PDFlib GmbH München, Germany

Reference Manual

PDFlib®

A library for generating PDF on the fly

4.0.1

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

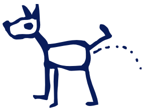
*Licensing contact: sales@pdflib.com
Support for commercial PDFlib licensees: support@pdflib.com (please include your license number)*

This publication and the information herein is furnished as is, is subject to change without notice, and should not be construed as a commitment by PDFlib GmbH. PDFlib GmbH assumes no responsibility or liability for any errors or inaccuracies, makes no warranty of any kind (express, implied or statutory) with respect to this publication, and expressly disclaims any and all warranties of merchantability, fitness for particular purposes and noninfringement of third party rights.

PDFlib and the PDFlib logo are trademarks of PDFlib GmbH. PDFlib licensees are granted the right to use the PDFlib name and logo in their product documentation. However, this is not required.

Adobe, Acrobat, and PostScript are trademarks of Adobe Systems Inc. AIX, IBM, OS/390, WebSphere, iSeries, and zSeries are trademarks or registered trademarks of International Business Machines Corporation. ActiveX, Microsoft, Windows, and Windows NT are trademarks of Microsoft Corporation. Apple, Macintosh and TrueType are trademarks of Apple Computer, Inc. Unicode and the Unicode logo are trademarks of Unicode, Inc. Unix is a trademark of The Open Group. Java and Solaris are a trademark of Sun Microsystems, Inc. Other company product and service names may be trademarks or service marks of others.

*PDFlib contains modified parts of the following third-party software:
PNG image reference library (libpng), Copyright © 1998, 1999, 2000 Glenn Randers-Pehrson
Zlib compression library, Copyright © 1995-1998 Jean-loup Gailly and Mark Adler
TIFFlib image library, Copyright © 1988-1997 Sam Leffler, Copyright © 1991-1997 Silicon Graphics, Inc.*



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

Contents

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 or maintaining the data to be processed, PDFlib takes over the task of generating the PDF code which graphically represents your data. While you must still format and arrange your text and graphical objects, PDFlib frees you from the internals and intricacies of PDF. PDFlib offers many useful functions for creating text, graphics, images and hypertext elements in PDF files.

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 400 and zSeries S/390. 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

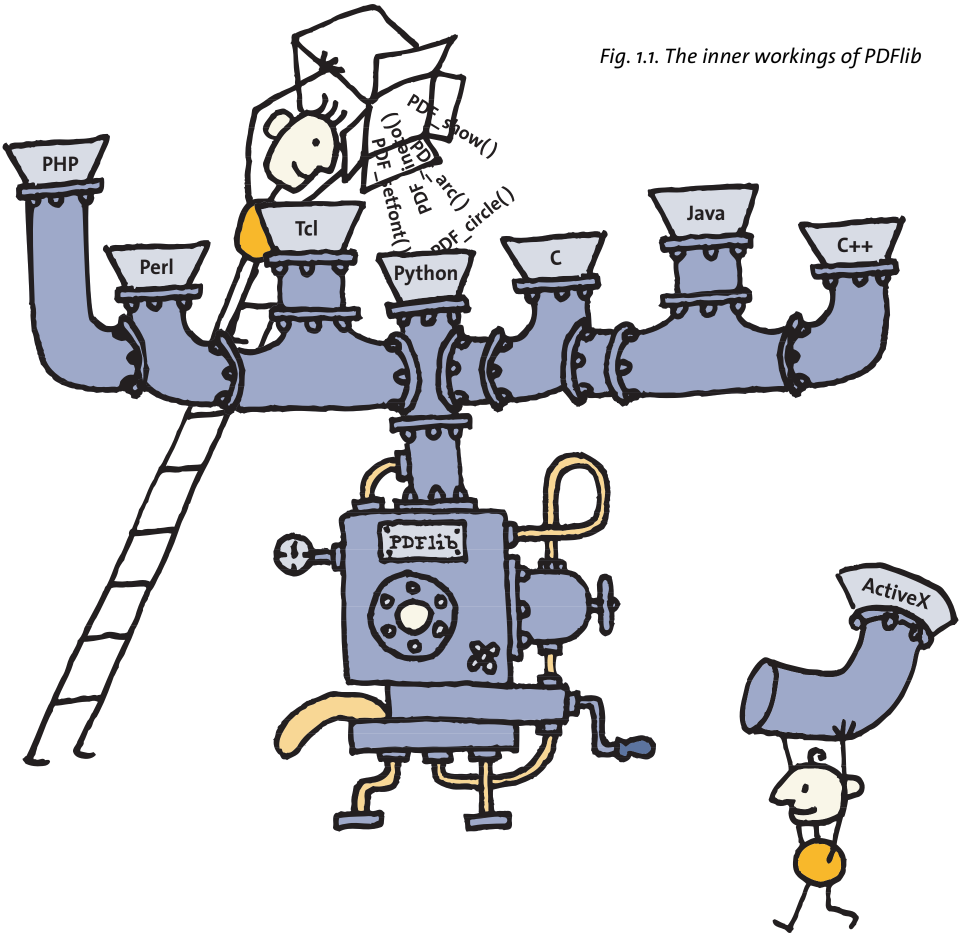


Fig. 1.1. The inner workings of PDFlib

bindings. These language bindings cover all major Web 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:

- ActiveX/COM, providing access from Visual Basic, Active Server Pages with VBScript or JScript, Allaire ColdFusion, Borland Delphi, Windows Script Host, and many other environments
- ANSI C
- ANSI C++
- Java, including servlets
- PHP hypertext processor
- Perl
- Python
- Tcl

What can I use PDFlib for? PDFlib's primary target is creating dynamic PDF within your own software, 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.

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

Requirements for using PDFlib. PDFlib makes PDF generation possible without wading through the 600+ page 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. The function interfaces described in this manual are believed to remain unchanged during future PDFlib development. 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.

1.2 PDFlib Features

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

Table 1.1. PDFlib features for generating PDF

topic	***features***
PDF input	► *existing PDF documents can be imported with the optional PDF import library (PDI)*
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 1.2, 1.3, and 1.4 (Acrobat 3, 4, and 5)*
Vector graphics	► *common vector graphics primitives: lines, curves, arcs, rectangles, etc.* ► *vector paths for stroking, filling, and clipping* ► *grayscale, RGB, CMYK, and spot color for stroking and filling objects* ► *pattern fills and strokes* ► *efficiently re-use text or vector graphics with templates*
Fonts	► *text output in different fonts; underlined, overlined, and strikeout text* ► *text column formatting* ► *built-in font metrics for PDF's 14 base fonts* ► *TrueType and PostScript Type 1 (PFB and PFA file formats) font support with or without font embedding; TrueType fonts can be pulled from the Windows host system* ► *support for AFM and PFM PostScript font metrics files* ► *library clients can retrieve character metrics for exact formatting* ► *on IBM eServer iSeries 400 and zSeries S/390: fetch encodings from the system*
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 (e.g., part number)* ► *file attachments and note annotations*
Internationalization	► *Unicode support (see below)* ► *support for a variety of encodings (both built-in and user-defined)* ► *CID font and CMap support for Chinese, Japanese, and Korean text* ► *support for the Euro character* ► *support for international standards and vendor-specific code pages*
Unicode	► *Unicode support for hypertext features: bookmarks, contents and title of text annotations, document information fields, attachment description, and author name* ► *Unicode code pages for TrueType and PostScript fonts* ► *Unicode encoding for Japanese, Chinese, and Korean text*
Images	► *embed GIF (non-interlaced), PNG, TIFF, JPEG, or CCITT raster images* ► *Images constructed by the client directly in memory* ► *efficiently re-use image data, e.g., for repeated logos on each page* ► *transparent (masked) images*
Programming	► *language bindings for ActiveX/COM, C, C++, Java (including servlets), Perl, PHP, Python, Tcl* ► *transparent Unicode handling for ActiveX, Java, and 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* ► *available for a wide variety of systems, including ASCII- and EBCDIC-based platforms*

1.3 PDFlib Output and Compatibility

PDFlib output. PDFlib generates binary PDF output. The PDF output will be compressed with the Flate (also known as ZIP) compression algorithm. Compression can also be deactivated. Compression applies to potentially large items, such as raster image data and file attachments, as well as text and vector operators on page descriptions. The compression speed/output size trade-off can be controlled with a PDFlib parameter.

Acrobat 4 features. Generally, we strive to produce PDF documents which may be used with a wide variety of PDF consumers. PDFlib generates output compatible with Acrobat 3 and higher.

However, certain features either require Acrobat 4, or don't work in Acrobat Reader but only the full Acrobat product. Table 1.2 lists those features. More details can be found at the respective function descriptions.

Table 1.2. PDFlib features which require Acrobat 4

topic	***remarks***
hypertext	► *file attachments are not recognized in Acrobat 3 (require full Acrobat 4)* ► *different icons for notes are not recognized in Acrobat 3*
page size	► *Acrobat 4 extends the limits for acceptable PDF page sizes*
Unicode	► *Unicode hypertext doesn't work in Acrobat 3*
font	► *the Euro symbol is not supported in Acrobat 3* ► *Unicode support for TrueType fonts doesn't work in Acrobat 3* ► *CID fonts for Chinese, Japanese, and Korean require Acrobat 3J or Acrobat 4*
color	► *the pattern color space is not supported in Acrobat 3 compatibility mode (although patterns can be printed with Acrobat 3, they do not display on screen).*
transparency	► *transparency information is ignored in Acrobat 3*
JPEG images	► *Acrobat 3 supports only baseline JPEG images, but not the progressive flavor*
external images	► *Acrobat 4 (but not the free Acrobat Reader) support external image references via URL. Acrobat 3 is unable to display such referenced images.*

Acrobat 3 compatibility mode. Basically, if you don't use the above-mentioned Acrobat 4 features, the generated PDF files will be compatible to Acrobat 3 and 4. However, due to a very subtle compatibility issue with certain output devices, PDFlib also offers a strict Acrobat 3 compatibility mode. In order to understand the problem, we must distinguish between the actual Acrobat viewer version required by a certain PDF file, and the very first line in the file which may read *%PDF-1.2* or *%PDF-1.3* for Acrobat 3 and Acrobat 4-generated files, respectively. It's important to know that Acrobat 3 viewers open files starting with the *%PDF-1.3* line without any problem, provided the file doesn't use any Acrobat 4 feature. This is the basis of PDFlib's multi-version compatibility approach.

However, some PDF consumers other than Acrobat implement a much stricter way of version control: they simply reject all files starting with the *%PDF-1.3* line, regardless of whether the actual content requires a PDF 1.2 or PDF 1.3 interpreter. For example, some EfI RIPs for high-speed digital printing machines are known to (mis-)behave in this manner. In order to work around this problem, PDFlib offers a strict Acrobat 3 compatibility mode in which a *%PDF-1.2* header is emitted, and Acrobat 4 features are disabled.

Note again that it is not necessary to use PDFlib's strict Acrobat 3 compatibility mode only to make sure the PDF files can be read with Acrobat 3 – this will automatically be the case if you refrain from using the above-mentioned Acrobat 4 features. The strict mode is only required for those rare situations where you have to deal with one of those broken PDF-enabled RIPs.

Acrobat 5 compatibility. PDFlib accepts Acrobat 5 PDF files for import, and will generate Acrobat 5 features in the future. Output compatibility may be set to PDF 1.4 (=Acrobat 5) if Acrobat 5 PDF files are to be imported into the generated document. The PDF import library PDI fully supports PDF 1.4 (see Section 3.5.2, »Using PDI Functions with PDFlib«).

1.4 What's new in PDFlib 4.0?

The following list gives a quick overview of new features in PDFlib 4.0 + PDI:

- support for TrueType fonts on all platforms
- TrueType host font support on Windows (grab fonts directly from the system)
- Unicode-based code pages
- CMYK and spot color
- template feature
- pattern tiling
- fast pass-through mode for PNG images
- PHP language binding
- completely integrates all auxiliary libraries (zlib, libpng, libtiff)
- separate PDF import library (PDI) for dealing with existing PDF documents
- on IBM eServer iSeries 400 and zSeries S/390: fetch encodings from the system
- additional language binding for ILE-RPG

2 PDFlib Language Bindings

2.1 Overview of the PDFlib Language Bindings

2.1.1 What's all the Fuss about Language Bindings?

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 platform-specific (such as ActiveX/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. This includes support for extensions written in the C language in all cases. Looking closer, each environment has its own restrictions and requirements regarding the implementation of extensions.

Fortunately enough, the task of writing language wrappers has been facilitated by a cute program called SWIG[1] (Simplified Wrapper and Interface Generator), written by Dave Beazley. SWIG is brilliant in design and implementation. With the help of SWIG, early PDFlib versions could easily be integrated into the Perl, Tcl, and Python scripting languages. However, over time the requirements for the PDFlib language wrappers grew, until finally it was necessary to manually fine-tune or partially rewrite the SWIG-generated wrapper code. For this reason the language wrappers are no longer generated automatically using SWIG. SWIG support for PDFlib was suggested and in its first incarnation implemented by Rainer Schaaf <*rjs@pdflib.com*>[2].

