position of the reference point (x, y) within the text's bounding box with {0 0} being the lower left corner of the text box, and {100 100} the upper right corner. If the boxsize option has been specified, the position option also specifies the positioning of the target box. The values are expressed as percentages of the text's width and height. If both percentages are equal it is sufficient to specify a single float value. Some examples: {0 50} results in left-justified text; {50 50} results in centered text; {100 50} results in right-justified text. Default: 0 (lower left corner)
rotate	float	Rotate the coordinate system, using the reference point as center and the specified value as rotation angle in degrees. This results in the box and the text being rotated. The rotation will be reset when the text has been placed. Default: 0.
strikeout	boolean	Strikeout mode (see Table 7.9). Default: the current value of the global strikeout parameter.
textformat	keyword	The format used to interpret the supplied text (see Section 4.5.2, »Unicode Text Formats«, page 92). Default: the current value of the global textformat parameter.
textrendering	int	The text rendering mode (see Table 4.4). Default: the current value of the global textrendering parameter.
textrise	float	The text rise mode (see Table 7.9). Default: the current value of the global text rise parameter.
underline	boolean	Underline mode (see Table 7.9). Default: the current value of the global underline parameter.
wordspacing	float	The word spacing (see Table 7.9). Default: the current value of the global wordspacing parameter.

int PDF_show_boxed(PDF *p, const char *text, float x, float y, float width, float height, const char *mode, const char *feature)

Format text into a text box according to the requested formatting mode.

text The text to be formatted into the box. The text must not contain any null characters.

x, y The coordinates of the lower left corner of the text box or the coordinates of the alignment point if *width* = *0* and *height* = *0*.

width, height The size of the text box, or 0 for single-line formatting.

mode *mode* selects the horizontal alignment mode. If *width* = *0* and *height* = *0*, *mode* can attain one of the values *left*, *right*, or *center*, and the text will be formatted according to the chosen alignment with respect to the point *(x, y)*, with *y* denoting the position of the baseline. In this mode, this function does not check whether the submitted parameters result in some text being clipped at the page edges, nor does it apply any line-wrapping. It returns the value 0 in this case.

If *width* or *height* is different from *0*, *mode* can attain one of the values *left*, *right*, *center*, *justify*, or *fulljustify*. The supplied text will be formatted into a text box defined by the lower left corner *(x, y)* and the supplied *width* and *height*. If the text doesn't fit into a line, a simple line-breaking algorithm is used to break the text into the next available line, using existing space characters for possible line-breaks. While the *left*, *right*, and *center* modes align the text on the respective line, *justify* aligns the text on both left and

right margins. According to common practice the very last line in the box will only be left-aligned in *justify* mode, while in *fulljustify* mode all lines (including the last one if it contains at least one space character) will be left- and right-aligned. *fulljustify* is useful if the text is to be continued in another column.

feature If the *feature* parameter is *blind*, all calculations are performed (with the exception of the internal *textx* and *texty* coordinates, which are not updated), but no text output is actually generated. This can be used for size calculations and possibly trying different font sizes for fitting some amount of text into a given box by varying the font size. Otherwise *feature* must be empty.

Returns The number of characters which could not be processed since the text didn't completely fit into the column. If the text did actually fit, it returns 0. Since no formatting is performed if *width* = *0* and *height* = *0*, the function always returns 0 in this case.

Details The current font must have been set before calling this function. The current values of font, font size, kerning, horizontal scaling, and leading are used for the text, but the word spacing is ignored. The current text position is moved to the end of the generated text.

Scope *page, pattern, template, glyph;* this function cannot be used with CJK fonts or *ebcdic* or *unicode* encoding. Calling this function is only allowed if the current text format is *auto* or *bytes*, font subsetting is disabled, and the current encoding has a *space* character at position 0x20.

Params See Table 7.9.

See also Restrictions of this functions are listed in Section 4.6.4, »Box Formatting«, page 99.

---

float PDF_stringwidth(PDF *p, const char *text, int font, float fontsize)
float PDF_stringwidth2(PDF *p, const char *text, int len, int font, float fontsize)

Return the width of *text* in an arbitrary font.

text The text for which the width will be queried. In C *text* must not contain null characters when using *PDF_stringwidth( )*, since it is assumed to be null-terminated; use *PDF_stringwidth2( )* for strings which may contain null characters.

len (Only for *PDF_stringwidth2( ).)* Length of *text* (in bytes) for strings which may contain null characters. If *len* = *0* a null-terminated must be provided.

font A font handle returned by *PDF_load_font( )*. The corresponding font must not be a CJK font with a non-Unicode CMap. If *font* refers to such a font, this function returns 0 regardless of the *text* and *fontsize* parameters (unless the *monospace* option has been supplied when loading the font).

fontsize Size of the font, measured in units of the user coordinate system (see *PDF_setfont( )*).

Returns The width of *text* in an arbitrary font which has been selected with *PDF_load_font( )*, and the supplied *fontsize*. The returned width value may be negative (e.g., when negative horizontal scaling has been set).

Details The width calculation takes the current values of the following text parameters into account: horizontal scaling, kerning, character spacing, and word spacing.

Scope *page, pattern, template, path, glyph, document*

Params See Table 7.9.

Bindings *PDF_stringwidth2( )* is only available in C since in all other bindings arbitrary string contents can be supplied with *PDF_stringwidth( )*.

7.4 Graphics Functions

7.4.1 Graphics State Functions

All graphics state parameters are restored to their default values at the beginning of a page. The default values are documented in the respective function descriptions. Functions related to the text state are listed in Section 7.3, »Text Functions«, page 158.

Note *None of the graphics state functions must be used in path scope (see Section 3.2, »Page Descriptions«, page 53).*

void PDF_setdash(PDF *p, float b, float w)

Set the current dash pattern.

b, w The number of alternating black and white units. *b* and *w* must be non-negative numbers.

Details In order to produce a solid line, set *b* = *w* = *0*. The dash parameter is set to solid at the beginning of each page.

Scope *page, pattern, template, glyph*

void PDF_setdashpattern(PDF *p, const char *optlist)

Set a dash pattern defined by an option list.

optlist An option list (see Section 3.1.4, »Option Lists«, page 44) according to Table 7.11. An empty *optlist* will generate a solid line.

Table 7.11 Options for PDF_setdashpattern( )

option	*type*	*description*
dasharray	*float list*	*An list of 2-8 alternating values for the lengths of dashes and gaps for stroked paths (measured in the user coordinate system) . The array values must be non-negative, and not all zero. The array values will be cyclically reused until the complete path is stroked.*
dashphase	*float*	*Distance into the dash pattern at which to start the dash. Default: 0*

Details The dash parameter is set to a solid line at the beginning of each page.

Scope *page, pattern, template, glyph*

void PDF_setpolydash(PDF *p, float *darray, int length)

Deprecated, use *PDF_setdashpattern( )* instead.

void PDF_setflat(PDF *p, float flatness)

Set the flatness parameter.

flatness A positive number which describes the maximum distance (in device pixels) between the path and an approximation constructed from straight line segments.

Details The flatness parameter is set to the default value of 1 at the beginning of each page.

Scope *page, pattern, template, glyph*

void PDF_setlinejoin(PDF *p, int linejoin)

Set the linejoin parameter.

linejoin Specifies the shape at the corners of paths that are stroked, see Table 7.12.

Details The *linejoin* parameter is set to the default value of 0 at the beginning of each page.

Scope *page, pattern, template, glyph*

Table 7.12 Values of the linejoin parameter

value	*description (from the PDF reference)*	*examples*
0	*Miter joins: the outer edges of the strokes for the two segments are continued until they meet. If the extension projects too far, as determined by the miter limit, a bevel join is used instead.*	
1	*Round joins: a circular arc with a diameter equal to the line width is drawn around the point where the segments meet and filled in, producing a rounded corner.*	
2	*Bevel joins: the two path segments are drawn with butt end caps (see the discussion of linecap parameter), and the resulting notch beyond the ends of the segments is filled in with a triangle.*	

void PDF_setlinecap(PDF *p, int linecap)

Set the linecap parameter.

linecap Controls the shape at the end of a path with respect to stroking, see Table 7.13.

Details The linecap parameter is set to the default value of 0 at the beginning of each page.

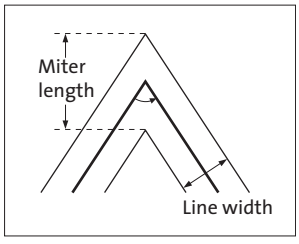
Scope *page, pattern, template, glyph*

Table 7.13 Values of the linecap parameter

value	*description (from the PDF reference)*	*examples*
0	*Butt end caps: the stroke is squared off at the endpoint of the path.*	
1	*Round end caps: a semicircular arc with a diameter equal to the line width is drawn around the endpoint and filled in.*	
2	*Projecting square end caps: the stroke extends beyond the end of the line by a distance which is half the line width and is squared off.*	

void PDF_setmiterlimit(PDF *p, float miter)

Set the miter limit.



miter A value greater than or equal to 1 which controls the spike produced by miter joins.

Details If the linejoin parameter is set to 0 (miter join), two line segments joining at a small angle will result in a sharp spike. This spike will be replaced by a straight end (i.e., the miter join will be changed to a bevel join) when the ratio of the miter length and the line width exceeds the miter limit. The miter limit is set to the default value of 10 at the beginning of each page. This corresponds to an angle of roughly 11.5 degrees.

Scope *page, pattern, template, glyph*

void PDF_setlinewidth(PDF *p, float width)

Set the current line width.

width The line width in units of the current user coordinate system.

Details The *width* parameter is set to the default value of 1 at the beginning of each page.

Scope *page, pattern, template, glyph*

void PDF_initgraphics(PDF *p)

Reset all color and graphics state parameters to their defaults.

Details The color, linewidth, linecap, linejoin, miterlimit, dash parameter, and the current transformation matrix (but not the text state parameters) are reset to their respective defaults. The current clipping path is not affected.

This function may be useful in situations where the program flow doesn't allow for easy use of *PDF_save( )/PDF_restore( )*.

Scope *page, pattern, template, glyph*

7.4.2 Saving and Restoring Graphics States

void PDF_save(PDF *p)

Save the current graphics state.

Details The graphics state contains parameters that control all types of graphics objects. Saving the graphics state is not required by PDF; it is only necessary if the application wishes to return to some specific graphics state later (e.g., a custom coordinate system) without setting all relevant parameters explicitly again. The following items are subject to save/restore:

- graphics parameters: clipping path, coordinate system, current point, flatness, line cap style, dash pattern, line join style, line width, miter limit;

- color parameters: fill and stroke colors;
- text position and other text-related parameters, see list below;
- some PDFlib parameters, see list below.

Pairs of *PDF_save( )* and *PDF_restore( )* may be nested. Although the PDF specification doesn't limit the nesting level of save/restore pairs, applications must keep the nesting level below 10 in order to avoid printing problems caused by restrictions in the PostScript output produced by PDF viewers, and to allow for additional save levels required by PDFlib internally.

Scope *page, pattern, template, glyph;* must always be paired with a matching *PDF_restore( )* call. *PDF_save( )* and *PDF_restore( )* calls must be balanced on each page, pattern, template, and glyph description.

Params The following parameters are subject to save/restore: *charspacing, wordspacing, horizscaling, leading, font, fontsize, textrendering, textrise;*

The following parameters are not subject to save/restore: *fillrule*, *kerning*, *underline*, *overline*, *strikeout*.

void PDF_restore(PDF *p)

Restore the most recently saved graphics state.

Details The corresponding graphics state must have been saved on the same page, pattern, or template.

Scope *page, pattern, template, glyph;* must always be paired with a matching *PDF_save( )* call. *PDF_save( )* and *PDF_restore( )* calls must be balanced on each page, pattern, template, and glyph description.

7.4.3 Coordinate System Transformation Functions

All transformation functions *(PDF_translate( )*, *PDF_scale( )*, *PDF_rotate( )*, *PDF_skew( ), PDF_concat( ), PDF_setmatrix( )*, and *PDF_initgraphics( ))* change the coordinate system used for drawing subsequent objects. They do not affect existing objects on the page at all.

void PDF_translate(PDF *p, float tx, float ty)

Translate the origin of the coordinate system.

tx, ty The new origin of the coordinate system is the point *(tx, ty)*, measured in the old coordinate system.

Scope *page, pattern, template, glyph*

void PDF_scale(PDF *p, float sx, float sy)

Scale the coordinate system.

sx, sy Scaling factors in *x* and *y* direction.

Details This function scales the coordinate system by *sx* and *sy*. It may also be used for achieving a reflection (mirroring) by using a negative scaling factor. One unit in the *x* direction in the new coordinate system equals *sx* units in the *x* direction in the old coordinate system; analogous for *y* coordinates.

Scope *page, pattern, template, glyph*

---

void PDF_rotate(PDF *p, float phi)

Rotate the coordinate system.

phi The rotation angle in degrees.

Details Angles are measured counterclockwise from the positive x axis of the current coordinate system. The new coordinate axes result from rotating the old coordinate axes by *phi* degrees.

Scope *page, pattern, template, glyph*

---

void PDF_skew(PDF *p, float alpha, float beta)

Skew the coordinate system.

alpha, beta Skewing angles in *x* and *y* direction in degrees.

Details Skewing (or shearing) distorts the coordinate system by the given angles in *x* and *y* direction. *alpha* is measured counterclockwise from the positive *x* axis of the current coordinate system, *beta* is measured clockwise from the positive *y* axis. Both angles must be in the range -360° < *alpha, beta* < 360°, and must be different from -270°, -90°, 90°, and 270°.