PDFlib scripting API. In order to avoid duplicating the PDFlib API reference manual for all supported languages, this manual is considered authoritative not only for the C binding but also for all other languages (except ActiveX, for which a separate edition of the manual is available). Of course, the script programmer has to mentally adapt certain conventions and syntactical issues from C to the relevant language. However, translating C API calls to, say, Perl calls is a straightforward process.

2.1.2 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). Given the broad range of platforms and languages (let alone different versions of both) supported by PDFlib, it shouldn't be much of a surprise that not all combinations of platforms, languages, and versions thereof can be tested. However, we strive to make PDFlib work with the latest available versions of the respective environments. Table 2.1 lists the language/platform combinations we used for testing.

[1] More information on SWIG can be found at http://www.swig.org

[2] On a totally unrelated note, Rainer and his wonderful family live in a nice house close to the Alps – definitely a great place for biking!

Table 2.1. Tested language and platform combinations

language	Unix (Linux, Solaris, HP-UX, AIX a.o.)	Windows	Mac OS (Classic)	IBM eServer iSeries 400, zSeries S/390
ActiveX/ COM	–	ASP (PWS, IIS 4 and 5) WSH (VBScript 5, JScript 5) Visual Basic 6.0 Borland Delphi 5 Allaire ColdFusion 4.5	–	–
ANSI C	gcc and other ANSI C compilers	Microsoft Visual C++ 6.0 Metrowerks CodeWarrior 5.3 Borland C++ Builder 5	Metrowerks CodeWarrior 5.3	IBM c89
ANSI C++	gcc and other ANSI C++ compilers	Microsoft Visual C++ 6.0 Metrowerks CodeWarrior 5.3	Metrowerks CodeWarrior 5.3	IBM c89
Java	Sun JDK 1.2.2 and 1.3 IBM JDK 1.1.8 Inprise JBuilder 3.5 Kaffe OpenVM 1.0.5	Sun JDK 1.1.8, 1.2.2, and 1.3 Inprise JBuilder 3.5 and 4 Allaire JRun 3.0 Allaire ColdFusion 4.5	MRJ 2.2, based on JDK 1.1.8	JDK 1.1
Perl	Perl 5.005, 5.6	ActivePerl 5.005 and 5.6	MacPerl 5.2.0r4, based on Perl 5.004	–
PHP	PHP 4.04 and 4.05	PHP 4.04 and 4.05	–	–
Python	Python 1.6 and 2.0	Python 1.6 and 2.0	Python 2.0	–
RPG	–	–	–	V3R7M0 and above
Tcl	Tcl 8.3.2 and 8.4a2	Tcl 8.3.2 and 8.4a2	Tcl 8.3.2	–

Using PDFlib on Mac OS. As shown in Table 2.1, all relevant language bindings are supported on Mac OS (Classic). The following Mac-specific differences should be noted:

- The directory separator (e.g., in *pdflib.upr* configuration files) is the colon ':' character;
- PDFlib correctly sets the file type and creator for generated PDF files;
- The built-in font metrics for the core fonts are arranged according to *macroman* encoding, allowing for easy output of Mac-encoded text.
- The Unix-based concepts of a standard output channel and environment variables don't exist, and are therefore not available in PDFlib;
- The files in the PDFlib source code distribution (source files, sample scripts, PDF documents, etc.) have file type and creator correctly set.

Note *The Darwin edition of PDFlib runs in native mode on Mac OS X.*

Using PDFlib on EBCDIC-based platforms. The operators and structure elements in the PDF file format are completely 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, the PDFlib core library has been carefully crafted in order to allow mixing of ASCII-based PDF operators and EBCDIC (or other) text output. In order to leverage PDFlib's features on EBCDIC-based platforms the following items are expected to be supplied in EBCDIC text format:

- PFA font files, UPR configuration files, AFM font metrics files
- encoding and code page files
- document information (if not Unicode, see function descriptions)
- 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.

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

- PDF output files
- PFB font outline and PFM font metrics files
- TrueType font files
- JPEG, GIF, TIFF, PNG, and CCITT image files

Note Due to restrictions in PDF, text box formatting (PDF_show_boxed( )) is not supported for EBCDIC encoding.

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 and EPOC environments as well as custom embedded systems. For use with restricted environments certain features are configurable in order to reduce PDFlib's overall memory footprint. If you are interested in details please contact us via *sales@pdflib.com*.

2.1.3 The »Hello world« Example

Being a well-known programming classic, the »Hello, world!« example will be used for our examples. It uses PDFlib to generate a one-page PDF file with some text on the page. In the following sections, the »Hello, world!« sample will be shown for all supported language bindings. The code for all language samples is contained in the PDFlib distribution. The distribution contains simple examples for text, vector, and image handling as well as PDF import for all supported language bindings.

2.1.4 Error Handling

PDFlib provides a sophisticated means for dealing with different kinds of programming and runtime errors. In order to allow for smooth integration to the respective language environment, PDFlib's error handling is integrated into the language's native way of dealing with exceptions. Basically, C and C++ clients can install custom code which is called when an error occurs. Other language bindings use the existing exception machinery provided by all modern languages. More details on PDFlib's exception handling can be found in Section 3.1.4, »Error Handling«. The sections on error handling in this chapter cover the language-specific details for the supported environments.

2.1.5 Version Control

Taking into account the rapid development cycles of software in general, and Internet-related software in particular, it is important to allow for future improvements without breaking existing clients. In order to achieve compatibility across multiple versions of the library, PDFlib supports several version control schemes depending on the respective language. If the language supports a native versioning mechanism, PDFlib seamlessly integrates it so the client doesn't have to worry about versioning issues except making use of the language-supplied facilities. In other cases, when the language doesn't support a suitable versioning scheme, PDFlib supplies its own major, minor, and revision version number at the interface level. These may be used by the client in order to decide whether the given PDFlib implementation is acceptable, or should be rejected because a newer version is required.

2.1.6 Unicode Support

PDFlib supports Unicode for a variety of features (see Section 3.3.8, »Unicode Support« for details). The language bindings, however, differ in their native support for Unicode. If a given language binding supports Unicode strings, the respective PDFlib language wrapper is aware of the fact, and automatically deals with Unicode strings in the correct way.

2.1.7 Summary of Language Bindings

For easy reference, Table 2.2 summarizes important features of the PDFlib language bindings. More details can be found in the respective sections of this manual.

Table 2.2. Summary of the language bindings

language	custom error handling	Unicode conversion	version control	thread-safe	EBCDIC-safe
COM/ActiveX	COM exceptions	yes	Class ID and ProgID	yes (both-threading)	–
C	client-supplied error handler	–	manually	yes	yes
C++	client-supplied error handler	–	manually	yes	yes
Java	Java exceptions	yes	automatically	yes	yes
Perl	Perl exceptions	–	via package mechanism	–	–
PHP	PHP warnings	–	manually	yes	–
Python	Python exceptions	–	manually	–	–
RPG	–	–	manually	yes	yes
Tcl	Tcl exceptions	yes (Tcl 8.2 and above)	via package mechanism	yes	–

2.2 ActiveX/COM Binding

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

2.3 C Binding

2.3.1 How does the C Binding work?

In order to use the PDFlib C binding, you need to build a static or shared library (DLL on Windows), and you need the central PDFlib include file *pdflib.h* for inclusion in your PDFlib client source modules. The PDFlib distribution is prepared for building both static or dynamic versions of the library.

On Windows, using DLLs involves some issues related to the function calling conventions and export or import of DLL functions. The *pdflib.h* header file deals with these issues by defining appropriate macros for both the library itself as well as for PDFlib clients. This macro system is set up in a way that PDFlib clients don't need to take any special measures in order to get the required import statements from the header file. However, if you are using function pointers for accessing PDFlib functions (instead of direct calls) you must make sure that your function pointers are declared using the same calling conventions as dictated by *pdflib.h* (depending on whether the static or shared library is used), since otherwise your program will immediately crash.

2.3.2 Availability and Special Considerations for C

PDFlib itself is written in the ANSI C language, and assumes ANSI C clients as well as 32-bit platforms (at least). No provisions have been made to make PDFlib compatible with older C compilers, or 16-bit platforms.

2.3.3 The »Hello world« Example in C

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

#include "pdflib.h"

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

    p = PDF_new();

    /* open new PDF file */
    if (PDF_open_file(p, "hello_c.pdf") == -1) {
        fprintf(stderr, "Error: couldn’t open PDF file.\n");
        exit(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 */

    font = PDF_findfont(p, "Helvetica-Bold", "host", 0);
    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_delete(p);                              /* delete the PDF "object" */
    return 0;
}
```

2.3.4 Error Handling in C

C or C++ clients can install a custom error handler routine with *PDF_new2()*. In case of an exception this routine will be called with a pointer to the PDF structure, the error type, and a descriptive string as arguments. A list of PDFlib error types can be found in Section 3.1.4, »Error Handling«. The opaque data pointer argument to *PDF_new2()* is useful for multi-threaded applications which want to supply a handle to thread- or class-specific data in the *PDF_new2()* call. PDFlib supplies the opaque pointer to the user-supplied error and memory handlers via a call to *PDF_get_opaque()*, but doesn't otherwise use it.

For C and C++ clients which do not install their own error handler, 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 inconsistent state! Since this may not be adequate for a library routine, for serious PDFlib projects it is strongly advised to leverage PDFlib's error handling facilities. A user-defined error handler may, for example, present the error message in a GUI dialog box, and take other measures instead of aborting.

An important task of the error handler is to clean up PDFlib internals using *PDF_delete()* and the supplied pointer to the PDF object. *PDF_delete()* will also close the output file if necessary. PDFlib functions other than *PDF_delete()* and *PDF_get_opaque()* 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.

Except for non-fatal errors (type *NonfatalError)*, client-supplied error handlers must not return to the library function which raised the exception. This can be achieved by using C's *setjmp()/longjmp()* facility. It is an error to call other PDFlib functions for the same PDF document after a fatal exception.

The following code may be used as a starting point for developing a custom error handler for your application and installing it in PDFlib (for general information about PDFlib's exception handling see Section 3.1.4, »Error Handling«):

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

#include "pdflib.h"

typedef struct {        /* there is no portable way to cast a void pointer to a    */
    jmp_buf jbuf;       /* jmp_buf pointer. Therefore we pack it into a structure. */
    /* ... */           /* You can add your thread-specific data here.             */
} err_info;

void custom_errorhandler(PDF *p, int type, const char *msg) {
    err_info *ep;

    fprintf(stderr, "Application error: %s\n", msg);        /* Issue a warning message */

    switch (type) {
        case PDF_NonfatalError:                 /* The error handler may return after  */
            return;                             /* a non-fatal exception.              */

        case PDF_MemoryError:                   /* You can act on specific errors here */
        case PDF_IOError:
        case PDF_RuntimeError:
        case PDF_IndexError:
        case PDF_TypeError:
        case PDF_DivisionByZero:
        case PDF_OverflowError:
        case PDF_SyntaxError:
        case PDF_ValueError:
        case PDF_SystemError:
```

```
        case PDF_UnknownError:
        default:                                    /* do not NOT return to PDFlib! */
            ep = (err_info *) PDF_get_opaque(p);    /* fetch our jmp_buf */
            PDF_delete(p);                          /* important: clean up PDFlib */
            longjmp(ep->jbuf, 1);                   /* now return to the application */
    }
}

int main(void) {
    err_info ei;

    if (setjmp(ei.jbuf) == 0) {
        PDF *p = PDF_new2(custom_errorhandler, 0, 0, 0, &ei);
        /* ... more PDFlib function calls ... */
        PDF_delete(p);
        return(0);
    } else {
        printf("exiting after PDFlib exception!\n");
        return(99);
    }
}
```

Obviously, the appropriate action when an exception occurs is completely application specific. The above sample doesn't even attempt to handle the error, but simply exits.

2.3.5 Version Control in C

In the C language binding there are two basic versioning issues:

- Does the PDFlib header file in use for a particular compilation correspond to the PDFlib binary?
- Is the PDFlib library in use suited for a particular application, or is it too old?

The first issue can be dealt with by comparing the macros *PDFLIB_MAJORVERSION* and *PDFLIB_MINORVERSION* supplied in *pdflib.h* with the return values of the API functions *PDF_get_majorversion( )* and *PDF_get_minorversion( )* which return PDFlib major and minor version numbers.

The second issue can be dealt with by comparing the return values of the above-mentioned functions with fixed values corresponding to the needs of the application.

On Unix platforms the PDFlib library file name may contain version information if the platform supports it (see Appendix A, »Shared Libraries and DLLs«). In this case PDFlib makes use of operating system support for library versioning.

2.3.6 Unicode Support in C

C developers must manually construct their Unicode strings according to Section 3.3.8, »Unicode Support«. For CJK encoding which may contain null characters, the *PDF_show2( )* functions etc. must be used, since their counterparts *PDF_show( )* etc. expect regular null-terminated C-style strings which don't support embedded null characters.

2.3.7 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 4.2, »General Functions«. 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 PDF 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 PDF 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 PDF object, for use in memory management or error handling routines.

2.4 C++ Binding

2.4.1 How does the C++ Binding work?

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

Using the C++ object wrapper effectively replaces the *PDF_* prefix in all PDFlib function names with the more object-oriented *p->* approach. Keep this in mind when reading the PDFlib API descriptions.

2.4.2 Availability and Special Considerations for C++

Although the PDFlib C++ binding assumes an ANSI C++ environment, this is not strictly required by the implementation. In fact, we work around some issues related to non-ANSI-conforming compilers in *pdflib.hpp* and *pdflib.cpp*. It may be worthwhile to add namespace support to the PDFlib C++ wrapper, but this is currently not implemented due to restrictions in the namespace handling of some widely used compilers.

2.4.3 The »Hello world« Example in C++

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

#include "pdflib.hpp"

int
main(void)
{
    PDF *p;                         // pointer to the PDF class
    int font;
    p = new PDF();

    // Open new PDF file
    if (p->open("hello_cpp.pdf") == -1) {
        fprintf(stderr, "Error: couldn't open PDF file.\n");
        exit(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);

    font = p->findfont("Helvetica-Bold", "host", 0);

    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
    delete p;

    return(0);
}
```

2.4.4 Error Handling in C++

Error handling for PDFlib clients written in C++ works the same as error handling in C, so everything in Section 2.3.4, »Error Handling in C« applies to C++, too. In addition, a number of C++ peculiarities must be observed:

A C++ error handler can be supplied in the PDF constructor, which has the same signature as the *PDF_new2()* function. The C++ error handler must be a "C" function, but it may call C++ methods in turn.

In the C++ binding, the *PDF* data type refers to a C++ class, not to the structure used in the C binding (this change is automatically accomplished via macro substitution in the header files). However, the C++ error handler lives on the client side, but has to deal with the PDFlib-internal C data structure. For this reason, C++ error handlers must use the (semi-private) data type name *PDF_c* although the PDFlib API reference calls for the *PDF* data type. A C++ error handler for PDFlib therefore is only slightly different from the C error handler above:

```
void custom_errorhandler(PDF_c *p, int type, const char *msg) {
        /* ... same as C error handler ... */
}
```

Since the error handler must be a "C" function, it should be defined in a separate module *errorhandler.h* which is referenced from the C++ code as follows:

```
extern "C" {
#include "errorhandler.h"
}

int main(void) {
    err_info ei;

    if (setjmp(ei.jbuf) == 0) {
        PDF *p;
        p = new PDF(custom_errorhandler, 0, 0, 0, &ei);
        /* ... more PDFlib method calls ... */
        delete p;
        return(0);
    } else {
        printf("exiting after PDFlib exception!\n");
        return(99);
    }
}
```

Finally, a note for those brave folks who want to throw C++ exceptions in their client-supplied PDFlib error handler: don't do it! Since PDFlib is a C implementation, the error handler will be called from a C-style stack without any exception and stack unwinding information, so throwing a C++ exception in the error handler is likely to result in a crash. The correct way to do it is to install a "C" style error handler, do a *longjmp()* to a C++ method, and throw the C++ exception from there (since we're now back on the C++ stack).

2.4.5 Version Control in C++

Version control for the C++ binding is identical to version control in the C binding (see Section 2.3.5, »Version Control in C«)

2.4.6 Unicode Support in C++

Unicode support for the C++ binding is identical to Unicode support in the C binding (see Section 2.3.6, »Unicode Support in C«).

2.4.7 Memory Management in C++

Client-supplied memory management for the C++ binding works the same as with the C language binding. As with the error handler, the signatures of the memory management routines must be slightly changed to use *PDF_c* instead of *PDF* as their first argument.

The PDF 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.

2.5 Java Binding

2.5.1 How does the Java Binding work?

Starting with the Java[1] Development Kit (JDK) 1.1, 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 (and the availability of source code), 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 runs on more platforms than the Java VM!

2.5.2 Installing the PDFlib Java Edition

Obviously, for developing Java applications you will need the JDK which includes support for the JNI. For compiling the PDFlib-supplied JNI wrapper file (C code), you will need the JNI header files for C, which are part of the JDK (or SDK, if the vendor distinguishes between runtime and development environment).

The JDK has been ported to many Unix and other platforms. Apple's Java implementation, the MacOS Runtime for Java (MRJ), version 2.0 and above, also supports the JNI. 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. In order to maintain a consistent look-and-feel for the Java developer, 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 JDK 1.2 and above 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.

[1] *See http://java.sun.com*

```
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* must be placed in one of the default locations for shared libraries, or in an appropriately configured directory (see Appendix A, »Shared Libraries and DLLs« for details).

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.

Macintosh. The library *pdf_java* will be searched in the *Systems Extensions* folder, the *MRJ Libraries* folder within the *Extensions* folder, and the folder where the starting application lives (JBindery, for example).

PDFlib servlets and Java application servers. PDFlib is perfectly suited for server-side Java applications, especially servlets. The PDFlib distribution contains an example of a PDFlib Java servlet which demonstrates 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.

2.5.3 The »Hello world« Example in Java

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

public class hello
{
    public static void main (String argv[]) throws
        OutOfMemoryError, IOException, IllegalArgumentException,
        IndexOutOfBoundsException, ClassCastException, ArithmeticException,
        RuntimeException, InternalError, UnknownError
    {
        int font;
        pdflib p;

        p = new pdflib();
        if (p.open_file("hello_java.pdf") == -1) {
            System.err.println("Couldn't open PDF file hello_java.pdf\n");
            System.exit(1);
        }

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

        p.begin_page(595, 842);

        font = p.findfont("Helvetica-Bold", "host", 0);

        p.setfont(font, 18);

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

        p.close();
    }
}
```

2.5.4 Error Handling in Java

The Java binding installs a special error handler which translates PDFlib errors to native Java exceptions according to Table 2.3.

Table 2.3. Java exceptions thrown by PDFlib

PDFlib error name	***Java exception***	***explanation***
MemoryError	*java.lang.OutOfMemoryError*	*not enough memory*
IOError	*java.io.IOException*	*input/output error, e.g. disk full*
RuntimeError	*java.lang.IllegalArgumentException*	*wrong order of PDFlib function calls*
IndexError	*java.lang.IndexOutOfBoundsException*	*array index error*
TypeError	*java.lang.ClassCastException*	*argument type error*
DivisionByZero	*java.lang.ArithmeticException*	*division by zero*
OverflowError	*java.lang.ArithmeticException*	*arithmetic overflow*
SyntaxError	*java.lang.RuntimeException*	*syntactical error*
ValueError	*java.lang.IllegalArgumentException*	*a value supplied as argument to PDFlib is invalid*
SystemError	*java.lang.InternalError*	*PDFlib internal error, or incompatible PDFlib library version*
NonfatalError	*java.lang.UnknownError*	*warnings (can be disabled)*
UnknownError	*java.lang.UnknownError*	*other error*

The Java exceptions can be dealt with by applying the appropriate language constructs, i.e., by bracketing critical sections:

```
try {
        ...some PDFlib instructions...
} catch (Throwable e) {
        System.err.println("Exception caught:\n" + e);
}
```

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

2.5.5 Version Control in Java

There is no intrinsic versioning scheme available for PDFlib Java clients. Applications must use manual version control.

2.5.6 Unicode Support in Java

Java supports Unicode natively. The Java language wrapper automatically converts all Java strings to Unicode or ISO Latin 1 (PDFDocEncoding), as appropriate. Java's Unicode-awareness, however, may lead to subtle problems regarding 8-bit encodings (such as *winansi*) and Unicode characters in literal strings. More details on this issue can be found in Section 3.3.8, »Unicode Support«.

Unicode characters can be written directly into code and string literals using a Unicode-aware text editor, or entered with an escape sequence such as

```
p.set_parameter("nativeunicode", "true");
Unicodetext = "\u039B\u039F\u0393\u039F\u03A3";
```

2.6 Perl Binding

2.6.1 How does the Perl Binding work?

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.

2.6.2 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/home/tm/pdflib/bind/perl/.libs hello.pl
```

Unix. Perl will search both *pdflib_pl.so* and *pdflib_pl.pm* 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*

Perl will also search the subdirectory *auto/pdflib_pl*. PDFlib's install mechanism will place the files in the correct directories. Typical output of the above command looks like

```
/usr/lib/perl5/site_perl/5.6/i686-linux
```

Windows. PDFlib supports the ActiveState port of Perl 5 to Windows, also known as ActivePerl.[1] PDFlib does not work with the Microsoft port or other old ports of Perl 5. 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};"
```

Typical output of the above command looks like

```
C:\Program Files\Perl5.6\site\lib
```

Note ActivePerl 5.6 is not compatible to older versions of ActivePerl with respect to extension modules. For this reason pdflib_pl.dll cannot be shared between ActivePerl 5.6 and older versions. The PDFlib binary distribution contains DLLs for both.

Macintosh. Both the shared library *pdflib_pl* and *pdflib_pl.pm* will be searched in the current folder, or in one of the following folders:

```
<MacPerl>:lib:MacPPC
<MacPerl>:lib
```

where *<MacPerl>* denotes the Perl installation folder. In order to run the supplied samples, start Perl and open the script via *Script, Run Script*. It should be noted that the generated PDF output ends up in the Perl interpreter's folder if a relative file name is supplied (as in the sample scripts).

2.6.3 The »Hello world« Example in Perl

```
use pdflib_pl 4.0;

$p = PDF_new();

die "Couldn't open PDF file" if (PDF_open_file($p, "hello_pl.pdf") == -1);

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);
$font = PDF_findfont($p, "Helvetica-Bold", "host", 0);

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);
```

[1] *See http://www.activestate.com*

```
PDF_delete($p);
```

2.6.4 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 $@;
```

2.6.5 Version Control in Perl

Perl's package mechanism supports a major/minor version number scheme for extension modules which is used by the PDFlib Perl binding. PDFlib applications written in Perl simply use the line

```
use pdflib_pl 4.0;
```

in order to make sure they will get the required library version (or a newer one).

2.6.6 Unicode Support in Perl

Perl developers must manually construct their Unicode strings according to Section 3.3.8, »Unicode Support«.

2.7 PHP Binding

2.7.1 How does the PHP Binding work?

The PHP hypertext processor[1] supports a mechanism for extending the language interpreter via internal or external native C libraries. The PDFlib wrapper for PHP consists of a C wrapper module. This module is used to build a shared library or DLL which is loaded at runtime by the PHP interpreter.

The first PDFlib binding for PHP was pioneered by Uwe Steinmann. The current PHP binding is based on Uwe's excellent work. In order to better synchronize PDFlib and PHP releases, and to supply a fully supported language binding for PHP, PDFlib 4.0 and above includes an extended PHP wrapper.

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 4, »PDFlib API Reference«. However, take care when reading the code fragment examples in Section 3, »PDFlib Programming Concepts« since these use the usual PDFlib convention of returning -1 in case of an error.

[1] *See http://www.php.net*

2.7.2 Installing the PDFlib PHP Edition

In order to guarantee full functionality we extended the previous PHP wrapper code. Our new PHP wrapper is included in PHP 4.05 and above, and is also included in PDFlib 4.0.0 and above. Other PHP PDF functions than those discussed in this manual are not supported, although the wrapper code contains a few unsupported functions for backward compatibility. If you are not using PHP 4.05 or above you must copy the PHP wrapper C module distributed with PDFlib to your PHP directory and replace the old PDFlib wrapper which is included with PHP releases older than 4.05.

Detailed information about the various flavors and options for using PDFlib with PHP, 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 source code and binary distributions.

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=php_pdf.dll           ; for 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("php_pdf.dll");              # for Windows
  ```

2.7.3 The »Hello world« Example in PHP

```
<?php
$p =PDF_new();
PDF_open_file($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);
$font = PDF_findfont($p, "Helvetica-Bold", "host", 0);
PDF_setfont($p, $font, 18.0);
PDF_set_text_pos($p, 50, 700);

PDF_show($p, "Hello world!");
PDF_continue_text($p, "(says PHP)");
PDF_end_page($p);
PDF_close($p);

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

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

PDF_delete($p);
?>
```

2.7.4 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");
```

2.7.5 Version Control in PHP

We are currently not aware of any intrinsic versioning scheme available for PHP. Starting with PDFlib 4.0, PHP applications can use manual version control.

2.7.6 Unicode Support in PHP

PHP developers must manually construct their Unicode strings according to Section 3.3.8, »Unicode Support«.

2.8 Python Binding

2.8.1 How does the Python Binding work?

Python[1] supports a mechanism for extending the language (interpreter) via native C libraries. The PDFlib wrapper for Python consists of a C wrapper file. The C module is used to build a shared library which is loaded at runtime by the Python interpreter. The shared library module is referred to from the Python script via an *import* statement.