Scope *page, pattern, template, glyph*

---

void PDF_concat(PDF *p, float a, float b, float c, float d, float e, float f)

Concatenate a matrix to the current transformation matrix.

a, b, c, d, e, f Elements of a transformation matrix. The six floating point values make up a matrix in the same way as in PostScript and PDF (see references). In order to avoid degenerate transformations, *a*d* must not be equal to *b*c*.

Details This function concatenates a matrix to the current transformation matrix (CTM) for text and graphics. It allows for the most general form of transformations. Unless you are familiar with the use of transformation matrices, the use of *PDF_translate( ), PDF_scale( ), PDF_rotate( ),* and *PDF_skew( )* is suggested instead of this function. The CTM is reset to the identity matrix *[1, 0, 0, 1, 0, 0]* at the beginning of each page.

Scope *page, pattern, template, glyph*

void PDF_setmatrix(PDF *p, float a, float b, float c, float d, float e, float f)

Explicitly set the current transformation matrix.

a, b, c, d, e, f See *PDF_concat( )*.

Details This function is similar to *PDF_concat( )*. However, it disposes of the current transformation matrix, and completely replaces it with a new matrix.

Scope *page, pattern, template, glyph*

7.4.4 Explicit Graphics States

int PDF_create_gstate(PDF *p, const char *optlist)

Create a graphics state object definition.

optlist An option list (see Section 3.1.4, »Option Lists«, page 44) containing options for explicit graphics states according to Table 7.14.

Returns A gstate handle that can be used in subsequent calls to *PDF_set_gstate( )* during the enclosing *document* scope.

Details The option list may contain any number of graphics state parameters. Not all parameters are allowed for all PDF versions. The table lists the minimum required PDF version.

Scope *document, page, pattern, template, glyph*

Table 7.14 Options for PDF_create_gstate( )

key	type	explanation and possible values	PDF
alphaisshape	boolean	sources of alpha are treated as shape (true) or opacity (false). Default: false	1.4
blendmode	keyword list	Name of the blend mode. Multiple blend modes can be specified. Possible values: None, Normal, Multiply, Screen, Overlay, Darken, Lighten, ColorDodge, ColorBurn, HardLight, SoftLight, Difference, Exclusion, Hue, Saturation, Color, Luminosity. Default: None	1.4
flatness	float	maximum distance between a path and its approximation (see PDF_setflat( )), must be > 0.	1.3
linecap	integer	shape at the end of a path(see PDF_setlinecap( )); must be 0, 1, or 2.	1.3
linejoin	integer	shape at the corners of paths (see PDF_setlinejoin( )); must be 0, 1, or 2	1.3
linewidth	float	line width (see PDF_setlinewidth( )); must be > 0	1.3
miterlimit	float	controls the spike produced by miter joins, which must be >= 1 (see PDF_setmiterlimit( ))	1.3
opacityfill	float	constant alpha for fill operations; must be >= 0 and <= 1.	1.4
opacitystroke	float	constant alpha for stroke operations; must be >=0 and <=1	1.4
overprintfill	boolean	overprint for operations other than stroke. Default: false	1.3
overprintmode	integer	overprint mode. 0 (default) means that each color component replaces previously placed marks; 1 means that a color component of 0 leaves the corresponding component unchanged.	1.3
overprintstroke	boolean	overprint for stroke operations. Default: false	1.3

Table 7.14 Options for PDF_create_gstate( )

key	type	explanation and possible values	PDF
renderingintent	keyword	color rendering intent used for gamut compression; possible keywords: Auto, AbsoluteColorimetric, RelativeColorimetric, Saturation, Perceptual	1.3
smoothness	float	maximum error of a linear interpolation for a shading; must be >= 0 and <= 1	1.3
strokeadjust	boolean	whether or not to apply automatic stroke adjustment. Default: false	1.3
textknockout	boolean	with respect to compositing, glyphs in a text object will be treated as separate objects (false) or as a single object (true). Default: true	1.4

void PDF_set_gstate(PDF *p, int gstate)

Activate a graphics state object.

gstate A handle for a graphics state object retrieved with *PDF_create_gstate( ).*

Details All options contained in the graphics state object will be set. Graphics state options accumulate when this function is called multiply. Options which are not explicitly set in the graphics state object will keep their current values. All graphics state options will be reset to their default values at the beginning of a page.

Scope *page, pattern, template, glyph*

7.4.5 Path Construction

Table 7.15 lists relevant parameters and values for this section.

Table 7.15 Parameters and values for path segment functions (see Section 7.2.3, »Parameter Handling«, page 153)

function	key	explanation
get_value	currentx currenty	The x or y coordinate (in units of the current coordinate system), respectively, of the current point. Scope: page, pattern, template, path

Note *Make sure to call one of the functions in Section 7.4.6, »Path Painting and Clipping«, page 180, after using the functions in this section, or the constructed path will have no effect, and subsequent operations may raise a PDFlib exception.*

void PDF_moveto(PDF *p, float x, float y)

Set the current point.

x, y The coordinates of the new current point.

Details The current point is set to the default value of *undefined* at the beginning of each page. The current points for graphics and the current text position are maintained separately.

Scope *page, pattern, template, path, glyph;* this function starts *path* scope.

Params *currentx, currenty*

void PDF_lineto(PDF *p, float x, float y)

Draw a line from the current point to another point.

x, y The coordinates of the second endpoint of the line.

Details This function adds a straight line from the current point to *(x, y)* to the current path. The current point must be set before using this function. The point *(x, y)* becomes the new current point.

The line will be centered around the »ideal« line, i.e. half of the linewidth (as determined by the value of the linewidth parameter) will be painted on each side of the line connecting both endpoints. The behavior at the endpoints is determined by the value of the linecap parameter.

Scope *path*

Params *currentx, currenty*

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, using three more control points.

x1, y1, x2, y2, x3, y3 The coordinates of three control points.

Details A Bézier curve is added to the current path from the current point to *(x3, y3)*, using *(x1, y1)* and *(x2, y2)* as control points. The current point must be set before using this function. The endpoint of the curve becomes the new current point.

Scope *path*

Params *currentx, currenty*

void PDF_circle(PDF *p, float x, float y, float r)

Draw a circle.

x, y The coordinates of the center of the circle.

r The radius of the circle.

Details This function adds a circle to the current path as a complete subpath. The point *(x + r, y)* becomes the new current point. The resulting shape will be circular in user coordinates. If the coordinate system has been scaled differently in *x* and *y* directions, the resulting curve will be elliptical.

Scope *page, pattern, template, path, glyph;* this function starts *path* scope.

Params *currentx, currenty*

void PDF_arc(PDF *p, float x, float y, float r, float alpha, float beta)

Draw a counterclockwise circular arc segment.

x, y The coordinates of the center of the circular arc segment.

r The radius of the circular arc segment. *r* must be nonnegative.

alpha, beta The start and end angles of the circular arc segment in degrees.

Details This function adds a counterclockwise circular arc segment to the current path, extending from *alpha* to *beta* degrees. For both *PDF_arc( )* and *PDF_arcn( )*, angles are measured counterclockwise from the positive x axis of the current coordinate system. If there is a current point an additional straight line is drawn from the current point to the starting point of the arc. The endpoint of the arc becomes the new current point.

The arc segment will be circular in user coordinates. If the coordinate system has been scaled differently in *x* and *y* directions the resulting curve will be elliptical.

Scope *page, pattern, template, path, glyph;* this function starts *path* scope.

Params *currentx, currenty*

---

void PDF_arcn(PDF *p, float x, float y, float r, float alpha, float beta)

Like *PDF_arc( )*, but draws a clockwise circular arc segment.

Details Except for the drawing direction, this function behave exactly like *PDF_arc( )*. In particular, the angles are still measured *counterclockwise* from the positive *x* axis.

---

void PDF_rect(PDF *p, float x, float y, float width, float height)

Draw a rectangle.

x, y The coordinates of the lower left corner of the rectangle.

width, height The size of the rectangle.

Details This function adds a rectangle to the current path as a complete subpath. Setting the current point is not required before using this function. The point *(x, y)* becomes the new current point. The lines will be centered around the »ideal« line, i.e. half of the linewidth (as determined by the value of the linewidth parameter) will be painted on each side of the line connecting the respective endpoints.

Scope *page, pattern, template, path, glyph;* this function starts *path* scope.

Params *currentx, currenty*

---

void PDF_closepath(PDF *p)

Close the current path.

Details This function closes the current subpath, i.e., adds a line from the current point to the starting point of the subpath.

Scope *path*

Params *currentx, currenty*

7.4.6 Path Painting and Clipping

Table 7.16 lists relevant parameters and values for this section.

Table 7.16 Parameters and values for path painting and clipping functions (see Section 7.2.3, »Parameter Handling«, page 153)

function	*key*	*explanation*
set_parameter	*fillrule*	*Set the current fill rule to winding or evenodd. The fill rule is used by PDF viewers to determine the interior of shapes for the purpose of filling or clipping. Since both algorithms yield the same result for simple shapes, most applications won't have to change the fill rule. The fill rule is reset to the default of winding at the beginning of each page. Scope: page, pattern, template, glyph.*

Note *Most functions in this section clear the path, and leave the current point undefined. Subsequent drawing operations must therefore explicitly set the current point (e.g., using PDF_moveto( )) after one of these functions has been called.*

void PDF_stroke(PDF *p)

Stroke the path with the current line width and current stroke color, and clear it.

Scope *path;* this function terminates *path* scope.

void PDF_closepath_stroke(PDF *p)

Close the path, and stroke it.

Details This function closes the current subpath (adds a straight line segment from the current point to the starting point of the path), and strokes the complete current path with the current line width and the current stroke color.

Scope *path;* this function terminates *path* scope.

void PDF_fill(PDF *p)

Fill the interior of the path with the current fill color.

Details This function fills the interior of the current path with the current fill color. The interior of the path is determined by one of two algorithms (see the *fillrule* parameter). Open paths are implicitly closed before being filled.

Scope *path;* this function terminates *path* scope.

Params *fillrule*

void PDF_fill_stroke(PDF *p)

Fill and stroke the path with the current fill and stroke color.

Scope *path;* this function terminates *path* scope.

Params *fillrule*

void PDF_closepath_fill_stroke(PDF *p)

Close the path, fill, and stroke it.

Details This function closes the current subpath (adds a straight line segment from the current point to the starting point of the path), and fills and strokes the complete current path.

Scope *path;* this function terminates *path* scope.

Params *fillrule*

void PDF_clip(PDF *p)

Use the current path as clipping path, and terminate the path.

Details This function uses the intersection of the current path and the current clipping path as the clipping path for subsequent operations. The clipping path is set to the default value of the page size at the beginning of each page. The clipping path is subject to *PDF_save( )/PDF_restore( )*. It can only be enlarged by means of *PDF_save( )/PDF_restore( )*.

Scope *path;* this function terminates *path* scope.

void PDF_endpath(PDF *p)

End the current path without filling or stroking it.

Details This function doesn't have any visible effect on the page. It generates an invisible path on the page.

Scope *path;* this function terminates *path* scope.

7.5 Color Functions

7.5.1 Setting Color and Color Space

Table 7.17 lists relevant parameters and values for this section.

Table 7.17 Parameter for color functions

function	*key*	*explanation*
set_parameter	*preserveold-pantonenames*	*If false, old-style Pantone spot color names will be converted to the corresponding new color names, otherwise they will be preserved. Default: false. Scope: any*
set_parameter	*spotcolor-lookup*	*If false, PDFlib will not use its internal database of spot color names. This can be used to provide custom definitions of known spot colors, which may be required as a workaround to match the definitions used by other applications. This feature should be used with care, and is not recommended. Default: true. Scope: any*

void PDF_setcolor(PDF *p,
const char *fstype, const char *colorspace, float c1, float c2, float c3, float c4)

Set the current color space and color.

fstype One of *fill, stroke,* or *fillstroke* to specify that the color is set for filling, stroking, or both. The deprecated value *both* is equivalent to *fillstroke*.

colorspace One of *gray, rgb, cmyk, spot, pattern, iccbasedgray, iccbasedrgb, iccbasedcmyk,* or *lab* to specify the color space.

c1, c2, c3, c4 Color components for the chosen color space:

- If *colorspace* is *gray, c1* specifies a gray value;
- If *colorspace* is *rgb, c1, c2, c3* specify red, green, and blue values;
- If *colorspace* is *cmyk, c1, c2, c3, c4* specify cyan, magenta, yellow, and black values;
- If *colorspace* is *spot*, *c1* specifies a spot color handle returned by *PDF_makespotcolor( ),* and *c2* specifies a tint value between 0 and 1;
- If *colorspace* is *pattern*, *c1* specifies a pattern handle returned by *PDF_begin_pattern( )* or *PDF_shading_pattern( ).*
- If *colorspace* is *iccbasedgray*, *c1* specifies a gray value;
- If *colorspace* is *iccbasedrgb, c1, c2, c3* specify red, green, and blue values;
- If *colorspace* is *iccbasedcmyk, c1, c2, c3, c4* specify cyan, magenta, yellow, and black values;
- If *colorspace* is *lab, c1, c2,* and *c3* specify color values in the CIE L*a*b* color space. *c1* specifies the L* (luminance) in the range 0 to 100, and *c2, c3* specify the a*, b* (chrominance) values in the range -127 to 128.

Details All color values for the *gray, rgb,* and *cmyk* color spaces and the *tint* value for the *spot* color space must be numbers in the inclusive range 0–1. Unused parameters should be set to 0.

Grayscale, RGB values and spot color tints are interpreted according to additive color mixture, i.e., 0 means no color and 1 means full intensity. Therefore, a gray value of 0 and RGB values with *(r, g, b) = (0, 0, 0)* mean black; a gray value of 1 and RGB values with *(r, g, b) = (1, 1, 1)* mean white. CMYK values, however, are interpreted according to subtractive color mixture, i.e., *(c, m, y, k) = (0, 0, 0, 0)* means white and *(c, m, y, k) = (0, 0, 0, 1)* means black. Color values in the range 0–255 must be scaled to the range 0–1 by dividing by 255.