2.8.2 Installing the PDFlib Python Edition

The Python 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* 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.

[1] *See http://www.python.org*

Note Python 2.0 is not compatible with older versions of Python with respect to extension modules. For this reason pdflib_py.dll cannot be shared between Python 2.0 and older versions. The PDFlib binary distribution contains DLLs for both.

Macintosh. The library *pdflib_py.ppc.slb* will be searched in the *Mac:Plugins* folder of the Python application folder.

2.8.3 The »Hello world« Example in Python

```
from sys import *
from pdflib_py import *

p = PDF_new()

if PDF_open_file(p, "hello_py.pdf") == -1:
    print "Couldn't open PDF file 'hello_py.pdf'\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_findfont(p, "Helvetica-Bold", "host", 0)

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);
```

2.8.4 Error Handling in Python

The Python binding installs a special error handler which translates PDFlib errors to native Python exceptions according to Table 2.4. The Python exceptions can be dealt with by applying the appropriate language constructs, i.e., by bracketing critical sections:

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

Table 2.4. Python exceptions thrown by PDFlib

PDFlib error name	*Python exception*	*explanation*
MemoryError	*MemoryError*	*not enough memory*
IOError	*IOError*	*input/output error, e.g. disk full*
RuntimeError	*RuntimeError*	*wrong order of PDFlib function calls*
IndexError	*IndexError*	*array index error*
TypeError	*TypeError*	*argument type error*
DivisionByZero	*ZeroDivisionError*	*division by zero*

Table 2.4. Python exceptions thrown by PDFlib

PDFlib error name	*Python exception*	*explanation*
OverflowError	*OverflowError*	*arithmetic overflow*
SyntaxError	*SyntaxError*	*syntactical error*
ValueError	*ValueError*	*a value supplied as argument to PDFlib is invalid*
SystemError	*SystemError*	*PDFlib internal error*
NonfatalError	*RuntimeError*	*warnings (can be disabled)*
UnknownError	*RuntimeError*	*other error*

2.8.5 Version Control in Python

We are currently not aware of any intrinsic versioning scheme available for Python. Currently PDFlib applications in Python must use manual version control.

2.8.6 Unicode Support in Python

Python developers must manually construct their Unicode strings according to Section 3.3.8, »Unicode Support«.

2.9 RPG Binding

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

2.10 Tcl Binding

2.10.1 How does the Tcl Binding work?

Tcl[1] supports a mechanism for extending the language (interpreter) via native C libraries. The PDFlib wrapper for Tcl consists of a C wrapper file. The C module is used to build a shared library which is loaded at runtime by the Tcl interpreter. All PDFlib functions are grouped into a single Tcl extension package. The shared library module must be referred to from the Tcl script via a *package* statement.

2.10.2 Installing the PDFlib Tcl Edition

The Tcl extension mechanism works by loading shared libraries at runtime. The supplied PDFlib binaries require Tcl 8.2 or above. The PDFlib wrapper code for Tcl may also be compiled for Tcl 8.0 or 8.1, although Unicode support will be missing.

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 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
```

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

Unix. The library *pdflib_tcl.so* must be placed in one of the default locations for shared libraries, or in an appropriately configured directory (see Appendix A, »Shared Libraries and DLLs« for details).

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
```

Macintosh. The files *pdflib_tcl.shlb* and *pkgIndex.tcl* will be searched in the Tcl shell's folder, and in the folders

```
System:Extensions:Tool Command Language:pdflib
System:Extensions:Tool Command Language:tcl8.3:pdflib
```

In order to run the supplied samples, start the *Wish* application and use the *Source* menu command to locate the Tcl script. It should be noted that the generated PDF output ends up in the Tcl shell's folder if a relative file name is supplied (as in the sample scripts).

2.10.3 The »Hello world« Example in Tcl

```
package require pdflib 4.0

set p [PDF_new]

if {[PDF_open_file $p "hello_tcl.pdf"] == -1} {
    puts stderr "Couldn't open PDF file!"
    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_findfont $p "Helvetica-Bold" "host" 0 ]

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
```

2.10.4 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 applying the appropriate language constructs, i.e., by bracketing critical sections:

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

2.10.5 Version Control in Tcl

Tcl's package mechanism supports a major/minor version number scheme for extension modules which is used by the PDFlib Tcl binding. PDFlib applications written in Tcl simply use the line

```
package require pdflib 4.0
```

in order to make sure they will get the required library version or a newer one.

2.10.6 Unicode Support in Tcl

Starting with version 8.2, Tcl supports Unicode natively. The Tcl language wrapper automatically converts all Tcl strings to Unicode or ISO Latin 1 (PDFDocEncoding), as appropriate. Tcl's Unicode-awareness, however, may lead to subtle problems regarding 8-bit encodings (such as *winansi)* and Unicode characters in literal strings. More details on this issue can be found in Section 3.3.8, »Unicode Support«.

Unicode characters can be written directly into code and string literals using a Unicode-aware text editor, or entered with an escape sequence such as

```
PDF_set_parameter $p "nativeunicode" "true"
set Unicodetext "\u039B\u039F\u0393\u039F\u03A3"
```

3 PDFlib Programming Concepts

3.1 General Programming Issues

3.1.1 The PDFlib Demo Stamp and Serial Numbers

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 serial string.

Once you purchased a PDFlib or PDI serial string you must apply it in order to get rid of the demo stamp. This can be achieved by supplying the serial at runtime:

```
PDF_set_parameter(p, "serial", "...your serial string...");
```

The serial string must be set only once, immediately after instantiating the PDFlib object (i.e., after *PDF_new()* or equivalent call). PDFlib and PDI serial strings are platform-specific, and must be purchased for a particular platform.

Note that PDFlib and PDI are different products, and require different serial strings although they are delivered in a single package. PDI serials will also be valid for PDFlib, but not the other way round. Also, PDFlib and PDI serial strings are platform-dependent, and can only be used on the platform for which they have been purchased.

3.1.2 PDFlib Program Structure

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 page 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 (see Section 4.1, »Data Types, Naming Conventions, and Scope«). The function descriptions document the allowed scope for a particular functions. Calling a function from a different scope will immediately trigger a PDFlib exception. PDFlib will also throw an exception if bad parameters are supplied by a library client.

3.1.3 Generating PDF Documents directly 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 *Fetching PDF data from a buffer requires binary access, and may not be usable from all environments due to restrictions of the respective development environment.*

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 3.1.

Table 3.1. 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*

3.1.4 Error 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, then, is to use conventional error reporting mechanisms (read: special function 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
- Using a font for which metrics information cannot be found
- Trying to open a corrupt image file
- Trying to import an encrypted PDF 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)

If the library detects such an exceptional situation, an error handler is called in order to deal with the situation, instead of passing special return values to the caller. Obviously, the appropriate way to deal with an error heavily depends on the language used for driving PDFlib. For this reason, details on error handling are given in the language-specific sections in Chapter 2. Generally, we let C and C++ clients decide what to do by installing a custom error handler in PDFlib, or propagate the error to the language's native exception handling mechanism (all other language bindings). In the case of native language exceptions, the library client has the choice of catching exceptions and appropriately dealing with them, using the means of the respective language.

PDFlib exceptions fall into one of several categories as shown in Table 3.2. The error handler will receive the type of PDFlib error along with a descriptive message, and present it to the user (for most language bindings), or perform custom operations if a user-supplied error handler was installed (for C and C++).

Non-fatal error messages (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.

Table 3.2. PDFlib runtime errors

error name	*explanation*
MemoryError	*not enough memory*
IOError	*input/output error, e.g. disk full*
RuntimeError	*wrong order of PDFlib function calls*

Table 3.2. PDFlib runtime errors

error name	*explanation*
IndexError	*array index error*
TypeError	*argument type error*
DivisionByZero	*division by zero*
OverflowError	*arithmetic overflow*
SyntaxError	*syntactical error*
ValueError	*a value supplied as argument to PDFlib is invalid*
SystemError	*PDFlib internal error*
NonfatalError	*non-fatal problem. Warnings can be suppressed using PDF_set_parameter( ).*
UnknownError	*other error*

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 in PDF lingo) has the origin in the lower left corner of the page, and uses the DTP point as unit:

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

The first coordinate increases to the right, the second coordinate increases upward. 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.

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 (take care to use suitable material since cheap overhead transparencies do not withstand heat, and may ruin your laser printer!) may provide a useful tool for preparing PDFlib development.

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

- The *Fit to Page* (or *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 page objects.

Note Hypertext functions, such as those for creating text annotations, links, and file annotations are not affected by user space transformations, and always use the default coordinate system instead.

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 above) in centimeters since *72 / 2.54 = 28.3465*.

Rotating objects. It is important to understand that object 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 future objects.
The following example generates some horizontal text, and rotates the coordinate system in order to show vertical text. The save/restore nesting makes it easy to continue with vertical 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, 90);                  /* rotate coordinates */
        PDF_set_text_pos(p, 18, 0);         /* provide for distance from horiz. text */
        PDF_show(p, "vertical 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 set up a coordinate system with the origin in the top left corner of the page and the *y* coordinate pointing downwards while maintaining the usual text direction (text stands upright on the page) use the following code sequence:

```
PDF_begin_page(p, width, height);               /* set up the page dimensions   */
PDF_translate(p, 0, height);                    /* move the coordinate origin   */
PDF_scale(p, 1, -1);                            /* reflect at the horiz. axis   */

font = PDF_findfont(p, "Helvetica-Bold", "host", 0);                /* sample text  */
PDF_setfont(p, font, -18.0);                    /* make the text point upwards  */
PDF_set_value(p, "horizscaling", -100);         /* compensate for the mirroring */

PDF_set_text_pos(p, 50, 100);                   /* now use top-down coordinates */
PDF_show(p, "Hello world!");
```

In order to format text into a text box with the upper right corner at *(x, y)*, width *w*, and height *h* use the following idiom (this is required because the function adds *h* to the starting *y* position):

```
c = PDF_show_boxed(p, text, x, y-h, w, h, "justify", "");
```

Similarly, the following idiom can be used in order to correctly place images when using top-down coordinates:

```
/* Place the image in the lower left corner of the page */
PDF_save(p);
       PDF_translate(p, 0, Height);    /* temporarily translate origin to lower left */
       PDF_scale(p, 1, -1);
       PDF_place_image(p, lImage, 0, 0, 1);
PDF_restore(p);
```

3.2.2 Page and Coordinate Limits

Page sizes. 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. If run in Acrobat 3 compatibility mode PDFlib will throw a *PDF_RuntimeError* exception if the Acrobat 3 limits are exceeded; if run in Acrobat 4 (the default) or 5 compatibility mode and the Acrobat 4 limits are exceeded, PDFlib will only issue a nonfatal warning message. Common standard page size dimensions can be found in Table 3.3.

Table 3.3. Minimum and maximum page size of several PDF consumers

PDF viewer	*minimum page size*	*maximum page size*
Acrobat 3	*1" = 72 pt = 2.54 cm*	*45" = 3240 pt = 114.3 cm*
Acrobat 4 and 5	*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 the box entries of Acrobat 4/PDF 1.3. 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.
- CropBox: the region to which the page contents are to be clipped;
- TrimBox: the intended dimensions of the finished page after trimming;
- ArtBox: extent of the page's meaningful content;
- BleedBox: the region to which the page contents are to be clipped when output in a production environment.

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.

Number of pages in a document. There is no intrinsic limit in PDFlib regarding the number of generated pages in a document. While previous versions of PDFlib generated output which resulted in bad Acrobat performance when navigating large files, PDFlib 4 introduces an improvement in the generated PDF structures which significantly accelerates document navigation in Acrobat even for 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 timing minimizing output file size. As detailed in Table 3.4 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 3.4. Output accuracy and coordinate range

absolute value	***output***
0 ... 0.000015	*0*
0.000015 ... 32767.999999	*rounded to four decimal digits, configurable for up to six digits*
32768 ... 2^{31}- 1	*rounded to next integer*
>= 2^{31}	*an exception of type ValueError will be raised*

3.2.3 Paths and Color

Graphics 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 4.4.4, »Path Painting and Clipping«):

- Stroking draws a line along the path, using client-supplied parameters 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 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 one of the above operations on 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.

Color. PDFlib clients may specify the colors used for filling and stroking the interior of paths and text characters. Colors may be specified in one of 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.
- spot color: an arbitrarily named 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.
- pattern: tiling with an object composed of arbitrary text, vector, or image graphics (patterns are not supported in Acrobat 3 compatibility mode since they don't show up on screen with Acrobat 3).

The default value for stroke and fill color is black, i.e. *(0, 0, 0)* in the RGB color space.

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.

Note *PDF templates are an efficient means for saving space in a PDF file. However, this advantage is usually not retained when printing template-based PDF files to a PostScript printer. Depending on the number of templates used, you should be prepared for print jobs which are significantly larger than the corresponding PDF files.*

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_place_image()* function just like images are placed on the page (see Section 3.4.2, »Code Fragments for Common Image Tasks«). 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_place_image(p, template, (float) 0.0, (float) 0.0, (float) 1.0);
...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 4.6, »Image Functions«, except *PDF_place_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 4.8, »Hypertext Functions«. 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.

Note You can apply all image manipulation algorithms in Section 3.4.2, »Code Fragments for Common Image Tasks« to templates, too. Simply substitute the template width for PDF_get_value (p, "imagewidth", image), similar for template and image height.

Templates and the graphics state. When a template is placed on a page, it will inherit all graphics state parameters from the page unless these are explicitly set within the template definition. For example, if a page description sets the current color to red and places a template which doesn't explicitly set the current color, text and vector elements on the template will be drawn in red color, too. This behavior can be used to change the color of templates, but may also be undesirable for some applications. You have the following choices in this situation:

- Templates which will always be drawn in the same color should specify all required graphics state parameters within the template definition.
- Templates which will be recolored (more precisely: which shall inherit graphics state parameters from the surrounding page) should not set any graphics state parameter. Using such context-dependent templates requires appropriate setup on the surrounding page.
- If the behavior of a template is not known (especially when the template consists of an imported PDF page of unknown origin) all affected graphics state parameters should be reset to their defaults. Since it is usually not known which parameters are used within an imported PDF page, using *PDF_initgraphics()* is the easiest way to assure correct behavior. Note that this function also resets the current transformation matrix.

In a similar fashion, some property of the graphics state may be modified after placing a template or imported PDI page. It is safer to explicitly set the color etc. after placing a template or PDI page, or to bracket the template or page with *PDF_save()/PDF_restore()*.

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, since this type of PDF construct is rarely generated by Acrobat Distiller, not all PDF consumers are prepared to deal with it. For example, not even Acrobat 4's touch-up tool can be used for manipulating templates (this has been fixed in Acrobat 5). Similarly, the PitStop 4.5 PDF editor can only move templates, but cannot access individual elements within a template. On the other hand, Adobe Illustrator 9 fully supports templates.

3.3 Text Handling

3.3.1 The PDF Core Fonts

PDF viewers support a core set of 14 fonts which need not be embedded in any PDF file. Even when a font isn't embedded in the PDF file, PDF and therefore PDFlib need to know about the width of individual characters. For this reason, metrics information for the core fonts is already built into the PDFlib binary. However, the builtin metrics information is only available for the native host encoding (see below). Using another encoding than the host encoding requires metrics information files. Metrics files for the PDF core fonts are included in the PDFlib distribution in order to make it possible to use encodings other than the host encoding. 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

3.3.2 8-Bit Encodings built into PDFlib

PDF supports flexible text encodings (the mapping of numerical code values to character glyphs) for 8-bit text fonts. PDFlib includes provisions for supporting diverse encoding vectors for dealing with text. The builtin encoding vectors are referred to via symbolic names. Table 3.5 lists the symbolic encoding names supported internally by PDFlib. Additional encodings are available in external encoding files distributed with PDFlib (see below), or can be defined by the user (see Section 3.3.3, »Custom Encoding and Code Page Files for 8-Bit Encodings«). 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.*

Table 3.5. Builtin character encodings supported by PDFlib

encoding	*description*
winansi	*Windows code page 1252, a superset of ISO 8859-1*
macroman	*Mac Roman encoding, i.e., the default Macintosh character set*
ebcdic	*EBCDIC code page 1047 as used on IBM AS/400 and S/390 systems*
builtin	*Original encoding used by non-text (symbol) or non-Latin text fonts*
host	*macroman on the Mac, ebcdic on EBCDIC-based systems, and winansi on all others*

The winansi encoding. This encoding reflects the Windows ANSI character set, more specifically code page 1252 including the three characters which Microsoft added for Windows 98 and Windows 2000 (*Euro*, *Zcaron*, and *zcaron*). The *winansi* encoding is a superset of ISO 8859-1 (Latin-1) and can therefore also be used on Unix systems.

Note *Most PostScript fonts do not yet contain the three additional Windows characters. They are supported by the core fonts in Acrobat 4, however.*

The macroman encoding. This encoding reflects the MacOS character set, albeit with the old currency symbol at position 219, and not the Euro character 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 MacOS character set.

The ebcdic encoding. This encoding relates to the EBCDIC *(Extended Binary Coded Decimal Interchange Code)* defined by IBM and used on the IBM AS/400, S/390, and other midrange and mainframe systems. More specifically, PDFlib's *ebcdic* encoding uses the EBCDIC code page 1047. As with all other PDFlib encodings, *ebcdic* encoding is always available for generating PDF output, and not only on native EBCDIC machines. The difference, however, is that on those machines the built-in metrics for the core fonts are sorted according to *ebcdic* encoding, and that *host* encoding (see below) also relates to *ebcdic* encoding.

The builtin encoding. The encoding name *builtin* doesn't describe a particular character ordering but rather means »take this font as it is, and don't mess around 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), or non-Latin text fonts (such as Greek and Cyrillic). Such fonts cannot be reencoded using one of the supported encodings since their character names don't match those in these encodings. Therefore, *builtin* must be used for all symbolic or non-text 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.

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.*

The host encoding. Like *builtin*, the *host* encoding plays a special role since it doesn't refer to some fixed character set. Instead, *host* encoding will be mapped to *macroman* on the Mac, *ebcdic* on EBCDIC-based systems, and *winansi* on all others. The *host* encoding is primarily useful as a vehicle for writing platform-independent test programs (like those contained in the PDFlib distribution) or other encoding-wise simple applications. Assuming that PDFlib client programs are always encoded in the host's native encoding, such programs will always generate PDF text output with the »correct« encoding. Contrary to all other aspects of PDFlib, the concept of a *host* encoding is inherently non-portable. For this reason *host* encoding is not recommended for production use.

3.3.3 Custom Encoding and Code Page Files for 8-Bit Encodings

In addition to a number of predefined encodings (see Section 3.3.2, »8-Bit Encodings built into PDFlib«) PDFlib supports user-defined 8-bit encodings in order to make PDFlib's font handling even more flexible. User-defined encodings 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. In addition to encoding tables defined by PostScript glyph names PDFlib also accepts code page tables which describe a mapping from Unicode to a set of up to 256 characters. These characters can be accessed with 8-bit character codes.

The following tasks must be done before a user-defined encoding can be leveraged in a PDFlib program:

- Generate a description of the encoding in a simple text format.
- Configure the encoding in the PDFlib resource file (see Section 3.3.6, »Resource Configuration and the UPR Resource File«) or via *PDF_set_parameter( )*.
- Provide a font (metrics and possibly outline file) that supports all characters used in the encoding. Of course, the characters in the font must use the correct PostScript glyph names as defined in the encoding table.

The encoding file simply lists glyph names and numbers line by line. As an example, the following excerpt shows the encoding definition for the ISO 8859-2 (Latin 2) character set:

```
% Encoding definition for PDFlib
% ISO 8859-2 (Latin-2)
space           32      % 0x20
exclam          33      % 0x21
quotedbl        34      % 0x22
...more glyph assignments...
yacute          253     % 0xFD
tcommaaccent    254     % 0xFE
dotaccent       255     % 0xFF
```

The next example shows a snippet from a Unicode code page for the same ISO 8859-2 character set:

```
% Code page definition for PDFlib
% ISO 8859-2 (Latin-2)
0x0020          32      % 0x20
0x0021          33      % 0x21
0x0022          34      % 0x22
...more glyph assignments...
0x00FD          253     % 0xFD
0x0163          254     % 0xFE
0x02D9          255     % 0xFF
```

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 character name or a hexadecimal Unicode value composed of a *0x* prefix and four hex digits (upper or lower case). This is followed by whitespace and a hexadecimal or decimal character code in the decimal range 0–255. Only Unicode values in the Adobe Glyph List (AGL) are allowed (see below).
- 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 used for unencoded characters.

As a naming convention we refer to name-based tables as encoding files *(*.enc)*, and Unicode-based tables as code page files *(*.cpg)*, although actually 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]), which is built into PDFlib. Encoding files are required for PostScript fonts with non-standard glyph names, while code pages are more convenient when dealing with Unicode-based TrueType fonts.

The relationship between the name of the encoding file and the name of the actual encoding (to be used with *PDF_findfont( ))* is specified in PDFlib's resource file or via *PDF_set_parameter( )* (see Section 3.3.6, »Resource Configuration and the UPR Resource File«).

Distributed encoding files. The PDFlib distribution contains several encoding and code page files (see Table 3.6) which may be useful if you need to use one of the supplied encodings directly, or want to use it as a starting point for writing your own encoding files. In order to use these, the PDFlib resource configuration file and font metrics files must be accessible (see Section 3.3.6, »Resource Configuration and the UPR Resource File«).

Host encoding support and CCSIDs. In addition to built-in and user-defined encodings, PDFlib can also be instructed to fetch encoding definitions from the operating system. This is very convenient since it frees you from implementing the code page definition yourself and configuring it via the UPR file.

The host encoding feature is platform-dependent, and is currently only available on IBM eServer iSeries 400 and zSeries S/390. Instead of supplying the name of a built-in or user-defined encoding for *PDF_findfont( )*, simply use an encoding name which is known to the system (more specifically, the *iconv* facility). On IBM systems any *Coded Character Set Identifier* (CCSID) can be used. The CCSID must be supplied as a string. PDFlib will fetch the corresponding code page definition from the system and transform it appropriately for internal use. The following example fetches the German EBCDIC code page:

```
PDF_findfont(p, "Helvetica", "00273", 0);
```

Finding PostScript character 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 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 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).[3]

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

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

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

[3] Useful raw material for writing encoding tables for a variety of standards and vendor-specific character sets can be found at ftp://ftp.unicode.org/Public/MAPPINGS; 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).

Table 3.6. Additional external character encodings distributed with PDFlib

encoding	***description***
iso8859-2	*Latin-2 supports the Slavic languages of Central Europe which use the Latin alphabet. Acrobat 4's core fonts do not contain all characters for ISO 8859-2.*
iso8859-3	*Latin-3 covers Esperanto and Maltese.*
iso8859-4	*Latin-4 covers Estonian, the Baltic languages, Greenlandic, and Lappish.*
iso8859-5	*ISO 8859-5 (Cyrillic) covers Bulgarian, Russian, and Serbian. Acrobat 4 does not correctly implement TrueType handling for Cyrillic characters.*
iso8859-6	*ISO 8859-6 covers Arabic, but not Persian or Pakistani/Urdu.*
iso8859-7	*ISO 8859-7 covers modern Greek*
iso8859-8	*ISO 8859-8 covers Hebrew and Yiddish.*
iso8859-9	*Latin-5 (yes, that's 5, not 9!) supports Danish, Dutch, English, Finnish, French, German, Irish, Italian, Norwegian, Portuguese, Spanish, Swedish, and Turkish. In addition, this PDFlib encoding contains the characters 130-159 (0x82–0x9F) as defined in the Windows code page 1254 (Turkish). Acrobat 4's core fonts do not contain the following characters for ISO 8859-9: Gbreve, gbreve, Idotaccent, Scommaaccent, scommaaccent.*
iso8859-10	*Latin-6 is a variation of Latin-4 and covers the Nordic area.*
iso8859-15	*Latin-9: this character set is a variation of Latin-1 which adds the Euro character as well as some missing French and Finnish characters. Latin-9 is sometimes also dubbed Latin-0, although this is not the official name.*
cp1250	*Windows code page 1250 (Central European). Acrobat 4's core fonts do not contain all characters for code page 1250.*
cp1251	*Windows code page 1251 (Cyrillic). Acrobat 4 does not correctly implement TrueType handling for Cyrillic characters.*
cp1253	*Windows code page 1253 (Greek)*
cp1254	*Windows code page 1254 (Turkish)*
cp1255	*Windows code page 1255 (Hebrew)*
cp1256	*Windows code page 1256 (Arabic)*
cp1257	*Windows code page 1257 (Baltic)*
cp1258	*Windows code page 1258 (Viet Nam)*

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, or view it with a Level-2-compatible PostScript viewer such as Ghostscript[1]. The program will print all characters in the font, sorted alphabetically by glyph name.

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

3.3.4 Hypertext Encoding

PDF supports two methods for encoding hypertext elements such as bookmarks, annotations, and document information fields. Up to Acrobat 3, all hypertext strings had to be encoded with a special 8-bit encoding called PDFDocEncoding (PDFDocEncoding can not be used for text used on page descriptions). Starting with Acrobat 4, Unicode strings can be used for all hypertext elements. For more information on Unicode see Section 3.3.8, »Unicode Support«.