The fill and stroke color values for the *gray, rgb,* and *cmyk* color spaces are set to a default value of black at the beginning of each page. There are no defaults for spot and pattern colors.

If the *iccbasedgray/rgb/cmyk* color spaces are used, the corresponding ICC profile must have been set before using the *setcolor:iccprofilegray/rgb/cmyk* parameters (see Table 7.19).

The CIE L*a*b* color space is interpreted with a D50 illuminant.

Scope *page, pattern* (only if the pattern's paint type is 1), *template, glyph* (only if the Type 3 font's *colorized* option is true), *document;* a pattern color can not be used within its own definition. Setting the color in *document* scope may be useful for defining spot colors with *PDF_makespotcolor( )*.

PDF/X PDF/X-1 and PDF/X-1a: *colorspace = rgb*, *iccbasedgray/rgb/cmyk*, and *lab* are not allowed. PDF/X-3: *colorspace = gray* requires that the *defaultgray* parameter has been set before unless the PDF/X output intent is a grayscale or CMYK device. *colorspace = rgb* requires that the *defaultrgb* parameter has been set before unless the PDF/X output intent is an RGB device. *colorspace = cmyk* requires that the *defaultcmyk* parameter has been set before unless the PDF/X output intent is a CMYK device. Using *iccbasedgray/rgb/cmyk* and *lab* color requires an ICC profile in the output intent (a standard name is not sufficient in this case).

Params *setcolor:iccprofilegray/rgb/cmyk*

---

int PDF_makespotcolor(PDF *p, const char *spotname, int reserved)

Find a built-in spot color name, or make a named spot color from the current fill color.

spotname The name of a built-in spot color, or an arbitrary name for the spot color to be defined. This name is restricted to a maximum length of 126 bytes. Only 8-bit characters are supported in the spot color name; Unicode or embedded null characters are not supported.

reserved (C language binding only.) Reserved, must be 0.

Returns A color handle which can be used in subsequent calls to *PDF_setcolor( )* throughout the document. Spot color handles can be reused across all pages, but not across documents. There is no limit for the number of spot colors in a document.

Details If *spotname* is known in the internal color tables and the *spotcolorlookup* parameter is *true* (which is default), the supplied spot color name will be used. Otherwise the (CMYK or other) color values of the current fill color will be used to define the appearance of a new spot color. These alternate values will only be used for screen preview and low-end printing. The supplied spot color name will be used for producing color separations instead of the alternate values.

If *spotname* has already been used in a previous call to *PDF_makespotcolor( )*, the return value will be the same as in the earlier call, and will not reflect the current color.

The special spot color name *All* can be used to apply color to all color separations, which is useful for painting registration marks. A spot color name of *None* will produce no visible output on any color separation.

Scope *page, pattern, template, glyph* (only if the Type 3 font's *colorized* option is true), *document;* the current fill color must not be a spot color or pattern if a custom color is to be defined.

PDF/X PANTONE® Colors are currently not supported in the PDF/X-1 and PDF/X-1a modes.

Params *spotcolorlookup, preserveoldpantonenames*

int PDF_load_iccprofile(PDF *p, const char *profilename, int reserved, const char *optlist)

Search a ICC profile, and prepare it for later use.

profilename The name of an *ICCProfile* resource, a disk-based or virtual file name, or a standard output condition name for PDF/X. The latter is only allowed if the *usage* option is set to *outputintent*.

reserved (C language binding only.) Reserved, must be 0.

optlist An option list (see Section 3.1.4, »Option Lists«, page 44) describing aspects of the profile handling according to Table 7.18.

Table 7.18 Options for PDF_load_iccprofile( )

key	*type*	*explanation and possible values*
usage	*keyword*	*Describes the intended use of the ICC profile (default: iccbased):* *iccbased: the ICC profile will be used to define an ICC-based color space, or will be applied to an image.* *outputintent: the ICC profile will be used to define a PDF/X output intent.*
description	*string*	*This option is only used if usage = outputintent. It contains the human-readable description of the ICC profile which will be used along with the PDF/X output intent. Default: if profilename refers to a standard output intent, the description will be taken from an internal list; otherwise there will be no description.*
embedprofile	*boolean*	*This option is only used if usage = outputintent. Force an embedded ICC profile even if a standard output intent has been provided as profilename. Default: false.*

Returns A profile handle which can be used in subsequent calls to *PDF_load_image( )* or for setting profile-related parameters. The return value must be checked for -1 (in PHP: 0) which signals an error. In order to get more detailed information about the nature of a profile-related problem (file not found, unsupported format, etc.) set the *iccwarning* parameter to *true*. The returned profile handle can not be reused across multiple PDF documents. Also, the returned handle can not be applied to an image if the *usage* option is *outputintent*. There is no limit to the number of ICC profiles in a document.

Details If the *usage* option is *iccbased* the named profile will be searched according to the search strategy discussed in Section 3.3.4, »Color Management and ICC Profiles«, page 63. If the profile is found, it will be checked whether it is suitable (e.g., number of color components). The *sRGB* profile is always available internally, and must not be configured.

If *usage* option is *outputintent* the named profile is first searched in an internal list of standard output intents. If this search was unsuccessful, the name will be searched in the list of user-configured output intents. If the supplied name was found to be a standard output intent according to the built-in or user-configured list, no ICC profile will

be searched, and the name supplied with the description option will be embedded in the PDF output as the PDF/X output intent. If the name was not found to be a standard output intent identifier, it is treated as a profile name and the corresponding ICC profile will be embedded in the PDF as the PDF/X output intent.

Scope *document;* if the *usage* option is *iccbased* the following scopes are also allowed: *page, pattern, template, glyph.*

Params See Table 7.19.

PDF/X The output intent for the generated document must be set either using this function, or by copying an imported document's output intent using *PDF_process_pdi( )*.

Table 7.19 Parameters and values for ICC profiles

function	*key*	*explanation*
set_parameter	*ICCProfile StandardOutputIntent*	*The corresponding resource file line as it would appear for the respective category in a UPR file (see Section 3.1.6, »Resource Configuration and File Searching«, page 47). Multiple calls add new entries to the internal list. (See also prefix and resourcefile in Table 7.2). Scope: any.*
set_parameter	*iccwarning*	*Enable or suppress warnings (nonfatal exceptions) related to ICC profiles. Possible values are true and false, default value is false. Scope: any*
get_value	*icccomponents*	*Return the number of color components in the ICC profile referenced by the handle provided in the modifier.*
set_value	*defaultgray defaultrgb defaultcmyk*	*Set a default Gray, RGB, or CMYK color space for the page according to the supplied profile handle.*
set_value	*setcolor:iccprofilegray*	*Set an ICC profile which specifies a Gray color space for use with PDF_setcolor( ). Scope: document, page, pattern, template, glyph*
set_value	*setcolor:iccprofilergb*	*Set an ICC profile which specifies an RGB color space for use with PDF_setcolor( ). Scope: document, page, pattern, template, glyph*
set_value	*setcolor:iccprofilecmyk*	*Set an ICC profile which specifies a CMYK color space for use with PDF_setcolor( ). Scope: document, page, pattern, template, glyph*

7.5.2 Patterns and Shadings

int PDF_begin_pattern(PDF *p,
float width, float height, float xstep, float ystep, int painttype)

Start a pattern definition.

width, height The dimensions of the pattern's bounding box in points.

xstep, ystep The offsets when repeatedly placing the pattern to stroke or fill some object. Most applications will set these to the pattern *width* and *height*, respectively.

painttype If *painttype* is 1 the pattern must contain its own color specification which will be applied when the pattern is used; if *painttype* is 2 the pattern must not contain any color specification but instead the current fill or stroke color will be applied when the pattern is used for filling or stroking.

Returns A pattern handle that can be used in subsequent calls to *PDF_setcolor( )* during the enclosing *document* scope.

Details Hypertext functions and functions for opening images must not be used during a pattern definition, but all text, graphics, and color functions (with the exception of the pattern which is in the process of being defined) can be used.

Scope *document*; this function starts *pattern* scope, and must always be paired with a matching *PDF_end_pattern( )* call.

void PDF_end_pattern(PDF *p)

Finish a pattern definition.

Scope *pattern;* this function terminates *pattern* scope, and must always be paired with a matching *PDF_begin_pattern( )* call.

int PDF_shading_pattern(PDF *p, int shading, const char *optlist)

Define a shading pattern using a shading object.

shading A shading handle returned by *PDF_shading( )*.

optlist An option list (see Section 3.1.4, »Option Lists«, page 44) describing aspects of the shading pattern according to Table 7.20.

Returns A pattern handle that can be used in subsequent calls to *PDF_setcolor( )* during the enclosing *document* scope.

Details This function can be used to fill arbitrary objects with a shading. To do so, a shading handle must be retrieved using *PDF_shading( ),* then a pattern must be defined based on this shading using *PDF_shading_pattern( ).* Finally, the pattern handle can be supplied to *PDF_setcolor( )* to set the current color to the shading pattern.

Scope *document, page, font*

Table 7.20 Options for PDF_shading_pattern( )

key	*type*	*explanation and possible values*	*PDF*
gstate	*handle*	*A graphics state handle*	*1.3*

void PDF_shfill(PDF *p, int shading)

Fill an area with a shading, based on a shading object.

shading A shading handle returned by *PDF_shading( )*.

Details This function allows shadings to be used without involving *PDF_shading_pattern( )* and *PDF_setcolor( ).* However, it works only for simple shapes where the geometry of the object to be filled is the same as that of the shading itself. Since the current clip area will be shaded (subject to the *extendo* and *extend1* options of the shading) this function will generally be used in combination with *PDF_clip( )*.

Scope *page, pattern* (only if the pattern's paint type is 1)*, template, glyph* (only if the Type 3 font's *colorized* option is true)*, document*

int PDF_shading(PDF *p, const char *shtype, float x0, float y0, float x1, float y1, float c1, float c2, float c3, float c4, const char *optlist)

Define a color blend (smooth shading) from the current fill color to the supplied color.

shtype The type of the shading; must be *axial* for linear shadings or *radial* for circle-like shadings.

x0, y0, x1, y1 For axial shadings, *(x0, y0)* and *(x1, y1)* are the coordinates of the starting and ending points of the shading. For radial shadings these points specify the centers of the starting and ending circles.

c1, c2, c3, c4 Color values of the shading's endpoint, interpreted in the current fill color space in the same way as the color parameters in *PDF_setcolor( )*. If the current fill color space is a spot color space *c1* will be ignored, and *c2* contains the tint value.

optlist An option list (see Section 3.1.4, »Option Lists«, page 44) describing aspects of the shading according to Table 7.21.

Returns A shading handle that can be used in subsequent calls to *PDF_shading_pattern( )* and *PDF_shfill( )* during the enclosing *document* scope.

Details The current fill color will be used as the starting color; it must not be based on a pattern.

Scope *document, page, font*

Table 7.21 Options for PDF_shading( )

key	*type*	*explanation and possible values*	*PDF*
N	*float*	*The exponent for the color transition function; must be > 0. Default: 1*	*1.3*
r0	*float*	*(Only for radial shadings, and required in this case) The radius of the starting circle.*	*1.3*
r1	*float*	*(Only for radial shadings, and required in this case) The radius of the ending circle.*	*1.3*
extend0	*boolean*	*Specifies whether to extend the shading beyond the starting point. Default: false*	*1.3*
extend1	*boolean*	*Specifies whether to extend the shading beyond the endpoint. Default: false*	*1.3*
antialias	*boolean*	*Specifies whether to activate antialiasing for the shading. Default: false*	*1.3*

7.6 Image and Template Functions

Table 7.22 lists relevant parameters and values for this section.

Table 7.22 Parameters and values for the image functions (see Section 7.2.3, »Parameter Handling«, page 153)

function	key	explanation
get_value	imagewidth imageheight	Get the width or height, respectively, of an image in pixels. The modifier is the integer handle of the selected image. Scope: page, pattern, template, glyph, document, path.
get_value	resx resy	Get the horizontal or vertical resolution of an image, respectively. The modifier is the integer handle of the selected image. Scope: page, pattern, template, glyph, document, path. If the value is positive, the return value is the image resolution in pixels per inch (dpi). If the return value is negative, resx and resy can be used to find the aspect ratio of non-square pixels, but don't have any absolute meaning. If the return value is zero, the resolution of the image is unknown.
set_parameter	honoriccprofile	Read ICC color profiles embedded in images, and apply them to the image data. Default: true.
set_parameter	imagewarning	This parameter can be used in order to obtain more detailed information about why an image couldn't be opened successfully with PDF_load_image( ). Scope: any. true Raise an exception when the image function fails, and return -1 (in PHP: 0). The message supplied with the exception may be useful for debugging. false Do not raise an exception when the image function fails. Instead, the function simply returns -1 (in PHP: 0) on error. This is the default.
set_value	image:iccprofile	Handle for an ICC profile which will be applied to all respective images unless the iccprofile option is supplied.
get_value	image:iccprofile	Return a handle for the ICC profile embedded in the image referenced by the image handle provided in the modifier.
set_value	renderingintent	The rendering intent for images. See Table 3.8 for a list of possible keywords and their meaning. Default: Auto.