[1] *See http://www.cs.wisc.edu/~ghost*

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	000	001	002	003	004	005	006	007	010	011	012	013	014	015	016	017
1	020	021	022	023	024	025	026	027	˘ 030	ˇ 031	ˆ 032	˙ 033	˝ 034	˛ 035	˚ 036	˜ 037
2	040	! 041	" 042	# 043	$ 044	% 045	& 046	™ 047	(050	) 051	* 052	+ 053	, 054	- 055	. 056	/ 057
3	0 060	1 061	2 062	3 063	4 064	5 065	6 066	7 067	8 070	9 071	: 072	; 073	< 074	= 075	> 076	? 077
4	@ 100	A 101	B 102	C 103	D 104	E 105	F 106	G 107	H 110	I 111	J 112	K 113	L 114	M 115	N 116	O 117
5	P 120	Q 121	R 122	S 123	T 124	U 125	V 126	W 127	X 130	Y 131	Z 132	[133	\ 134	] 135	^ 136	_ 137
6	, 140	a 141	b 142	c 143	d 144	e 145	f 146	g 147	h 150	i 151	j 152	k 153	l 154	m 155	n 156	o 157
7	p 160	q 161	r 162	s 163	t 164	u 165	v 166	w 167	x 170	y 171	z 172	{ 173	| 174	} 175	~ 176	177
8	• 200	† 201	‡ 202	… 203	— 204	– 205	ƒ 206	⁄ 207	‹ 210	› 211	− 212	‰ 213	„ 214	“ 215	” 216	‘ 217
9	’ 220	‚ 221	™ 222	ﬁ 223	ﬂ 224	Ł 225	Œ 226	Š 227	Ÿ 230	Ž 231	ı 232	ł 233	œ 234	š 235	ž 236	237
A	€ 240	¡ 241	¢ 242	£ 243	¤ 244	¥ 245	¦ 246	§ 247	¨ 250	© 251	ª 252	« 253	¬ 254	255	® 256	¯ 257
B	° 260	± 261	² 262	³ 263	´ 264	µ 265	¶ 266	· 267	¸ 270	¹ 271	º 272	» 273	¼ 274	½ 275	¾ 276	¿ 277
C	À 300	Á 301	Â 302	Ã 303	Ä 304	Å 305	Æ 306	Ç 307	È 310	É 311	Ê 312	Ë 313	Ì 314	Í 315	Î 316	Ï 317
D	Ð 320	Ñ 321	Ò 322	Ó 323	Ô 324	Õ 325	Ö 326	× 327	Ø 330	Ù 331	Ú 332	Û 333	Ü 334	Ý 335	Þ 336	ß 337
E	à 340	á 341	â 342	ã 343	ä 344	å 345	æ 346	ç 347	è 350	é 351	ê 352	ë 353	ì 354	í 355	î 356	ï 357
F	ð 360	ñ 361	ò 362	ó 363	ô 364	õ 365	ö 366	÷ 367	ø 370	ù 371	ú 372	û 373	ü 374	ý 375	þ 376	ÿ 377

Fig. 3.1. The PDFDocEncoding character set as defined in PDF 1.3 with hex and octal codes. Note the Euro character at position hexadecimal A0 = octal 240.

PDFDocEncoding (see Figure 3.1) is a superset of ISO 8859-1 (Latin 1) and therefore contains all ASCII characters in the lower part. Although PDFDocEncoding and the Windows code page 1252 are quite similar, they differ substantially in the character range 128-160 (0x80–0xA0).

Many clients will be able to directly use PDFDocEncoding. However, since the Mac encoding substantially differs from PDFDocEncoding, it is necessary to convert Mac strings to PDFDocEncoding when it comes to hypertext elements, and non-ASCII special characters are to be used. Mac special characters must be converted to Unicode before they can be used in hypertext elements. This conversion must be performed by the client.

Note Hypertext strings will automatically be converted to PDFDocEncoding on EBCDIC systems.

3.3.5 PostScript and TrueType Fonts

Font embedding in PDF. PDF supports fonts outside the set of 14 core fonts in several ways. PDFlib is capable of embedding font descriptions into the generated PDF output. Alternatively, a font descriptor consisting of the character metrics and some general information about the font (without the actual character outline data) 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 in the PDF. Table 3.7 lists different situations with respect to font usage, each of which poses different requirements on the necessary font and metrics files.
When a font with font-specific encoding (a symbol font) is used, but not embedded in the PDF output, the resulting PDF will be unusable unless the font in question 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 3.7. 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 with PDFlib's host encoding[1,2]	*no*	*no*
One of the 14 core fonts with an encoding other than PDFlib's host encoding[2]	*yes (AFM files supplied with the PDFlib distribution)*	*no*
Non-core PostScript fonts without embedding	*yes*	*no*
Non-core PostScript fonts with embedding	*yes*	*yes*
Additional font/encoding combinations for which the metrics have been compiled into PDFlib (see below)	*no*	*yes, if embedding is requested*
TrueType fonts with or without embedding	*no*	*yes*
Standard CID fonts[3]	*no*	*no*
Non-standard CID fonts	*(not supported)*	*(not supported)*

[1] See Section 3.3.1, »The PDF Core Fonts« for a list of core fonts.
[2] See Section 3.3.2, »8-Bit Encodings built into PDFlib« for the definition of PDFlib's host encoding.
[3] See Section 3.3.7, »CID Font Support for Japanese, Chinese, and Korean Text« for more information on CID fonts.

PostScript fonts. PDFlib supports the following formats for PostScript 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«). PostScript Type 3 fonts are not supported.

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.

PostScript font names. It is important to use the exact (case-sensitive) PostScript font name whenever a font is referenced in PDFlib. There are several possibilities to find a PostScript font's exact name:

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

- 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, 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.

Performance notes for PostScript fonts. It is important to be aware of the impact of font handling issues on PDFlib's performance. Generally, the font metrics (either incore or on file) are accessed whenever a certain font/encoding combination is used for the first time. Subsequent requests for the same combination will be satisfied from PDFlib's internal font cache without any further performance penalty. Regarding font handling performance, the following observations may be useful:

- Due to their small size and binary nature, PFM metrics files can be read much faster than the text-based AFM metrics files. However, they cannot be used for arbitrary encodings.
- AFM files contain much useful information about many aspects of font usage, and can be used for arbitrary encodings. However, although only the bare character metrics are required for PDFlib, the complete AFM file must be parsed in a time-consuming manner. For performance-critical applications it might be worthwhile to strip the unneeded data (e.g., the kerning information) from the AFM file.
- For specific applications the performance can be improved very much by compiling the metrics information for the required font/encoding combinations into the PDFlib binary, thereby obviating the use for external metrics files at all. The *compile_metrics* utility supplied with PDFlib can be used for constructing a C header file with the required data. (The default metrics data built into PDFlib have also been generated with this utility.) *compile_metrics* requires re-compiling the PDFlib binary and is therefore only useful to C or C++ developers.

TrueType fonts. PDFlib supports TrueType and OpenType fonts on all platforms. The TrueType font file must be supplied in Windows TTF format (Macintosh resource format is not supported). OpenType fonts are only supported if they contain TrueType outlines (as opposed to OpenType fonts which contain PostScript outlines, also known as CFF fonts). Contrary to PostScript fonts, TrueType fonts do not require any additional metrics file since the metrics information can be extracted from the font file itself. PDFlib currently supports the following flavors of TrueType fonts:

- Standard latin text fonts with the Windows character set (in TrueType lingo: *cmap* table with platform id 3, encoding id 1). These must be used with PDFlib encoding *winansi*. Note that most (but not all) TrueType fonts for Windows can also be used with PDFlib encoding *macroman* since they contain the necessary Mac information *(cmap* table with platform id 1, encoding id 0) in addition to the Windows information.
- Unicode-compatible TrueType fonts. These can be used with any PDFlib encoding as long as the font actually contains the characters required by that encoding. You can

check which code pages are supported by a particular font with the »font properties extension« mentioned below.

- Symbol fonts with a custom character set (in TrueType lingo: *cmap* table with platform id 3, encoding id 0). These must be used with PDFlib encoding *builtin*.

The above distinction between text and symbol fonts may seem obvious, but in practise it may be hard to find the appropriate category for a given font. For various reasons, text fonts may be coded as TrueType *symbol* fonts, and vice versa. In case of an encoding mismatch PDFlib tries to help, and supplies encoding suggestions in the message which is part of a PDFlib exception.

TrueType host fonts. In addition to accessing font files which have been configured via PDFlib's resource and parameter machinery (see Section 3.3.6, »Resource Configuration and the UPR Resource File«) TrueType fonts can also be fetched directly from 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 *fonts* directory), and PDFlib will happily use it.

Note *Host font support is restricted to TrueType fonts and the Windows platform. Host fonts will not be used for embedding one of the core fonts.*

TrueType font names. It is important to specify the exact (case-sensitive) TrueType font name whenever a font is referenced in PDFlib. This must be the Windows name of the font as it is exposed at the user interface. You can easily find this name by double-clicking the TrueType font file in Windows, 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, and must also be used in translated form in PDFlib.

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 differ from its genuine TrueType name (e.g., *TimesNewRomanPSMT* vs. *Times New Roman).*

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.

Note *Contrary to PostScript fonts, TrueType font names may contain blank characters (and often do).*

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 the fonts are subsetted. Please check the legal implications of font embedding before attempting to embed fonts with PDFlib. PDFlib will honour embedding restrictions which may be specified in a True-

[1] *See http://www.microsoft.com/typography/property/property.htm*

Type 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.

Note PDFlib currently doesn't implement font subsetting.

3.3.6 Resource Configuration and the UPR Resource File

In order to make PDFlib's font and encoding handling platform-independent and customizable, a configuration file can be supplied for describing the available fonts along with the names of their outline and metrics files, and the names of additional encoding files. In addition to the 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 will take the liberty of extending the original UPR format for our purposes. The UPR file format as used by PDFlib will be described below.[2] 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.

Note As an alternative to configuring fonts in UPR files or via PDF_set_parameter( ) you can make use of PDFlib host font feature in certain situations (see Section 3.3.5, »PostScript and TrueType Fonts«)

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 any character, including newline characters. This may be used to extend lines. Windows directory names must be separated by double backslashes '\\' or a single forward slash '/'.
- The period character ' . ' serves as a section terminator, and must therefore be escaped when used at the start of any other line.
- 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.

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 types of 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. This section exists for compatibility only, and is ignored by PDFlib.
- The optional directory line may be used as a shortcut for a path prefix common to all resource files described in the file. The prefix will be added to all file names given in the UPR file. If present, the directory line starts with a slash character, immediately

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

[2] *The complete specification can be found in the book »Programming the Display PostScript System with X« (Appendix A), available at http://partners.adobe.com/asn/developer/PDFS/TN/DPS.refmanuals.TK.pdf*

followed by the directory prefix. Note that the initial slash character is required on all platforms, and is not part of the path name. Using the directory prefix a UPR file may, for example, point to some central PostScript font directory somewhere in the file system.

- 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), an equal sign, and the corresponding relative or absolute file name for the resource. Relative file names will have the directory prefix applied, if one is present in the file. Using a double equal sign forces the file name to be interpreted absolute, i.e., the prefix is not used.

Supported resource categories. The resource categories supported by PDFlib are listed in Table 3.8. Other resource categories may be present in the UPR file for compatibility with Display PostScript installations, but they will silently be ignored.

Table 3.8. Resource categories supported in PDFlib

resource category name	***explanation***
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*

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).

Sample UPR file. The following listing gives an example of a UPR configuration file as used by PDFlib. It describes the 14 PDF core fonts' metrics, plus metrics and outline files for some additional fonts, plus a custom encoding:

```
PS-Resources-1.0
FontAFM
FontPFM
FontOutline
Encoding
.
% Directory prefix example for Windows: /c:/psfonts
//usr/local/lib/fonts

FontAFM
Code-128=Code_128.afm
Courier=Courier.afm
Courier-Bold=Courier-Bold.afm
Courier-BoldOblique=Courier-BoldOblique.afm
Courier-Oblique=Courier-Oblique.afm
Helvetica=Helvetica.afm
Helvetica-Bold=Helvetica-Bold.afm
Helvetica-BoldOblique=Helvetica-BoldOblique.afm
Helvetica-Oblique=Helvetica-Oblique.afm
```

```
Symbol=Symbol.afm
Times-Bold=Times-Bold.afm
Times-BoldItalic=Times-BoldItalic.afm
Times-Italic=Times-Italic.afm
Times-Roman=Times-Roman.afm
ZapfDingbats=ZapfDingbats.afm
.
FontPFM
Foobar-Bold=foobb___.pfm
% Example for an absolute path name with the prefix not applied (two equal signs)
Mistral==c:/psfonts/pfm/mist____.pfm
.
FontOutline
Code-128=Code_128.pfa
ArialMT=Arial.ttf
.
Encoding
iso8859-2=iso8859-2.enc
cp1250=cp1250.cpg
.
```

Searching for the UPR resource file. If only the built-in resources are to be used (PDF core fonts with host encoding), a UPR configuration file is not required, since PDFlib contains all necessary resources.