7.6.1 Images

int PDF_load_image(PDF *p,
const char *imagetype, const char *filename, int reserved, const char *optlist)

Open a (disk-based or virtual) image file with various options.

imagetype The string *auto* instructs PDFlib to automatically detect the image file type (this is not possible for CCITT and raw images, and must not be used if the *reftype* option has the value *url* or *fileref)*. Explicitly specifying the image format with the strings *bmp, ccitt, gif, jpeg, png*, *raw*, or *tiff* offers slight performance advantages (for details see Section 5.1.2, »Supported Image File Formats«, page 112). Type *ccitt* is different from a TIFF file which contains CCITT-compressed image data.

filename Generally the name of the image file to be opened. This must be the name of a local or virtual file; PDFlib will not pull image data from URLs. However, if *reftype=url* (deprecated, though) this parameter contains the URL of the image; If *reftype=fileref* (al-

so deprecated) it contains a file name on the viewing system, and not a local file name. Only 8-bit characters are supported in the image file name; Unicode or embedded null characters are not supported.

reserved (C language binding only.) Reserved, must be 0.

optlist An option list (see Section 3.1.4, »Option Lists«, page 44) specifying image-related properties according to Table 7.23.

Returns An image handle which can be used in subsequent image-related calls. The return value must be checked for -1 (in PHP: 0) which signals an error. In order to get more detailed information about the nature of an image-related problem (wrong image file name, unsupported format, bad image data, etc.), set the *imagewarning* parameter or option to *true* (see Table 7.22 and Table 7.23). The returned image handle can not be reused across multiple PDF documents.

Details This function opens and analyzes a raster graphics file in one of the supported formats as determined by the *imagetype* parameter. PDFlib will open the image file with the provided *filename*, process the contents, and close the file before returning from this call. Although images can be placed multiply within a document (see *PDF_fit_image( )*), the actual image file will not be kept open after this call.

If *imagetype* = *raw* or *ccitt*, the *width, height, components,* and *bpc* options must be supplied since PDFlib cannot deduce those from the image data. The user is responsible for supplying option values which actually match the image. Otherwise corrupt PDF output may be generated, and Acrobat may respond with the message *Insufficient data for an Image*.

If *imagetype* = *raw*, the length of the supplied image data must be equal to *[width x components x bpc / 8] x height* bytes, with the bracketed term adjusted upwards to the next integer. The image samples are expected in the standard PostScript/PDF ordering, i.e., top to bottom and left to right (assuming no coordinate transformations have been applied). The polarity of the pixel values is as discussed in Section 3.3.1, »Color and Color Spaces«, page 59. Even if *bpc* is not 8, each pixel row begins on a byte boundary, and color values must be packed from left to right within a byte. Image samples are always interleaved, i.e., all color values for the first pixel are supplied first, followed by all color values for the second pixel, and so on.

Scope If the *inline* option is not provided, the scope is *document, page, font,* and this function must always be paired with a matching *PDF_close_image( )* call. Loading images in *document* or *font* scope instead of *page* scope offers slight output size advantages.
If the *inline* option is provided, the scope is *page, pattern, template, glyph,* and *PDF_close_image( )* must not be called.

PDF/X All PDF/X conformance levels: GIF and LZW-compressed TIFF images are not allowed.
PDF/X-1 and PDF/X-1a: RGB images are not allowed.
PDF/X-3: Grayscale images require that the *defaultgray* parameter must have been set before unless the PDF/X output intent is a grayscale or CMYK device. RGB images require that the *defaultrgb* parameter must have been set before unless the PDF/X output intent is an RGB device. CMYK images require that the *defaultcmyk* parameter must have been set before unless the PDF/X output intent is a CMYK device.

Params *imagewidth, imageheight, resx, resy, imagewarning*

Table 7.23 Options for PDF_load_image( )

key	type	explanation
bitreverse	boolean	(Only for imagetype = ccitt) If true, do a bitwise reversal of all bytes in the compressed data. Default: false.
bpc	integer	(Only for imagetype = raw, and required in this case) The number of bits per component; must be 1, 2, 4, or 8.
colorize	spot color handle	Colorize the image with a spot color handle, which must have been retrieved with PDF_makespotcolor( ). The image must be a grayscale image with 1, 2, 4, or 8 bits color depth.
components	integer	(Only for imagetype=raw, and required in this case) The number of image components (channels); must be 1, 3, or 4.
height	integer	(Only for imagetype = raw and ccitt, and required in this case) The height of the image in pixels.
honor-iccprofile	boolean	(Only for imagetype = jpeg, png, and tiff) Read an embedded ICC profile (if any) and apply it to the image. Default: the value of the global honoriccprofile parameter.
iccprofile	icc handle	(Only for imagetype = jpeg, png, and tiff) Handle of the ICC profile which will be applied to the image. Default: the value of the global image:iccprofile parameter.
ignoremask	boolean	Ignore any transparency information which may be present in the image. Default: false.
image-warning	boolean	Enable exceptions when the image cannot be opened. Default: the value of the global imagewarning parameter.
inline	boolean	(Only for imagetype = ccitt, jpeg, and raw) If true, the image will be written directly into the content stream of the page, pattern, template, or glyph description (see Section 5.1.1, »Basic Image Handling«, page 111).
interpolate	boolean	Enable image interpolation in Acrobat to improve the appearance on screen and paper. This is especially useful for bitmap images to be used as glyph descriptions in Type 3 fonts. Default: false.
invert	boolean	Invert the image (swap light and dark colors). This can be used as a workaround for certain images which are interpreted differently by different applications. Default: false.
K	integer	(Only for imagetype = ccitt) CCITT compression parameter for encoding scheme selection. -1 indicates G4 compression; 0 indicates one-dimensional G3 compression (G3-1D); 1 indicates mixed one- and two-dimensional compression (G3, 2-D) as supported by PDF. Default: 0.
mask	boolean	The image is going to be used as a mask (see Section 5.1.3, »Image Masks and Transparency«, page 114). This is required for 1-bit masks, but optional for masks with more than 1 bit per pixel. However, masks with more than 1 bit require PDF 1.4. Default is false. There are two uses for masks: Masking another image: the returned image handle may be used in subsequent calls for opening another image and can be supplied for the »masked« option. Placing a colorized transparent image: treat the 0-bit pixels in the image as transparent, and colorize the 1-bit pixels with the current fill color.
masked	image handle	An image handle for an image which will be applied as a mask to the current image. The integer is an image handle which has been returned by a previous call to PDF_load_image( ) (with the »mask« option if it is a 1-bit mask), and has not yet been closed. In PDF 1.3 compatibility mode the image handle must refer to a 1-bit image since soft masks are only supported in PDF 1.4.
page	integer	(Only for imagetype = gif or tiff; must be 1 if used with other formats) Extract the image with the given number from a multi-page image file. The first image has the number 1. Default: 1.

Table 7.23 Options for PDF_load_image( )

key	type	explanation
rendering-intent	*keyword*	*The rendering intent for the image. See Table 3.8 for a list of possible keywords and their meaning. Default: the value of the global renderingintent parameter.*
reftype	*keyword*	*(Deprecated; only for imagetype = ccitt, jpeg, and raw) One of direct (for image data to be read from a file), fileref or url to specify image data via a reference to a file on the viewing system or an URL. The following options are required for reftype=url or fileref: width, height, components, bpc; the following options will be ignored for reftype=url or fileref: bitreverse, honoriccprofile, inline. Default: direct.*
width	*integer*	*(Only for imagetype = raw and ccitt, and required in this case) The width of the image in pixels.*

void PDF_close_image(PDF *p, int image)

Close an image.

image A valid image handle retrieved with *PDF_load_image( )*.

Details This function only affects PDFlib's associated internal image structure. If the image has been opened from file, the actual image file is not affected by this call since it has already been closed at the end of the corresponding *PDF_load_image( )* call. An image handle cannot be used any more after it has been closed with this function, since it breaks PDFlib's internal association with the image.

Scope *document, page, font;* must always be paired with a matching call to *PDF_load_image( )* unless the *inline* option has been used.

void PDF_fit_image(PDF *p, int im, float x, float y, const char *optlist)

Place an image or template at *(x, y)* with various options.

image A valid image or template handle retrieved with one of the *PDF_load_image*( )* or *PDF_begin_template( )* functions.

x, y The coordinates of the reference point in the user coordinate system where the image or template will be located, subject to various options.

optlist An option list (see Section 3.1.4, »Option Lists«, page 44) specifying placement details according to Table 7.24.

Details The image or template (collectively referred to as an object below) will be placed relative to the reference point *(x, y)*. By default, the lower left corner of the object will be placed at the reference point. However, the *orientate, boxsize, position,* and *fitmethod* options can modify this behavior. By default, an image will be scaled according to its resolution value(s). This behavior can be modified with the *dpi, scale,* and *fitmethod* options.

Scope *page, pattern, template, glyph* (only if the Type 3 font's *colorized* option is true, or if the image is a mask); this function can be called an arbitrary number of times on arbitrary pages, as long as the image handle has not been closed with *PDF_close_image( )*.

Table 7.24 Options for PDF_fit_image( ) and PDF_fit_pdi_page( )

key	type	explanation
adjustpage	boolean	Adjust the dimensions of the current page to the object such that the upper right corner of the page coincides with the upper right corner of the object plus (x, y). With the value 0 for the position option the following useful cases shall be noted: x >= 0 and y >= 0: the object is surrounded by a white margin. This margin has thickness y in horizontal direction and thickness x in vertical direction. x < 0 and y < 0: horizontal and vertical strips will be cropped from the image. This option is only effective in scope page, and must not be used when the topdown parameter has been set to true. Default: false.
blind	boolean	If true, all positioning and scaling calculations will be done, but the object will not be placed on the output page. This is useful for processing the blocks on a page without actually using the page's contents. Default: false.
boxsize	float list	Two values specifying the width and height of a box, relative to which the object will be placed and possibly scaled. The lower left corner of the box coincides with the reference point (x, y). Placing the image and fitting it into the box is controlled by the position and fitmethod options. If width = 0, only the height is considered; If height = 0, only the width is considered. In these cases the object will be placed relative to the vertical line from (x, y) to (x, y+height), or the horizontal line from (x, y) to (x+width, y), respectively. Default: {0 0}.
dpi	float list	One or two values specifying the desired image resolution in pixels per inch in horizontal and vertical direction. If a single value is supplied it will be used for both directions. With the value 0 the image's internal resolution will be used if available, or 72 dpi otherwise. As an alternative to the value 0, the keyword internal can be supplied. This option will be ignored for templates and PDF pages, or if the fitmethod option has been supplied with one of the keywords meet, slice, or entire. Default: internal.
fitmethod	keyword	Specifies the method used to fit the object into the specified box. This option will be ignored if no box has been specified. Default: nofit. nofit — Position the object only, without any scaling or clipping. clip — Position the object, and clip it at the edges of the box. meet — Position the object according to the position option, and scale it such that it entirely fits into the box while preserving its aspect ratio. Generally at least two edges of the object will meet the corresponding edges of the box. The dpi and scale options are ignored. auto — This method tries to fit the object into the box automatically. In detail: Same as nofit if the object fits into the box. Otherwise a scaling factor is calculated such that the object fits into the box. If this factor is larger than 0.75 the object is distorted to fit intothe box, otherwise the meet method is applied. slice — Position the object according to the position option, and scale it such that it entirely covers the box, while preserving the aspect ratio and making sure that at least one dimension of the object is fully contained in the box. Generally parts of the object's other dimension will extend beyond the box, and will therefore be clipped. The dpi and scale options are ignored. entire — Position the object according to the position option, and scale it such that it entirely covers the box. Generally this method will distort the object. The dpi and scale options are ignored.
orientate	keyword	Specifies the desired orientation of the object when it is placed. Default: north. north — upright east — pointing to the right south — upside down west — pointing to the left

Table 7.24 Options for PDF_fit_image( ) and PDF_fit_pdi_page( )

key	*type*	*explanation*
position	*float list*	*One or two values specifying the position of the reference point (x, y) within the object with {0 0} being the lower left corner of the object, and {100 100} the upper right corner. If the boxsize option has been specified, the position option also specifies the positioning of the box. The values are expressed as percentages of the object's width and height. If both percentages are equal it is sufficient to specify a single float value. Some examples: 0 or {0 0} means lower left corner; {50 100} means middle of the top edge; 50 or {50 50} means the center of the object. Default: 0.*
rotate	*float*	*Rotate the coordinate system, using the reference point as center and the specified value as rotation angle in degrees. This results in the box and the object being rotated. The rotation will be reset when the object has been placed. Default: 0.*
scale	*float list*	*Scale the object in horizontal and vertical direction by the specified scaling factors (not percentages). If both factors are equal it is sufficient to specify a single float value. This option will be ignored if the fitmethod option has been supplied with one of the keywords meet, slice, or entire. Default: 1*

7.6.2 Templates

int PDF_begin_template(PDF *p, float width, float height)

Start a template definition.

width, height The dimensions of the template's bounding box in points.

Returns A template handle which can be used in subsequent image-related calls, especially *PDF_fit_image( )*. There is no error return.

Details Hypertext functions and functions for opening images must not be used during a template definition, but all text, graphics, and color functions can be used.

Scope *document;* this function starts *template* scope, and must always be paired with a matching *PDF_end_template( )* call.

void PDF_end_template(PDF *p)

Finish a template definition.

Scope *template;* this function terminates *template* scope, and must always be paired with a matching *PDF_begin_template( )* call.

7.6.3 Deprecated Functions

int PDF_open_image_file(PDF *p,
const char *imagetype, const char *filename, const char *stringparam, int intparam)

Deprecated, use *PDF_load_image( )* with the *colorize, ignoremask, invert, mask, masked,* and *page* options instead.

int PDF_open_CCITT(PDF *p,
const char *filename, int width, int height, int BitReverse, int K, int BlackIs1)

Deprecated, use *PDF_load_image( )* with *type = ccitt* and the options *width, height, bitreverse, K,* and *invert* instead.

int PDF_open_image(PDF *p, const char *imagetype, const char *source, const char *data,
long length, int width, int height, int components, int bpc, const char *params)

Deprecated, use virtual files and *PDF_load_image( )* with *type = jpeg*, *ccitt*, or *raw* and the options *width, height, components, bpc, mask, invert, reftype,* and *inline* instead.

void PDF_place_image(PDF *p, int image, float x, float y, float scale)

Deprecated, use *PDF_fit_image( )* instead.

7.7 PDF Import (PDI) Functions

Note *All functions described in this section require the additional PDF import library (PDI) which is not part of PDFlib Lite and PDFlib. Please visit our Web site for more information on obtaining PDI.*

7.7.1 Document and Page

int PDF_open_pdi(PDF *p, const char *filename, const char *optlist, int reserved)

Open a (disk-based or virtual) PDF document and prepare it for later use.

filename The name of the PDF file. Only 8-bit characters are supported in the PDF file name; Unicode or embedded null characters are not supported.

optlist An option list (see Section 3.1.4, »Option Lists«, page 44) specifying PDF open options according to Table 7.25.

reserved Reserved, must be 0.

Returns A document handle which can be used for processing individual pages of the document or for querying document properties. A return value of -1 (in PHP: 0) indicates that the PDF document couldn't be opened. An arbitrary number of PDF documents can be opened simultaneously. The return value can be used until the end of the enclosing document scope.

Details In order to get more detailed information about the nature of a PDF import-related problem (wrong PDF file name, unsupported format, bad PDF data, etc.), set the *pdiwarning* parameter to *true*.

Scope *object, document, page;* in *object* scope a PDI document handle can only be used to query information from a PDF document.

Params See Table 7.28 and Table 7.29.

Table 7.25 Options for PDF_open_pdi( )

key	*type*	*explanation*
password	*string*	*(Maximum string length: 32 characters) The master password required to open a protected PDF document.*
pdiwarning	*boolean*	*Specifies whether or not this function will throw an exception in case of an error. Default is the value of the pdiwarning parameter (see Table 7.29).*

int PDF_open_pdi_callback(PDF *p, void *opaque, size_t filesize,
size_t (*readproc)(void *opaque, void *buffer, size_t size),
int (*seekproc)(void *opaque, long offset),
const char *optlist)

Open an existing PDF document from a custom data source and prepare it for later use.

opaque A pointer to some user data that might be associated with the input PDF document. This pointer will be passed as the first parameter of the callback functions, and can be used in any way. PDI will not use the opaque pointer in any other way.

filesize The size of the complete PDF document in bytes.

readproc A callback function which copies *size* bytes to the memory pointed to by *buffer*. If the end of the document is reached it may copy less data than requested. The function must return the number of bytes copied.

seekproc A callback function which sets the current read position in the document. *offset* denotes the position from the beginning of the document (0 meaning the first byte). If successful, this function must return 0, otherwise -1.

optlist An option list (see Section 3.1.4, »Option Lists«, page 44) specifying PDF open options according to Table 7.25.

Returns A document handle which can be used for processing individual pages of the document or for querying document properties. A return value of -1 indicates that the PDF document couldn't be opened. An arbitrary number of PDF documents can be opened simultaneously. The return value can be used until the end of the enclosing document scope.

Details This is a specialized interface for applications which retrieve arbitrary chunks of PDF data from some data source instead of providing the PDF document in a disk file or in memory.

Scope *object, document, page;* in *object* scope a PDI document handle can only be used to query information from a PDF document.

Params See Table 7.28 and Table 7.29.

Bindings Only available in the C binding.

void PDF_close_pdi(PDF *p, int doc)

Close all open PDI page handles, and close the input PDF document.

doc A valid PDF document handle retrieved with *PDF_open_pdi*( )*.

Details This function closes a PDF import document, and releases all resources related to the document. All document pages which may be open are implicitly closed. The document handle must not be used after this call. A PDF document should not be closed if more pages are to be imported. Although you can open and close a PDF import document an arbitrary number of times, doing so may result in unnecessary large PDF output files.

Scope *object, document, page*

Params See Table 7.28 and Table 7.29.

int PDF_open_pdi_page(PDF *p, int doc, int pagenumber, const char* optlist)

Prepare a page for later use with *PDF_place_pdi_page( )*.

doc A valid PDF document handle retrieved with *PDF_open_pdi*( )*.

pagenumber The number of the page to be opened. The first page has page number 1.

optlist An option list (see Section 3.1.4, »Option Lists«, page 44) specifying page options according to Table 7.26.

Returns A page handle which can be used for placing pages with *PDF_fit_pdi_page( )*. A return value of -1 (in PHP: 0) indicates that the page couldn't be opened. The return value can be used until the end of the enclosing document scope.

Details In order to get more detailed information about a problem related to PDF import (unsupported format, bad PDF data, etc.), set the *pdiwarning* parameter to *true*.

An arbitrary number of pages can be opened simultaneously. If the same page is opened multiply, different handles will be returned, and each handle must be closed exactly once. Opening the same page more than once is not recommended because the actual page data will be copied to the output document more than once.

Scope *document, page*

Params See Table 7.28 and Table 7.29.

Table 7.26 Options for PDF_open_pdi_page( )

key	type	explanation
pdiusebox	*keyword*	*Specifies which box dimensions to use for importing the page (see Section 5.2.2, »Using PDI Functions with PDFlib«, page 118). Default is the value of the pdiusebox parameter (see Table 7.29).*
pdiwarning	*boolean*	*Specifies whether or not this function will throw an exception in case of an error. Default is the value of the pdiwarning parameter (see Table 7.29).*

void PDF_close_pdi_page(PDF *p, int page)

Close the page handle, and free all page-related resources.

page A valid PDF page handle (not a page number!) retrieved with *PDF_open_pdi_page( )*.

Details This function closes the page associated with the page handle identified by *page*, and releases all related resources. *page* must not be used after this call.

Scope *document, page*

Params See Table 7.28 and Table 7.29.

void PDF_fit_pdi_page(PDF *p, int page, float x, float y, const char *optlist)

Place an imported PDF page at *(x, y)* with various options.

page A valid PDF page handle (not a page number!) retrieved with *PDF_open_pdi_page( )*. The page handle must not have been closed.

x, y The coordinates of the reference point in the user coordinate system where the page will be located, subject to various options.

optlist An option list (see Section 3.1.4, »Option Lists«, page 44) specifying placement details according to Table 7.24.

Details This function is similar to *PDF_fit_image( )*, but operates on imported PDF pages instead. Most scaling and placement options discussed in Table 7.24 are supported for PDF pages, too.

Scope *page, pattern, template, glyph*

Params See Table 7.28 and Table 7.29.

PDF/X The document from which the page is imported must conform to the same PDF/X conformance level and must use the same output intent as the generated document.

void PDF_place_pdi_page(PDF *p, int page, float x, float y, float sx, float sy)

Deprecated, use *PDF_fit_pdi_page( )* instead.

7.7.2 Other PDI Processing

int PDF_process_pdi(PDF *p, int doc, int page, const char* optlist)

Process certain elements of an imported PDF document.

doc A valid PDF document handle retrieved with *PDF_open_pdi*( )*.

page If *optlist* requires a page handle (see Table 7.27), *page* must be a valid PDF page handle (not a page number!) retrieved with *PDF_open_pdi_page( )*. The page handle must not have been closed. If *optlist* does not require any page handle, *page* must be -1.

optlist An option list (see Section 3.1.4, »Option Lists«, page 44) specifying processing options according to Table 7.27.

Returns The value 1 if the function succeeded, or an error code of -1 (in PHP: 0) if the function call failed.

Details The details of this function are determined by the options provided in *optlist*.

Scope *document*

Params See Table 7.29.

PDF/X The output intent for the generated document must be set either using this function with the *copyoutputintent* option, or by calling *PDF_load_profile( )*.

Table 7.27 Options for PDF_process_pdi( )

key	*type*	*explanation*
action[1]	*keyword*	*Specifies the kind of PDF processing:* *copyoutputintent: copy the PDF/X output intent of the imported document to the output document. The second and subsequent attempts to copy an output intent will be ignored.*
pdiwarning[1]	*boolean*	*Specifies whether or not this function will throw an exception in case of an error. Default is the value of the pdiwarning parameter (see Table 7.29).*

1. Does not require a page handle.

7.7.3 Parameter Handling

float PDF_get_pdi_value(PDF *p, const char *key, int doc, int page, int reserved)

Get some PDI document parameter with numerical type.

key Specifies the name of the parameter to be retrieved, see Table 7.28 and Table 7.29.

doc A valid PDF document handle retrieved with *PDF_open_pdi( )*.

page A valid PDF page handle (not a page number!) retrieved with *PDF_open_pdi_page( )*. For keys which are not page-related *page* must be -1 (in PHP: 0).

reserved Currently unused, must be 0.

Returns The numerical value retrieved from the document.

Scope *any*

Table 7.28 Page-related values for PDF import

function	*key*	*explanation*
get_pdi_value	*width* *height*	*Get the width or height, respectively, of an imported page in default units. Cropping and rotation will be taken into account.*
get_pdi_value	*/Rotate*	*page rotation in degrees (0, 90, 180, or 270)*
get_pdi_value	*/CropBox,* */BleedBox,* */ArtBox,* */TrimBox*	*Query one of the box parameters of the page. The parameter name must be followed by a slash '/' character and one of llx, lly, urx, ury, for example: /CropBox/llx (see Section 3.2.2, »Page Sizes and Coordinate Limits«, page 55 for details). Note that these will not have the /Rotate key applied, unlike the width and height values which already reflect any rotation which may be applied to the page.*

const char * PDF_get_pdi_parameter(
PDF *p, const char *key, int doc, int page, int reserved, int *len)

Get some PDI document parameter with string type.

key Specifies the name of the parameter to be retrieved, see Table 7.28 and Table 7.29.

doc A valid PDF document handle retrieved with *PDF_open_pdi( )*.

page A valid PDF page handle (not a page number!) retrieved with *PDF_open_pdi_page( )*. For keys which are not page-related *page* must be -1 (in PHP: 0).

reserved Currently unused, must be 0.

len A C-style pointer to an integer which will receive the length of the returned string in bytes (for Unicode strings: including the BOM, but excluding the terminating double-null). This parameter is only required for C and C++, and not allowed in other language bindings.

Returns The string parameter retrieved from the document. Unicode info strings *(/Info/<key>)* will be returned with initial BOM and terminating double-null. Currently PDFlib does not construct a proper Unicode string object from document info keys containing Unicode text. If no information is available an empty string will be returned.

The contents of the string will be valid until the next call of this function, or the end of the surrounding *object* scope (whatever happens first).

Details This function gets some string parameter related to an imported PDF document, in some cases further specified by *page* and *index*. Table 7.29 lists relevant parameter combinations.

Bindings C and C++: The *len* parameter must be supplied.

Other bindings: The *len* parameter must be omitted; instead, a string object of appropriate length will be returned.

Scope *any*

Table 7.29 Document-related parameters and values for PDF import

function	key	explanation
get_parameter	pdi[1]	Returns the string true if the PDI library is attached (and not restricted to demo mode), and false otherwise. Scope: any, null[2].
get_pdi_value	/Root/Pages/Count[1]	total number of pages in the imported document
get_pdi_parameter	filename[1]	name of the imported PDF file; if the file has been opened with PDF_open_pdi_callback( ) a dummy name will be returned.
get_pdi_parameter	/Info/<key>[1]	Retrieves the string value of a key in the document info dictionary, e.g., /Info/Title. No conversion will be applied to the string. If the key cannot be found in the document an empty string will be returned. However, if pdiwarning is set to true, an exception will be thrown for a key that couldn't be found.
get_pdi_parameter	pdfx[1]	Retrieves the PDF/X conformance level of the imported document. The result is one of »PDF/X-1:2001« , »PDF/X-1a:2001«, »PDF/X-3:2002«, »none«, or a string designating a later PDF/X conformance level (see Section 3.4, »PDF/X Support«, page 67).
get_pdi_value	version[1]	PDF version number multiplied by 10, e.g. 14 for PDF 1.4
set_parameter	pdiwarning[1]	This parameter can be used to obtain more detailed information about why a PDF or page couldn't be opened. Default: false true Raise a nonfatal exception when the PDI function fails. The information string supplied with the exception may be useful in debugging import-related problems. false Do not raise an exception when the PDI function fails. Instead, the function returns -1 (in PHP: 0) on error.
set_parameter	pdiusebox[1]	This parameter determines which of the Box entries of a page will be used for determining an imported page's size. Default: crop media Use the MediaBox (which is always present) crop Use the CropBox if present, else the MediaBox bleed Use the BleedBox if present, else the CropBox trim Use the TrimBox if present, else the CropBox art Use the ArtBox if present, else the CropBox The pdiusebox parameter must be set before calling PDF_open_pdi_page( ).
get_pdi_parameter get_pdi_value	vdp/Blocks/<block>/<property> or vdp/Blocks/<block>/Custom/<property>	Query standard and custom block properties (see Section 6.4, »Querying Block Names and Properties«, page 141). Only available in the PDFlib Personalization Server (PPS).
get_pdi_value	vdp/blockcount	Query the total number of blocks on the page.

1. The page parameter must be -1 (in PHP: 0).
*2. May be called with a PDF * argument of NULL or 0.*

7.8 Block Filling Functions (PPS)

The PDFlib Personalization Server (PPS) offers dedicated functions for processing variable data blocks of type *Text, Image,* and *PDF.* These blocks must be contained in the imported PDF page, but will not be retained in the generated output. The imported page must have been placed on the output page before using any of the block filling functions. When calculating the block position on the page, the block functions will take into account the scaling options provided to the most recent call to *PDF_fit_pdi_page( )* with the respective PDF page handle.

If only block processing is desired without actually placing the contents of the page on the output (i.e., the imported page is only used as a container of blocks) the *blind* option of *PDF_fit_pdi_page( )* can be used. This is useful if you want to place blocks below the contents of the original page. To achieve this, use *PDF_fit_pdi_page( )* with the *blind* option, fill the blocks as desired, and call *PDF_fit_pdi_page( )* again, this time without the *blind* option.

Note *The block processing functions discussed in this section require the PDFlib Personalization Server (PPS). The PDFlib Block plugin for Adobe Acrobat is required for creating blocks in PDF templates. See Chapter 6 for more information about the PDFlib Block plugin.*

int PDF_fill_textblock(PDF *p,
int page, const char *blockname, const char *text, int len, const char *optlist)

Fill a text block with variable data according to its properties.

page A valid PDF page handle for a page containing blocks.

blockname The name of the block.

text The text to be filled into the block, or an empty string if the default text is to be used.

len (C binding only) Length of *text* (in bytes) for strings which may contain null characters. If *len = 0* a null-terminated string must be provided.

optlist An option list (see Section 3.1.4, »Option Lists«, page 44) specifying filling details according to Table 7.30.

Returns -1 (in PHP: 0) if the named block doesn't exist on the page, the block cannot be filled (e.g., due to font problems), or the block requires a newer PDFlib version for processing; 1 if the block could be processed successfully. Use the *pdiwarning* option to get more information about the nature of the problem.

Details The supplied text will be formatted into the block, subject to the block's properties. If *text* is empty the function will use the block's default text if available, and silently return otherwise. This may be useful to take advantage of other block properties, such as fill or stroke color.

If the PDF document is found to be corrupt, this function will either throw an exception or return -1 subject to the *pdiwarning* parameter or option.

Scope *page, template*

Note *This function is only available in the PDFlib Personalization Server (PPS).*

int PDF_fill_imageblock(PDF *p,
int page, const char *blockname, int image, const char *optlist)

Fill an image block with variable data according to its properties.

page A valid PDF page handle for a page containing blocks.

blockname The name of the block.

image A valid image handle for the image to be filled into the block, or -1 if the default image is to be used.

optlist An option list (see Section 3.1.4, »Option Lists«, page 44) specifying filling details according to Table 7.30.

Returns -1 (in PHP: 0) if the named block doesn't exist on the page, the block cannot be filled, or the block requires a newer PDFlib version for processing; 1 if the block could be processed successfully. Use the *pdiwarning* option to get more information about the nature of the problem.

Details The image referred to by the supplied image handle will be placed in the block, subject to the block's properties. If *image* is -1 (in PHP: 0) the function will use the block's default image if available, and silently return otherwise.

If the PDF document is found to be corrupt, this function will either throw an exception or return -1 subject to the *pdiwarning* parameter or option.

Scope *page, template*

Note *This function is only available in the PDFlib Personalization Server (PPS).*

int PDF_fill_pdfblock(PDF *p,
int page, const char *blockname, int contents, const char *optlist)

Fill a PDF block with variable data according to its properties.

page A valid PDF page handle for a page containing blocks.

blockname The name of the block.

contents A valid PDF page handle for the PDF page to be filled into the block, or -1 if the default PDF page is to be used.

optlist An option list (see Section 3.1.4, »Option Lists«, page 44) specifying filling details according to Table 7.30.

Returns -1 (in PHP: 0) if the named block doesn't exist on the page, the block cannot be filled, or the block requires a newer PDFlib version for processing; 1 if the block could be processed successfully. Use the *pdiwarning* option to get more information about the nature of the problem.

Details The PDF page referred to by the supplied page handle *contents* will be placed in the block, subject to the block's properties. If *contents* is -1 (in PHP: 0) the function will use the block's default PDF page if available, and silently return otherwise.

If the PDF document is found to be corrupt, this function will either throw an exception or return -1 subject to the *pdiwarning* parameter or option.

Scope *page, template*

Note *This function is only available in the PDFlib Personalization Server (PPS).*

Table 7.30 Options for the PDF_fill_*block( ) functions

key	type	explanation
embedding	*boolean*	*(Only for PDF_fill_textblock( )) The embedding option for the font. Default: false.*
encoding	*string*	*(Required for PDF_fill_textblock( ) unless no text is supplied and the defaulttext property is used) Encoding for the font as required by PDF_load_font( ).*
fontwarning	*boolean*	*(Only for PDF_fill_textblock( )) Specifies whether or not this function will throw an exception in case of font-related problems. Default is the value of the pdiwarning option.*
imagewarning	*boolean*	*(Only for PDF_fill_imageblock( )) Specifies whether or not this function will throw an exception in case of image-related problems. Default is the value of the pdiwarning option.*
pdiwarning	*boolean*	*Specifies whether or not this function will throw an exception in case of an error in the PDF page containing the block or the page to be used as block contents. Default is the value of the pdiwarning parameter (see Table 7.29).*
textformat	*string*	*(Only for PDF_fill_textblock( ) unless the defaulttext property is used) The format used to interpret the supplied text (see Section 4.5.2, »Unicode Text Formats«, page 92). Default: auto*
almost any property name	*various*	*Block property names and values (see Section 6.3, »Standard Properties for automated Processing«, page 137) which will be used to override those in the block definition. See Section 6.1.2, »Block Properties«, page 128, for details. The following block properties can not be overridden:* *Name, Description, Locked, Subtype, Type* *defaulttext, defaultimage, defaultpdf, defaultpdfpage* *As an alternative to supplying the fontname property the font option can be used to supply a font handle (fontname will be ignored in this case).* *Color properties support the following color space keywords: none, gray, rgb, cmyk, spot, spotname.*

7.9 Hypertext Functions

Strings for hypertext functions may contain 8-bit-encoded text or Unicode. The string will be converted according to the *hypertextencoding* parameter unless it contains Unicode.

Table 7.31 lists relevant parameters and values for this section.

Table 7.31 Parameters for hypertext functions (see Section 7.2.3, »Parameter Handling«, page 153)

function	key	explanation
set_parameter get_parameter	hypertextencoding	Specifies the encoding in which hypertext functions will expect the client-supplied strings (see Section 4.5.3, »Unicode for Hypertext Elements«, page 93). An empty string is equivalent to unicode. Default: empty string for Unicode-capable language bindings, otherwise auto. Scope: any.
set_parameter get_parameter	hypertextformat	Set the format in which the hypertext functions will expect the client-supplied strings. Possible values are bytes, utf8, utf16, utf16le, utf16be, and auto . Default: auto. Scope: any.
set_parameter	usercoordinates	If false, coordinates for hypertext rectangles will be expected in the default coordinate system (see Section 3.2.1, »Coordinate Systems«, page 53); otherwise the current user coordinate system will be used. Default: false. Scope: any.

7.9.1 Document Open Action and Open Mode

Table 7.32 lists relevant parameters and values for this section.

Table 7.32 Parameters for document open action and open mode (see Section 7.2.3, »Parameter Handling«, page 153). Scope: document.

function	key	explanation
set_parameter	openaction	Set the open action, i.e., the page number and zoom factor which will be visible upon opening the document. The value is an option list according to Table 7.40. Default: »type fitwindow«.
set_parameter	openmode	Set the appearance when the document is opened.Default: bookmarks if the document contains any bookmarks, and otherwise none. none — Neither bookmarks nor thumbnails are visible bookmarks — Open the document with bookmarks visible. thumbnails — Open document with thumbnails visible fullscreen — Open in fullscreen mode (does not work in the browser).

7.9.2 Viewer Preferences

Table 7.33 lists relevant parameters and values for this section.

Table 7.33 Parameters for viewer preferences (see Section 7.2.3, »Parameter Handling«, page 153). Scope: document

function	key	explanation
set_parameter	hidetoolbar[1]	Boolean specifying whether to hide Acrobat's tool bar. Default: false.
set_parameter	hidemenubar	Boolean specifying whether to hide Acrobat's menu bar. Default: false.
set_parameter	hidewindowui	Boolean specifying whether to hide Acrobat's window controls. Default: false.
set_parameter	fitwindow	Boolean specifying whether to resize the document's window to the size of the first page. Default false.

Table 7.33 Parameters for viewer preferences (see Section 7.2.3, »Parameter Handling«, page 153). Scope: document

function	key	explanation
set_parameter	centerwindow	Boolean specifying whether to position the document's window in the center of the screen. Default: false.
set_parameter	displaydoctitle	Boolean specifying whether to display the Title document info field in Acrobat's title bar (true) or the file name (false). Default: false.
set_parameter	nonfullscreen-pagemode	Specifies how to display the document on exiting full-screen mode (only relevant if the openmode parameter is set to fullscreen). Default: usenone. useoutlines display page and document outline (bookmarks) usethumbs display page and thumbnails usenone neither document outline nor thumbnails, only page
set_parameter	direction	The reading order of the document. Default l2r. l2r Left to right r2l Right to left (including vertical writing systems) This parameter affects the scroll ordering in double-page view.
set_parameter	viewarea viewclip printarea printclip	The value of the page boundary box representing the area of a page to be displayed or clipped when viewing the document on screen or printing it. Acrobat ignores this setting. Default crop: art Use the ArtBox bleed Use the BleedBox crop Use the CropBox media Use the MediaBox trim Use the TrimBox

1. Acrobat ignores this setting when viewing PDFs in a browser.

7.9.3 Bookmarks

Table 7.34 lists relevant parameters for this section.

Table 7.34 Parameters for bookmarks (see Section 7.2.3, »Parameter Handling«, page 153)

function	key	explanation
set_parameter	bookmark-dest	Set the the page number and zoom factor for subsequently generated bookmarks. The value is an option list according to Table 7.40. This parameter can be changed an arbitrary number of times. Default: »type fitwindow«.

Note Adding bookmarks sets the open mode (see Section 7.9.1, »Document Open Action and Open Mode«, page 204) to bookmarks unless another mode has explicitly been set.

int PDF_add_bookmark(PDF *p, const char *text, int parent, int open)
int PDF_add_bookmark2(PDF *p, const char *text, int len, int parent, int open)

Add a nested bookmark under *parent*, or a new top-level bookmark.

text Contains the text of the bookmark. It may contain Unicode. The maximum length of *text* is 255 single-byte characters (8-bit encodings), or 126 Unicode characters. However, a practical limit of 32 characters for *text* is advised.

len (Only for *PDF_add_bookmark2()*, and only for the C binding.) Length of *text* (in bytes) for strings which may contain null characters. If *len* = *0* a null-terminated string must be provided.

parent If *parent* contains a valid bookmark handle returned by a previous call to *PDF_add_bookmark( )*, a new bookmark will be generated which is a subordinate of the given parent. In this way, arbitrarily nested bookmarks can be generated. If *parent = 0* a new top-level bookmark will be generated.

open If 0, child bookmarks will not be visible. If *open = 1*, all children will be folded out.

Returns An identifier for the bookmark just generated. This identifier may be used as the *parent* parameter in subsequent calls.

Details This function adds a PDF bookmark with the supplied *text*. The bookmark appearance (font style and color) and target can be controlled with the *bookmarkdest* parameter (see Table 7.34). If the *page* option in the *bookmarkdest* parameter has been set to 0 the bookmark will point to the current page.

Scope *page*

Params *openmode, bookmarkdest*

7.9.4 Document Information Fields

void PDF_set_info(PDF *p, const char *key, const char *value)
void PDF_set_info2(PDF *p, const char *key, const char *value, int len)

Fill document information field *key* with *value*.

key The name of the document info field, which may be any of the standard names, or an arbitrary custom name (see Table 7.35). There is no limit for the number of custom fields. Regarding the use and semantics of custom document information fields, PDFlib users are encouraged to take a look at the Dublin Core Metadata element set.[1]

value The string to which the *key* parameter will be set. It may contain Unicode. Acrobat imposes a maximum length of *value* of 255 bytes. Note that due to a bug in Adobe Reader 6 for Windows the & character does not display properly in some info strings.

len (Only for *PDF_set_info2( )*, and only for the C binding.) Length of *value* (in bytes) for strings which may contain null characters. If *len = 0* a null-terminated string must be provided.

Scope *object, document, page*

Table 7.35 Values for the document information field key

key	*explanation*
Subject	*Subject of the document*
Title	*Title of the document*
Creator	*Software used to create the document (as opposed to the Producer of the PDF output, which is always PDFlib)*
Author	*Author of the document*
Keywords	*Keywords describing the contents of the document*

1. See http://dublincore.org

Table 7.35 Values for the document information field key

key	*explanation*
Trapped	*Indicates whether trapping has been applied to the document. Allowed values are True, False, and Unknown.*
any name other than CreationDate, Producer, and ModDate	*User-defined field. PDFlib supports an arbitrary number of fields.*

7.9.5 Page Transitions

PDF files may specify a page transition in order to achieve special effects which may be useful for presentations or »slide shows«. In Acrobat, these effects cannot be set document-specific or on a page-by-page basis, but only for the full screen mode. PDFlib, however, allows setting the page transition mode and duration for each page separately. Table 7.36 lists relevant parameters and values for this section.

Table 7.36 Parameters and values for page transitions (see Section 7.2.3, »Parameter Handling«, page 153)

function	*key*	*explanation*
set_parameter	*transition*	*Set the page transition effect for the current and subsequent pages until the transition is changed again. The transition types below are supported. type may also be empty to reset the transition effect. Default is replace. Scope: any.* *split* Two lines sweeping across the screen reveal the page *blinds* Multiple lines sweeping across the screen reveal the page *box* A box reveals the page *wipe* A single line sweeping across the screen reveals the page *dissolve* The old page dissolves to reveal the page *glitter* The dissolve effect moves from one screen edge to another *replace* The old page is simply replaced by the new page (default)
set_value	*duration*	*Set the page display duration in seconds for the current page. Default is one second. Scope: any*

7.9.6 File Attachments

void PDF_attach_file(PDF *p, float llx, float lly, float urx, float ury, const char *filename, const char *description, const char *author, const char *mimetype, const char *icon)
void PDF_attach_file2(PDF *p, float llx, float lly, float urx, float ury, const char *filename, int reserved, const char *description, int desc_len, const char *author, int author_len, const char *mimetype, const char *icon)

Add a file attachment annotation.

llx, lly, urx, ury *x* and *y* coordinates of the lower left and upper right corners of the annotation rectangle in default coordinates (if the *usercoordinates* parameter is *false)* or user coordinates (if it is *true)*. Acrobat will align the upper left corner of the icon at the upper left corner of the specified rectangle.

filename The name of the file which will be attached to the PDF document. If the file cannot be opened PDFlib will throw an exception.

reserved (C language binding only.) Reserved, must be 0.

description A string with some explanation of the attachment. It may contain Unicode.

desc_len (Only for *PDF_attach_file2( )*, and only for the C binding.) Length of *description* (in bytes) for strings which may contain null characters. If *len* = *0* a null-terminated string must be provided.

author A string with the author's name or function. It may contain Unicode.

author_len (Only for *PDF_attach_file2( )*, and only for the C binding.) Length of *author* (in bytes) for strings which may contain null characters. If *len* = *0* a null-terminated string must be provided.

mimetype The MIME type of the file. It will be used by Acrobat for launching the appropriate program when the file attachment annotation is activated.

icon Controls the display of the unopened file attachment in Acrobat (see Table 7.37).

Details This function adds a file attachment annotation at the specified rectangle. Acrobat Reader is unable to deal with file attachments and will display a question mark instead. File attachments only work in the full Acrobat software. The color of the attachment icon can be controlled with *PDF_set_border_color( )*.

Scope *page*

Table 7.37 Icon names for file attachments

icon name	*icon appearance*	*icon name*	*icon appearance*
graph		*pushpin*	
paperclip		*tag*	

7.9.7 Note Annotations

Note *All annotation coordinates are different from the parameters of the PDF_rect( ) function. While all annotation functions expect parameters for two corners directly, PDF_rect( ) expects the coordinates of one corner, plus width and height values.*

---

void PDF_add_note(PDF *p, float llx, float lly, float urx, float ury,
const char *contents, const char *title, const char *icon, int open)
void PDF_add_note2(PDF *p, float llx, float lly, float urx, float ury,
const char *contents, int contents_len, const char *title, const char *icon, int open)

Add a note annotation.

llx, lly, urx, ury *x* and *y* coordinates of the lower left and upper right corners of the note rectangle in default coordinates (if the *usercoordinates* parameter is *false)* or user coordinates (if it is *true)*. Acrobat will align the upper left corner of the icon at the upper left corner of the specified rectangle.

contents Text content of the note. It may contain Unicode. The maximum length of *contents* is 65535 bytes.

contents_len (Only for *PDF_add_note2( )*, and only for the C binding.) Length of *contents* (in bytes) for strings which may contain null characters. If *len* = *0* a null-terminated string must be provided.

title Heading text of the note. It may contain Unicode. The maximum length of *title* is 255 single-byte characters or 126 Unicode characters. However, a practical limit of 32 characters for *title* is advised.

title_len (Only for *PDF_add_note2( )*, and only for the C binding.) Length of *title* (in bytes) for strings which may contain null characters. If *len* = *0* a null-terminated string must be provided.

icon Controls the display of the unopened note annotation in Acrobat (see Table 7.38).

open The annotation will be displayed in open state if *open* = *1*, and closed if *open* = *0*.

Details This function adds a note annotation at the specified rectangle. The color of the note icon can be controlled with *PDF_set_border_color( )*.

Scope *page*

Table 7.38 Icon names for note annotations

icon name	*icon appearance*	*icon name*	*icon appearance*
comment		*newparagraph*	
insert		*key*	
note		*help*	
paragraph			

7.9.8 Link Annotations and Named Destinations

Table 7.39 lists relevant parameters for this section.

Note *PDF doesn't support links with shapes other than rectangles.*

Table 7.39 Parameters for links (see Section 7.2.3, »Parameter Handling«, page 153)

function	*key*	*explanation*
set_parameter	*base*	*Set the document's base URL. This is useful when a document with relative Web links to other documents is moved to a different location. Setting the base URL to the »old« location makes sure that relative links will still work. Scope: page, pattern, template, document.*
set_parameter	*launchlink: parameters*	*Set additional parameters which will be passed to an application launched via PDF_add_launchlink( ). This is only supported by Acrobat on Windows. Multiple parameters can be separated with a space character, but individual parameters must not contain any space characters. Scope: any*[1]

Table 7.39 Parameters for links (see Section 7.2.3, »Parameter Handling«, page 153)

function	key	explanation
set_parameter	launchlink: operation	Specify an operation which will be applied to a document launched via PDF_add_launchlink( ). This must be either »open« or »print«. In the latter case the launched file must be a document (not an application). It is only supported by Acrobat on Windows. Scope: any
set_parameter	launchlink: defaultdir	Set an additional default directory for an application launched via PDF_add_launchlink( ). This is only supported by Acrobat on Windows. Scope: any

1. The next call to PDF_add_launchlink( ) will use any of the launchlink parameters which have been set, and reset it after use. For subsequent function calls the parameters must be set again.

void PDF_add_pdflink(PDF *p, float llx, float lly, float urx, float ury, const char *filename, int page, const char *optlist)

Add a file link annotation (to a PDF target).

llx, lly, urx, ury *x* and *y* coordinates of the lower left and upper right corners of the link rectangle in default coordinates (if the *usercoordinates* parameter is *false)* or user coordinates (if it is *true)*.

filename The name of the target PDF file.

page The physical page number of the target page, which must be greater than 0. This parameter will be ignored if the *page* option is present in *optlist*. Note that due to a bug Acrobat 6.0 will ignore the page number, and will always jump to page 1. This bug has been fixed in Acrobat 6.0.1, and is not present in older versions.

optlist An option list (see Section 3.1.4, »Option Lists«, page 44) specifying the destination according to Table 7.40.

Scope *page*

void PDF_add_locallink(PDF *p, float llx, float lly, float urx, float ury, int page, const char *optlist)

Add a link annotation to a target within the current PDF file.

llx, lly, urx, ury *x* and *y* coordinates of the lower left and upper right corners of the link rectangle in default coordinates (if the *usercoordinates* parameter is *false)* or user coordinates (if it is *true)*.

page The physical page number of the target page, which must be greater than 0. This may be a previously generated page, or a page in the same document that will be generated later (after the current page). However, the application must make sure that the target page will actually be generated; PDFlib will issue a warning message otherwise. The value 0 can be used for the current page. This parameter will be ignored if the *page* option is present in *optlist*.

optlist An option list (see Section 3.1.4, »Option Lists«, page 44) specifying the destination according to Table 7.40.

Scope *page*

Table 7.40 Options to specify destinations for use with PDF_add_pdflink( ), PDF_add_locallink( ), and PDF_add_nameddest( ). The same options are also used for the openaction and bookmarkdest parameters.

option	type	explanation
type	keyword	Specifies the location of the window on the target page: file (Only for bookmarkdest) Open an external (PDF or other) file specified by the filename option. fixed Use a fixed destination view specified by the left, top, and zoom options. If any of these options is missing its current value will be retained. fitwindow Fit the complete page to the window. fitwidth Fit the page width to the window, with the y coordinate top at the top edge of the window. fitheight Fit the page height to the window, with the x coordinate left at the left edge of the window. fitrect Fit the rectangle specified by left, bottom, right, and top to the window. fitvisible Fit the visible contents of the page (the ArtBox) to the window. fitvisiblewidth Fit the visible contents of the page to the window with the y coordinate top at the top edge of the window fitvisibleheight Fit the visible contents of the page to the window with the x coordinate left at the left edge of the window. nameddest (Not for PDF_add_nameddest( )) A named destination specified with the name option. Default: fitwindow
color	list of three float values	(Only for bookmarkdest) Three float values for the red, green, and blue color components of the bookmark text. Default: (0, 0, 0) = black
filename	string	(Only for bookmarkdest and type = file) The name of an external (PDF or other) file which will be opened when the bookmark is activated.
fontstyle	keyword	(Only for bookmarkdest) A keyword which specifies the font style of the bookmark text: normal, bold, italic, bolditalic. Default: normal
name	string	(Not for PDF_add_nameddest( ); required if type = nameddest, and ignored otherwise). String designating a named destination which must be defined in the target file. If this option is provided no other option except type, color, and fontstyle must be used. Destination names must be unique within a document.
page	integer	The page number of the destination page (first page is 1). The page must exist in the destination PDF. Page 0 means the current page for bookmarkdest, PDF_add_nameddest( ), PDF_add_locallink( ) and PDF_add_bookmark( ). Default: 1 for openaction; 0 for bookmarkdest, PDF_add_nameddest( ), and PDF_add_locallink( ); the page parameter of PDF_add_pdflink( ) and PDF_add_locallink( ).
zoom	float	(Only for type = fixed) The zoom factor (1 means 100%) to be used to display the page contents. If this option is missing or 0 the zoom factor which was in effect when the link was activated will be retained.
left	float	(Only for type = fixed, fitheight, fitrect, or fitvisibleheight) The x coordinate of the page which will positioned at the left edge of the window. Default: 0
right	float	(Only for type = fitrect) The x coordinate of the page which will positioned at the right edge of the window. Default: 1000
bottom	float	(Only for type = fitrect) The y coordinate of the page which will positioned at the bottom edge of the window. Default: 0
top	float	(Only for type = fixed, fitwidth, fitrect, or fitvisiblewidth) The y coordinate of the page which will positioned at the top edge of the window. Default: 1000
fitbbox	boolean	(Deprecated) true is equivalent to »type fitvisible«
fitheight	boolean	(Deprecated) true is equivalent to »type fitheight left 0«
fitpage	boolean	(Deprecated) true is equivalent to »type fitwindow«

Table 7.40 Options to specify destinations for use with PDF_add_pdflink( ), PDF_add_locallink( ), and PDF_add_nameddest( ). The same options are also used for the openaction and bookmarkdest parameters.

option	type	explanation
fitwidth	boolean	(Deprecated) true is equivalent to »type fitwidth top 10000«
retain	boolean	(Deprecated) true is equivalent to »type fixed« (left, top, zoom will be retained)

void PDF_add_launchlink(PDF *p, float llx, float lly, float urx, float ury, const char *filename)

Add a launch annotation (to a target of arbitrary file type).

llx, lly, urx, ury *x* and *y* coordinates of the lower left and upper right corners of the link rectangle in default coordinates (if the *usercoordinates* parameter is *false)* or user coordinates (if it is *true)*.

filename The name of the file which will be launched upon clicking the link.

Scope *page*

Params *launchlink:parameters, launchlink:operation, launchlink:defaultdir.* These parameters will be reset to empty values after each call to this function.

void PDF_add_weblink(PDF *p, float llx, float lly, float urx, float ury, const char *url)

Add a weblink annotation to a target URL on the Web.

llx, lly, urx, ury *x* and *y* coordinates of the lower left and upper right corners of the link rectangle in default coordinates (if the *usercoordinates* parameter is *false)* or user coordinates (if it is *true)*.

url A Uniform Resource Identifier encoded in 7-bit ASCII specifying the link target. It can point to an arbitrary (Web or local) resource.

Scope *page*

Params The *textx/texty, currentx/currenty*, and *imagewidth/imageheight* parameters may be useful for retrieving positioning information for calculating the dimension of link rectangles.

void PDF_set_border_style(PDF *p, const char *style, float width)

Set the border style for all kinds of links.

style Specifies the link's border style, and must be one of *solid* or *dashed*.

width Specifies the link's border width in points. If *width = 0* the borders will be invisible.

Details The settings made by this function are used for all links until a new style is set. At the beginning of a document the links border style is set to a default of a solid line with a width of 1.

Scope *document, page*

void PDF_set_border_color(PDF *p, float red, float green, float blue)

Set the border color for links, notes, and file attachments (annotations).

red, green, blue The RGB color values for annotation borders (in the range 0..1).The settings made by this function are used for all annotations until a new color is set. At the beginning of a document the annotation border color is set to black *(0, 0, 0)*.

Scope *document, page*

void PDF_set_border_dash(PDF *p, float b, float w)

Set the border dash style for all kinds of links.

b, w Specify the border dash style (see *PDF_setdash( )*).

Details At the beginning of a document the links border dash style is set to a default of *(3, 3)*. However, this default will only be used when the border style is explicitly set to *dashed*.

Scope *document, page*

void PDF_add_nameddest(PDF *p, const char *name, int reserved, const char *optlist)

Create a named destination on an arbitrary page in the current document.

name The name of the destination, which can be used as a target for links. Destination names must be unique within a document. Duplicate names will be silently ignored.

reserved (C language binding only.) Reserved, must be 0.

optlist An option list (see Section 3.1.4, »Option Lists«, page 44) specifying the destination according to Table 7.40. However, since the destination must be specified explicitly, the *nameddest* option is not allowed.

Details The destination details must be specified in *optlist*, and the destination may be located on any page in the current document. The provided *name* can be used as a target for all functions and parameters which accept destination optlists according to Table 7.40.

Scope *document, page*

7.9.9 Thumbnails

void PDF_add_thumbnail(PDF *p, int image)

Add an existing image as thumbnail for the current page.

image A valid image handle retrieved with *PDF_load_image( )*.

Details This function adds the supplied image as thumbnail image for the current page. A thumbnail image must adhere to the following restrictions:

- The image must be no larger than 106 x 106 pixels.
- The image must use the grayscale, RGB, or indexed RGB color space.
- Multi-strip TIFF images can not be used as thumbnails because thumbnails must be constructed from a single PDF image object, and multi-strip TIFF images result in

multiple PDF image objects (see Section 5.1.2, »Supported Image File Formats«, page 112).

This function doesn't generate thumbnail images for pages, but only offers a hook for adding existing images as thumbnails. The actual thumbnail images must be generated by the client or some other application. The client must ensure that color, height/width ratio, and actual contents of a thumbnail match the corresponding page contents.

Since Acrobat 5 generates thumbnails on the fly (though not in the Browser), and thumbnails increase the overall file size of the generated PDF, it is recommended not to add thumbnails, but rely on client-side thumbnail generation instead.

Scope *page;* must only be called once per page. Not all pages need to have thumbnails attached to them.

Params *openmode*

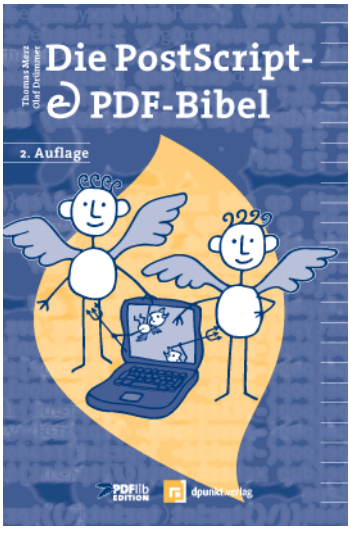
8 References

[1] Adobe Systems Incorporated: PDF Reference, Fourth Edition: Version 1.5. Available as PDF from *http://partners.adobe.com/asn/tech/pdf/specifications.jsp*

[2] Adobe Systems Incorporated: PostScript Language Reference Manual, Third Edition. Published by Addison-Wesley 1999, ISBN 0-201-37922-8; also available as PDF from *http://partners.adobe.com/asn/tech/ps/technotes.jsp*

[3] The following book by the principal author of PDFlib is currently only available in German. It discusses a variety of PostScript, PDF and font-related topics:

Thomas Merz, Olaf Drümmer: Die PostScript- & PDF-Bibel.
Zweite Auflage. ISBN 3-935320-01-9, PDFlib Edition 2002
PDFlib GmbH, 80331 München, Tal 40, fax +49 • 89 • 29 16 46 86
Available as PDF from *http://www.pdflib.com*
Order by e-mail via *books@pdflib.com*



A PDFlib Quick Reference

General Functions

Font Functions

Text Output Functions

Function prototype	page
void PDF_set_text_pos(PDF *p, float x, float y)	163
void PDF_show(PDF *p, const char *text)	165
void PDF_show_xy(PDF *p, const char *text, float x, float y)	165
void PDF_continue_text(PDF *p, const char *text)	166
void PDF_fit_textline(PDF *p, const char *text, int len, float x, float y, const char *optlist)	166
int PDF_show_boxed(PDF *p, const char *text, float x, float y, float width, float height, const char *mode, const char *feature)	168
float PDF_stringwidth(PDF *p, const char *text, int font, float fontsize)	169

Graphics Functions

Function prototype	page
void PDF_setdash(PDF *p, float b, float w)	171
void PDF_setdashpattern(PDF *p, const char *optlist)	171
void PDF_setflat(PDF *p, float flatness)	171
void PDF_setlinejoin(PDF *p, int linejoin)	172
void PDF_setlinecap(PDF *p, int linecap)	172
void PDF_setmiterlimit(PDF *p, float miter)	173
void PDF_setlinewidth(PDF *p, float width)	173
void PDF_initgraphics(PDF *p)	173
void PDF_save(PDF *p)	173
void PDF_restore(PDF *p)	174
void PDF_translate(PDF *p, float tx, float ty)	174
void PDF_scale(PDF *p, float sx, float sy)	174
void PDF_rotate(PDF *p, float phi)	175
void PDF_skew(PDF *p, float alpha, float beta)	175
void PDF_concat(PDF *p, float a, float b, float c, float d, float e, float f)	175
void PDF_setmatrix(PDF *p, float a, float b, float c, float d, float e, float f)	176
int PDF_create_gstate(PDF *p, const char *optlist)	176
void PDF_set_gstate(PDF *p, int gstate)	177
void PDF_moveto(PDF *p, float x, float y)	177
void PDF_lineto(PDF *p, float x, float y)	178
void PDF_curveto(PDF *p, float x1, float y1, float x2, float y2, float x3, float y3)	178
void PDF_circle(PDF *p, float x, float y, float r)	178
void PDF_arc(PDF *p, float x, float y, float r, float alpha, float beta)	178
void PDF_arcn(PDF *p, float x, float y, float r, float alpha, float beta)	179
void PDF_rect(PDF *p, float x, float y, float width, float height)	179
void PDF_closepath(PDF *p)	179
void PDF_stroke(PDF *p)	180
void PDF_closepath_stroke(PDF *p)	180
void PDF_fill(PDF *p)	180
void PDF_fill_stroke(PDF *p)	180
void PDF_closepath_fill_stroke(PDF *p)	181

Function prototype	page
void PDF_clip(PDF *p)	181
void PDF_endpath(PDF *p)	181

Color Functions

Function prototype	page
void PDF_setcolor(PDF *p, const char *fstype, const char *colorspace, float c1, float c2, float c3, float c4)	182
int PDF_makespotcolor(PDF *p, const char *spotname, int reserved)	183
int PDF_load_iccprofile(PDF *p, const char *profilename, int reserved, const char *optlist)	184
int PDF_begin_pattern(PDF *p, float width, float height, float xstep, float ystep, int painttype)	185
void PDF_end_pattern(PDF *p)	186
int PDF_shading_pattern(PDF *p, int shading, const char *optlist)	186
void PDF_shfill(PDF *p, int shading)	186
int PDF_shading(PDF *p, const char *shtype, float x0, float y0, float x1, float y1, float c1, float c2, float c3, float c4, const char *optlist)	187

Image Functions

Function prototype	page
int PDF_load_image(PDF *p, const char *imagetype, const char *filename, int reserved, const char *optlist)	188
void PDF_close_image(PDF *p, int image)	191
void PDF_fit_image(PDF *p, int im, float x, float y, const char *optlist)	191
int PDF_begin_template(PDF *p, float width, float height)	193
void PDF_end_template(PDF *p)	193

PDF Import (PDI) Functions

Function prototype	page
int PDF_open_pdi(PDF *p, const char *filename, const char *optlist, int reserved)	195
int PDF_open_pdi_callback(PDF *p, void *opaque, size_t filesize, size_t (*readproc)(void *opaque, void *buffer, size_t size), int (*seekproc)(void *opaque, long offset), const char *optlist)	195
void PDF_close_pdi(PDF *p, int doc)	196
int PDF_open_pdi_page(PDF *p, int doc, int pagenumber, const char* optlist)	196
void PDF_close_pdi_page(PDF *p, int page)	197
void PDF_fit_pdi_page(PDF *p, int page, float x, float y, const char *optlist)	197
int PDF_process_pdi(PDF *p, int doc, int page, const char* optlist)	198
float PDF_get_pdi_value(PDF *p, const char *key, int doc, int page, int reserved)	199
const char * PDF_get_pdi_parameter(PDF *p, const char *key, int doc, int page, int reserved, int *len)	199

Block Filling Functions

Function prototype	page
int PDF_fill_textblock(PDF *p, int page, const char *blockname, const char *text, int len, const char *optlist)	201
int PDF_fill_imageblock(PDF *p, int page, const char *blockname, int image, const char *optlist)	202
int PDF_fill_pdfblock(PDF *p, int page, const char *blockname, int contents, const char *optlist)	202

Hypertext Functions

Function prototype	*page*
*int PDF_add_bookmark(PDF *p, const char *text, int parent, int open)*	*205*
*void PDF_set_info(PDF *p, const char *key, const char *value)*	*206*
*void PDF_attach_file(PDF *p, float llx, float lly, float urx, float ury, const char *filename, const char *description, const char *author, const char *mimetype, const char *icon)*	*207*
*void PDF_add_note(PDF *p, float llx, float lly, float urx, float ury, const char *contents, const char *title, const char *icon, int open)*	*208*
*void PDF_add_pdflink(PDF *p, float llx, float lly, float urx, float ury, const char *filename, int page, const char *optlist)*	*210*
*void PDF_add_locallink(PDF *p, float llx, float lly, float urx, float ury, int page, const char *optlist)*	*210*
*void PDF_add_launchlink(PDF *p, float llx, float lly, float urx, float ury, const char *filename)*	*212*
*void PDF_add_weblink(PDF *p, float llx, float lly, float urx, float ury, const char *url)*	*212*
*void PDF_set_border_style(PDF *p, const char *style, float width)*	*212*
*void PDF_set_border_color(PDF *p, float red, float green, float blue)*	*213*
*void PDF_set_border_dash(PDF *p, float b, float w)*	*213*
*void PDF_add_nameddest(PDF *p, const char *name, int reserved, const char *optlist)*	*213*
*void PDF_add_thumbnail(PDF *p, int image)*	*213*

Parameters and Values

category	function	keys
setup	set_parameter	resourcefile, SearchPath, compatibility, pdfx, license, licensefile, warning, openwarning, asciifile, flush, trace, tracefile, tracemsg
	set_value	compress
versioning	get_value	major, minor, revision
	get_parameter	version
page	set_value	pagewidth, pageheight CropBox, BleedBox, ArtBox, TrimBox: these must be followed by a slash '/' character and one of llx, lly, urx, ury, for example: CropBox/llx
	get_value	pagewidth, pageheight
font	set_parameter	FontAFM, FontPFM, FontOutline, Encoding, fontwarning, kerning, autosubsetting, autocidfont, textformat, unicodemap
	get_parameter	fontname, fontencoding, fontstyle, textformat
	set_value	subsetlimit, subsetminsize
	get_value	ascender, capheight, descender, font, fontsize, fontmaxcode, monospace
text	set_value	leading, textrise, horizscaling, textrendering, charspacing, wordspacing
	get_value	leading, textrise, horizscaling, textrendering, charspacing, wordspacing, textx, texty
	set_parameter	underline, overline, strikeout, kerning, glyphwarning
	get_parameter	underline, overline, strikeout, fontstyle
graphics	set_parameter	fillrule, topdown
	get_parameter	scope
	get_value	currentx, currenty
color	set_parameter	iccwarning, honoriccprofile, ICCProfile, StandardOutputIntent, renderingintent, preserveoldpantonenames, spotcolorlookup
	set_value	defaultgray, defaultrgb, defaultcmyk, setcolor:iccprofilegray, setcolor:iccprofilergb, setcolor:iccprofilecmyk
	get_value	image:iccprofile, icccomponents
image	get_value	imagewidth, imageheight, resx, resy
	set_parameter	imagewarning
PDI	get_parameter	pdi
	set_parameter	pdiwarning, pdiusebox
	get_pdi_value	/Root/Pages/Count, /Rotate, version, width, height CropBox, BleedBox, ArtBox, TrimBox: these must be followed by a slash '/' character and one of llx, lly, urx, ury, for example: CropBox/llx
	get_pdi_parameter	filename, /Info/<key>, vdp/Blocks/<blockname>/<propertyname>, vdp/Blocks/<blockname>/Custom/<propertyname>
hypertext	set_parameter	openaction, openmode, bookmarkdest, transition, base, hypertextformat, hypertextencoding, usercoordinates, hidetoolbar, hidemenubar, hidewindowui, fitwindow, centerwindow, displaydoctitle, nonfullscreenpagemode, direction, viewarea, viewclip, printarea, printclip, launchlink:parameters, launchlink:operation, launchlink:defaultdir,
	get_parameter	hypertextformat
	set_value	duration
security	set_parameter	userpassword, masterpassword, permissions

B Revision History

Revision history of this manual

Date	Changes
January 21, 2004	▸ Minor additions and corrections for PDFlib 5.0.3
September 15, 2003	▸ Minor additions and corrections for PDFlib 5.0.2; added block specification
May 26, 2003	▸ Minor updates and corrections for PDFlib 5.0.1
March 26, 2003	▸ Major changes and rewrite for PDFlib 5.0.0
June 14, 2002	▸ Minor changes for PDFlib 4.0.3 and extensions for the .NET binding
January 26, 2002	▸ Minor changes for PDFlib 4.0.2 and extensions for the IBM eServer edition
May 17, 2001	▸ Minor changes for PDFlib 4.0.1
April 1, 2001	▸ Documents PDI and other features of PDFlib 4.0.0
February 5, 2001	▸ Documents the template and CMYK features in PDFlib 3.5.0
December 22, 2000	▸ ColdFusion documentation and additions for PDFlib 3.03; separate COM edition of the manual
August 8, 2000	▸ Delphi documentation and minor additions for PDFlib 3.02
July 1, 2000	▸ Additions and clarifications for PDFlib 3.01
Feb. 20, 2000	▸ Changes for PDFlib 3.0
Aug. 2, 1999	▸ Minor changes and additions for PDFlib 2.01
June 29, 1999	▸ Separate sections for the individual language bindings ▸ Extensions for PDFlib 2.0
Feb. 1, 1999	▸ Minor changes for PDFlib 1.0 (not publicly released)
Aug. 10, 1998	▸ Extensions for PDFlib 0.7 (only for a single customer)
July 8, 1998	▸ First attempt at describing PDFlib scripting support in PDFlib 0.6
Feb. 25, 1998	▸ Slightly expanded the manual to cover PDFlib 0.5
Sept. 22, 1997	▸ First public release of PDFlib 0.4 and this manual

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z