If other resources are to be used, PDFlib will search several places for a resource file. The process is configurable and consists of the following steps:

- On Unix and Windows systems, the environment variable *PDFLIBRESOURCE* is examined and used as a resource file name.
- If no file name is found, the client-settable *resourcefile* parameter is examined and used as a resource file name, if set. This parameter can be set at runtime:

  ```
  PDF_set_parameter(p, "resourcefile", "/usr/local/fonts/pdflib.upr");
  ```

- If no file name is found, the file name *pdflib.upr* in the current directory is used.
- If this file can’t be opened, an *IOError* is raised.
- If a resource file can be opened during any of the above steps, but a required resource category cannot be found, a *SystemError* is raised.

Note Don’t forget to set the prefix entry in the upr file accordingly. The path to the upr file is not automatically prepended to the resource file names listed in the upr file. To prevent the prefix from being applied to a particular resource entry use double equal signs as described above.

Setting resources without a UPR file. 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")
```

Similar to UPR files, if two equal signs are present, the file name will be interpreted absolute. If only a single one equal sign is present, the directory prefix will be used if one has been configured.

3.3.7 CID Font Support for Japanese, Chinese, and Korean Text

CJK support in Acrobat and PDF[1]. While Japanese font support was already available in Acrobat 3J, Acrobat 4 added full support for CID (Character ID) fonts for Japanese, Chinese, and Korean (CJK) text even in the non-Japanese versions of the full Acrobat package as well as the free Acrobat Reader. In order to use CJK documents in Acrobat you must do one of the following:

- 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 encodings) 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 4 product CD-ROM, or on the Web.[2]

CJK encodings and fonts. Historically, a wide variety of CJK encoding schemes has been developed by diverse standards bodies, companies, and other organizations. Fortunately enough, all prevalent encodings are supported by Acrobat and PDF by default. Acrobat 4 supports a wealth of different encoding schemes for CJK fonts. Since the concept of an encoding is much more complicated for CJK text than for Latin text, simple encoding vectors with 256 entries no longer suffice. Instead, PostScript and PDF use the concept of character collections and character maps (CMaps) for organizing the characters in a font. Conceptually, CMaps can be thought of as large encodings for CJK fonts.

Acrobat 4 supports a set 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 (this parallels the use of the 14 core fonts for Latin text). 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 3.9. 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[3]; CMap descriptions can be found in Table 3.10.

CJK font support in PDFlib. Having realized the similarity between core fonts/encoding vector on the one hand, and CJK standard fonts/CMaps on the other hand, it won't be much of a surprise that both Latin and CJK fonts can be selected with the same PDFlib interface, using the CMap name in lieu of the encoding name, and taking into account that a given CJK font supports only a certain set of CMaps (see Table 3.9). The *HeiseiKakuGo* sample in Table 3.9 has been generated with the following code:

```
font = PDF_findfont(p, "HeiseiKakuGo-W5", "Ext-RKSJ-H", 0);
PDF_setfont(p, font, 24);
PDF_set_text_pos(p, x, y);
PDF_show(p, "\x93\xFA\x96\x7B\x8C\xEA");
```

[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/prodindex/acrobat/cjkfontpack.html*

[3] *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 3.9. Acrobat’s standard fonts for Japanese, Chinese, and Korean text

locale	*font name*	*font samples*	*supported CMaps (encodings)*
Simplified Chinese	*STSong-Light*	国际	*GB-EUC-H, GB-EUC-V, GBpc-EUC-H, GBpc-EUC-V, GBK-EUC-H, GBK-EUC-V, UniGB-UCS2-H, UniGB-UCS2-V*
Traditional Chinese	*MHei-Medium* *MSung-Light*	中文 中文	*B5pc-H, B5pc-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* *HeiseiMin-W3*	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* *HYSMyeongJo-Medium*	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*

These instructions locate one of the Japanese standard fonts, choosing a Shift-JIS-compatible CMap *(Ext-RKSJ)* encoding and horizontal writing mode *(H)*. The *fontname* parameter must be the exact name of the font (strictly speaking, the value of the */CIDFontName* entry in the corresponding CID PostScript font file), 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 below)*. 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, asking PDFlib to use a Korean font with a Japanese encoding will result in an exception of type *PDF_ValueError*.

Although CID font embedding is technically possible in PDF 1.3, it is not practical due to the size of typical CID fonts, and due to the fact that most CJK font licenses do not permit embedding. For this reason the *embed* parameter is not used for CID fonts, and must be 0.

PDFlib doesn’t require any font-specific metrics information for CID fonts, and doesn’t make any attempt to decode the client-supplied text strings, or verify whether they are correctly encoded with respect to the underlying CMap. For this reason the following features are currently not supported for CID fonts:

- calculating the extent of text with *PDF_stringwidth( )*
- box formatting with *PDF_show_boxed( )*
- activating underline/overline/strikeout mode
- retrieving the *textx/texty* position

Also, all characters in CJK fonts are considered to have the same width, including Latin characters. The character width is equal to the font size. If you want Latin characters which have a smaller width than the CJK characters you must switch to a Latin 8-bit font such as Courier or Helvetica.

Note PDFlib currently only supports the standard CID fonts supplied with Acrobat (see Table 3.9). Neither custom CID fonts nor Japanese, Chinese, or Korean TrueType fonts can be used. However, you can simulate bold fonts by rendering »filled and stroked« text (rendering mode 2, see textrendering parameter).

Horizontal and vertical writing mode. PDFlib supports both horizontal and vertical writing modes. The mode is selected along with the encoding by choosing the appropri-

Table 3.10. Predefined CMaps for Japanese, Chinese, and Korean text (from the PDF Reference)

locale	***supported CMaps***	***description***
Simplified Chinese	*GB-EUC-H* *GB-EUC-V*	*Microsoft Code Page 936 (lfCharSet 0x86), 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 *GBK-EUC-V*	*Microsoft Code Page 936 (lfCharSet 0x86), GBK character set, GBK encoding*
	UniGB-UCS2-H *UniGB-UCS2-V*	*Unicode (UCS-2) encoding for the Adobe-GB1 character collection*
Traditional Chinese	*B5pc-H* *B5pc-V*	*Macintosh, Big Five character set, Big Five encoding, Script Manager code 2*
	ETen-B5-H *ETen-B5-V*	*Microsoft Code Page 950 (lfCharSet 0x88), 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 *CNS-EUC-V*	*CNS 11643-1992 character set, EUC-TW encoding*
	UniCNS-UCS2-H *UniCNS-UCS2-V*	*Unicode (UCS-2) encoding for the Adobe-CNS1 character collection*
Japanese	*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 (lfCharSet 0x80), 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 *Add-RKSJ-V*	*JIS X 0208 character set with Fujitsu FMR extensions, Shift-JIS encoding*
	EUC-H *EUC-V*	*JIS X 0208 character set, EUC-JP encoding*
	Ext-RKSJ-H *Ext-RKSJ-V*	*JIS C 6226 (JIS78) character set with NEC extensions, Shift-JIS encoding*
	H *V*	*JIS X 0208 character set, ISO-2022-JP encoding*
	UniJIS-UCS2-H *UniJIS-UCS2-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*
Korean	*KSC-EUC-H* *KSC-EUC-V*	*KS X 1001:1992 character set, EUC-KR encoding*
	KSCms-UHC-H *KSCms-UHC-V*	*Microsoft Code Page 949 (lfCharSet 0x81), 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*
	UniKS-UCS2-H *UniKS-UCS2-V*	*Unicode (UCS-2) encoding for the Adobe-Korea1 character collection*

ate CMap name. CMaps with names ending in *-H* select horizontal writing mode, while the *-V* suffix selects vertical writing mode.

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. The details are discussed in the respective function descriptions.

CJK text encoding in PDFlib. The client is responsible for supplying text such that its encoding matches the encoding requested for the CID font. PDFlib does not check whether the supplied text conforms to the requested encoding.Since several of the supported encodings may contain null characters in the text strings, C and 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 bindings, the text functions support binary strings, and *PDF_show2( )* etc. are not required. For multi-byte encodings, the high-order byte of a character must appear first.

PDFlib language bindings which are natively Unicode-aware automatically convert Unicode strings supplied to the library. For this reason only Unicode-compatible CMaps should be used with these language bindings when the *nativeunicode* parameter is set to *true* (see also Section 3.3.8, »Unicode Support«).

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 with Acrobat, consider one or more of the following:

- 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 (unless the printer is equipped with the Type 0 font extensions). 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 (disk files for CID fonts typically are 5–10 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).

3.3.8 Unicode Support

Starting with version 4, Acrobat supports the Unicode standard, almost identical to ISO 10646[1]. This 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 and operating systems. PDFlib supports the Unicode standard for the following features:

- bookmarks (see Figure 3.2)
- contents and title of note annotations (see Figure 3.2)
- standard and user-defined document information field contents (but not user-defined field names – the PDF specification unfortunately doesn't allow this)
- description and author of file attachments
- CJK text on page descriptions, provided a Unicode-compatible encoding is used (see Section 3.3.7, »CID Font Support for Japanese, Chinese, and Korean Text«)
- 8-bit code pages for TrueType and PostScript fonts

Before delving into the Unicode implementation, however, you should be aware of the following restrictions regarding Unicode support in Acrobat:

- Acrobat 4 does not display all characters from the Adobe Glyph List correctly (this has been fixed in Acrobat 5). This bug affects, for example, Cyrillic characters
- The usability of Unicode-enhanced PDF documents 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 and MacOS support Unicode internally, availability of appropriate Unicode fonts is still an issue.
- Acrobat on Windows is unable to handle more than one script in a single annotation. This seems to be related to an OS-specific issue (restrictions of the text edit widget used in Acrobat's implementation of the annotation feature).

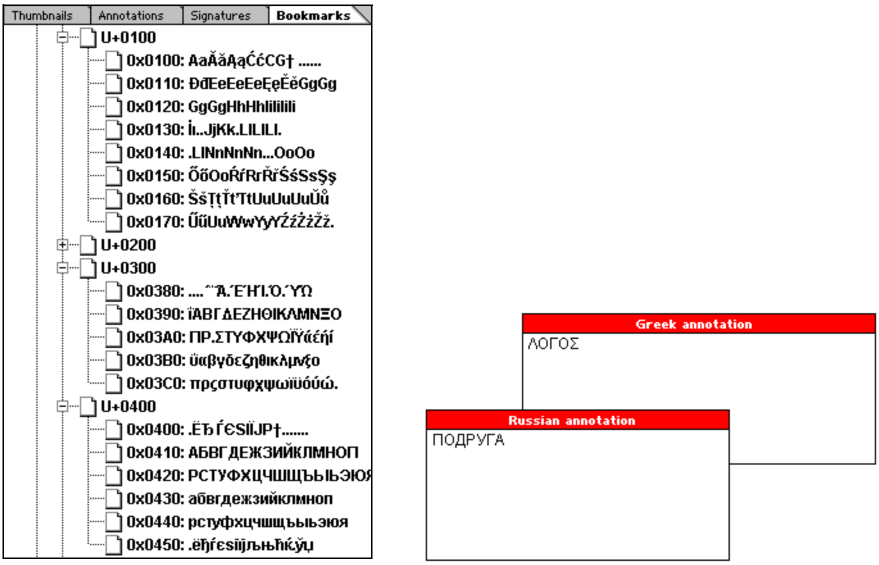
Unicode code pages for PostScript and TrueType fonts. PDFlib supports Unicode for page descriptions for characters within the Adobe Glyph List (AGL). While text strings still must contain 8-bit characters, an arbitrary set of up to 256 characters can be selected using a Unicode-based code page definition file. This kind of Unicode support is available for Unicode-based TrueType fonts and PostScript with glyph names in the AGL. For details on code pages and AGL see Section 3.3.3, »Custom Encoding and Code Page Files for 8-Bit Encodings«.

Unicode encoding for CID fonts. PDF allows Unicode-encoded text on document pages (as opposed to hypertext as discussed above). Unfortunately, this holds only true for CID fonts, but not regular Type 1 PostScript fonts. In order to place Unicode-conforming Chinese, Japanese, or Korean text on a page, a Unicode-compatible CMap must be used. These are easily identified by the *Uni* prefix in their name (see Table 3.10). These CMaps, however, only support the characters required for the respective locale, but not other Unicode characters.

Unicode text on page descriptions must be supplied »as is«, i.e., it must not be wrapped with BOM and double-null like hypertext. In addition, clients of the C and C++ language bindings (except when the ANSI string class is used in the latter case) must take care not to use the standard text functions *(PDF_show( )*, *PDF_show_xy( )*, and *PDF_continue_text( ))* when the text may contain embedded null characters. In such cases the alternate functions *PDF_show2( )* etc. must be used. 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 encoding for hypertext elements. PDFlib supports a dual-encoding approach with respect to all text supplied by the client for one of the Unicode-enabled hypertext

[1] See http://www.unicode.org for more information about the Unicode standard



- U+0100
  - 0x0100: AaĂăĄąĆćCG† ......
  - 0x0110: ĐđEeEeEeĘęĚěGgGg
  - 0x0120: GgGgHhHhIiIiIiIi
  - 0x0130: İı..JjKk.LlLlLl.
  - 0x0140: .LlNnNnNn...OoOo
  - 0x0150: ŐőOoŔŕRrŘřŚśSsŞş
  - 0x0160: ŠšŢţŤťTtUuUuUuŮů
  - 0x0170: ŰűUuWwYyŸŹźŻżŽž.
- U+0200
- U+0300
  - 0x0380: ....΄΅Ά.ΈΉΊ.Ό.ΎΏ
  - 0x0390: ΐΑΒΓΔΕΖΗΘΙΚΛΜΝΞΟ
  - 0x03A0: ΠΡ.ΣΤΥΦΧΨΩΪΫάέήί
  - 0x03B0: ΰαβγδεζηθικλμνξο
  - 0x03C0: πρςστυφχψωϊϋόύώ.
- U+0400
  - 0x0400: .ЁЂЃЄЅІЇЈР†.......
  - 0x0410: АБВГДЕЖЗИЙКЛМНОП
  - 0x0420: РСТУФХЦЧШЩЪЫЬЭЮЯ
  - 0x0430: абвгдежзийклмноп
  - 0x0440: рстуфхцчшщъыьэюя
  - 0x0450: .ёђѓєѕіїјљњћќўџ

Greek annotation

ΛΟΓΟΣ

Russian annotation

ПОДРУГА

Fig. 3.2. Unicode bookmarks (left) and Unicode text annotations (right)

functions (bookmarks, annotations, etc.). PDF expects Unicode hypertext according to the following rules (these are also known as big-endian UTF-16 serialization with signature):

- In order to distinguish »regular« 8-bit encoded text strings from 16-bit Unicode strings, the Unicode Byte Order Mark (BOM) is used as a sentinel at the beginning of the string. The BOM consists of the following two byte values which must be the first 16-bit character in all Unicode strings for hypertext:

  ```
  hex FE, FF = octal 376, 377
  ```

- Subsequent characters in the Unicode string are encoded with 2 bytes each, where the high order byte occurs first in the linear ordering (big-endian byte ordering, unlike the little-endian ordering used on Windows/Intel systems).
- Since Unicode strings may contain null characters, the usual C convention for strings cannot be used. For this reason, all non-Unicode-aware PDFlib language bindings (e.g., the C and C++ language bindings) expect Unicode strings to be terminated with a Unicode null character, i.e., two null bytes.

For example, the following string (in octal notation) encodes the Greek string »ΛΟΓΟΣ« (see Figure 3.2):

```
\0376\0377\003\233\003\237\003\223\003\237\003\243\0\0
```

or in hexadecimal notation:

```
\xFE\xFF\x03\x9B\x03\x9F\x03\x93\x03\x9F\x03\xA3\0\0
```

Clients of non-Unicode-aware language bindings (see below) must manually wrap Unicode hypertext with BOM and double-null as described above.

Wrong Unicode character assignments on Windows. The following PDFlib language bindings are Unicode-aware, and can automatically convert Unicode strings to the format expected by PDFlib:

- ActiveX/COM
- Java
- Tcl (requires Tcl 8.2 or above)

However, in order to avoid the character conversion problem described below, Unicode support is disabled by default in these bindings. It can be activated by setting the PDFlib parameter *nativeunicode* to *true* (see also Section 4.3.2, »Text Output«):

```
p.set_parameter("nativeunicode", "true");
```

Native Unicode mode means that the wrapper code will internally distinguish the following cases, and apply the appropriate conversion:

- 8-bit strings, i.e., strings which contain only characters from U+0000 to U+00FF are interpreted as PDFDocEncoding (for hypertext) or 8-bit characters according to the current encoding (for page descriptions).
- Unicode strings for hypertext functions will be encoded according to the PDF reference (wrapped with BOM and double-null).
- Unicode strings for page descriptions will be supplied without any conversion. This requires a Unicode-compatible CMap to be selected (see Table 3.9).

The developer generally need not care about the encoding specifics detailed above, but can simply use Unicode text as supported by the environment. (More details on Unicode usage from within the supported languages can be found in the manual section for the respective binding in Chapter 2). However, there’s a subtle issue related to literal Unicode characters embedded in ActiveX, Java, or Tcl source code which we will try to explain with a small example.

Java’s native support for Unicode strings is just fine for PDF’s hypertext elements, but can be dangerous with respect to page descriptions and non-Unicode-compliant 8-bit encodings. For example, while most characters in the Windows code page 1252 are compatible with Unicode, not all are (more specifically, the range 0x80-0x9F). Consider the following attempt to show the endash character with PDFlib’s Java binding:

```
// Literal character 0x96 = Alt-150 in the code. Works only if nativeunicode == false
p.show("–");
```

When this snippet is compiled under Unix with the Latin-1 character set (which is fully Unicode-compatible), it will work just fine. However, when it is compiled under Windows with code page 1252 and *nativeunicode* == *true*, the literal endash character (0x96 in code page 1252) will be translated to the corresponding Unicode character (0x2013 in this example), which is unsuited for an 8-bit PDF encoding such as *winansi*. In order to prevent this problem in native Unicode mode rewrite the above code snippet as follows:

```
// Safe way of selecting characters outside Latin-1 if nativeunicode == true
p.show("\u0096");
```

This will pass the intended character code 0x96 to PDFlib, which will correctly interpret it according to the chosen encoding vector (although the Java compiler will be fooled into believing it deals with the *Unicode* character 0x0096, which doesn’t actually exist).

3.3.9 Text Metrics, Text Variations, and Text Box Formatting

Font and 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 are supplied in the font metrics file, and can be queried from PDFlib as follows:

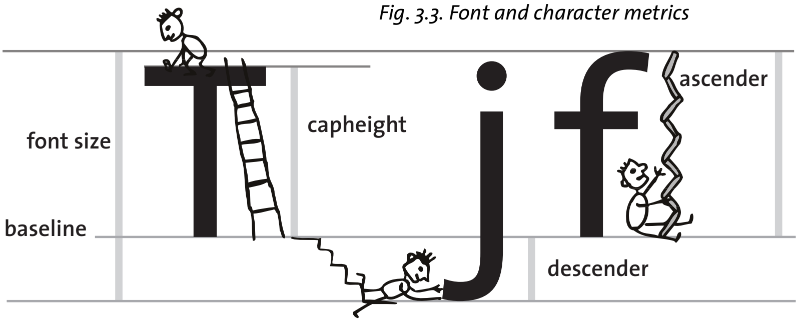
```
float capheight, ascender, descender, fontsize;
...
font = PDF_findfont(p, "Times-Roman", "host", 0);
PDF_setfont(p, font, fontsize);

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 since this information is not contained in AFM metrics files.

CPI calculations. While most fonts have varying character widths, so-called mono-spaced 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 environments, 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
```

Fig. 3.3. Font and character metrics



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.

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 CID fonts.

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 3.11. They can be set with *PDF_set_value()*.

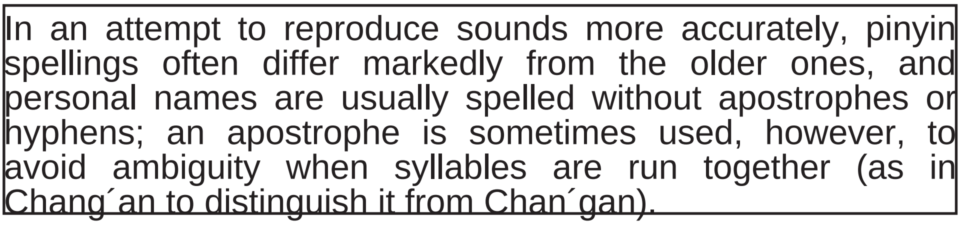
Table 3.11. Values for the text rendering mode

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

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

Text color. Text will usually be display in the current fill color, which can be set using *PDF_set_color()*. 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.

Text 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 rea-



In an attempt to reproduce sounds more accurately, pinyin spellings often differ markedly from the older ones, and personal names are usually spelled without apostrophes or hyphens; an apostrophe is sometimes used, however, to avoid ambiguity when syllables are run together (as in Chang´an to distinguish it from Chan´gan).

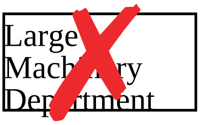


Fig. 3.4. Top: Text box formatting: the bottom edge will serve as the last baseline, not as a clipping border. Right: text box formatting doesn't work if only a single word fits on a line. In the situation in the figure to the right, PDF_ show_boxed( ) will not actually format any text.

son, descenders of the last text line may appear outside the specified box (see Figure 3.4).

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.

ASCII newline characters *(0x0A)* 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_findfont(p, "Helvetica", "host", 0);
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:

- 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. In particular, if only a single word fits in a column, *PDF_show_boxed( )* will not format any text at all, but leave the column empty (see Figure 3.4).
- Since the bottom part of the box is used as a baseline, descenders in the last line may extend beyond the box area.
- Using *PDF_show_boxed()* with top-down coordinates isn't exactly intuitive. Please review the information in Section 3.2.1, »Coordinate Systems«.
- 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 not supported for CID fonts.

3.4 Image Handling

3.4.1 Supported Image File Formats

Embedding raster images in the generated PDF is an important feature of PDFlib. PDFlib currently deals with the image file formats described below. For most formats PDFlib passes the compressed image data unchanged to the PDF output since PDF internally supports most compression schemes used in 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 and you need to know more details about the reason set the *imagewarning* parameter to *true* (see Section 4.6, »Image Functions« for more details):

```
PDF_set_parameter(p, "imagewarning", "true");          /* enable image warnings */
```

PNG images. PDFlib supports all flavors of PNG images (Portable Network Graphics).[1] 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 3.4.5, »Image Masks and Transparency«). 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 images:

- The »baseline« JPEG flavor which accounts for the vast majority of JPEG images.
- Progressive JPEG compression which is supported since Acrobat 4/PDF 1.3. If run in Acrobat 3 compatibility mode PDFlib will refuse to import progressive JPEGs.

[1] *See http://www.w3.org/Graphics/PNG and http://www.libpng.org/pub/png*

PDFlib applies a workaround which is necessary to correctly process Photoshop-generated CMYK JPEG files.

GIF images. GIF images are always handled in pass-through mode (PDFlib does not use 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«, choosing 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.
- Only the first image of a multi-frame (animated) GIF image will be imported.

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, and PackBits (=RunLength) are handled in pass-through mode; other compression schemes are handled by uncompressing.
- color depth: black and white, grayscale, RGB, and CMYK images; any alpha channel which may be present in the file is ignored.
- TIFF files containing more than one image (see Section 3.4.6, »Multi-Page Image Files«)
- 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 image selection tool. Some TIFF features (e.g., CIE color space, JPEG compression) and certain combinations of features (e.g., LZW compression and alpha channel, LZW compression and tiling) are not supported.

Note *Converting certain flavors of CCITT group 3 compressed TIFF images with PDFlib may trigger the message »Read less image data than expected« in Acrobat 4. Since the problem does not exist in Ghostscript or Acrobat 5, and the image displays just fine despite the error message, we consider this a bug in Acrobat 4. You may be able to work around it by choosing a different TIFF compression scheme.*

CCITT images. Raw 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.

Raw data. Uncompressed (raw) image data may be useful for some special applications, e.g., constructing a color ramp directly in memory. 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.

3.4.2 Code Fragments for Common Image Tasks

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_open_image_file( )* function returns a handle which serves as an image descriptor. This handle can be used in a call to *PDF_place_image( )*, along with positioning and scaling parameters:

```
if ((image = PDF_open_image_file(p, "jpeg", "image.jpg", "", 0)) == -1) {
        fprintf(stderr,"Error: Couldn't read image file.\n");
} else {
        PDF_place_image(p, image, (float) 0.0, (float) 0.0, (float) 1.0);
        PDF_close_image(p, image);
}
```

The call to *PDF_close_image( )* may or may not be required, depending on whether the same image will be used again in the same document (see Section 3.4.3, »Re-using Image Data«).

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. 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).

Resolution (dpi) values which may be contained in the original image file are ignored by PDFlib, but may be queried via the *resx* and *resy* parameters; the user is responsible for scaling the coordinate system appropriately (beware of non-square pixels). The following algorithm may be used to import an image at the resolution given in the file (or at 72 dpi if the image file doesn't contain any dpi value), and place it on the full page:

```
/* query the dpi values which may be present in the image file */
dpi_x = PDF_get_value(p, "resx", image);
dpi_y = PDF_get_value(p, "resy", image);

/* calculate scaling factors from the dpi values, see description of resx/resy */
if (dpi_x > 0 && dpi_y > 0) {               /* resx and resy are specified in the file */
        scale_x = ((float) 72.0) / dpi_x;
        scale_y = ((float) 72.0) / dpi_y;
} else if (dpi_x < 0 && dpi_y < 0) {        /* only the ratio of resx and resy is known*/
        scale_x = (float) 1.0;
        scale_y = dpi_y / dpi_x;
} else {                                    /* no information about resx and resy av. */
```

```
        scale_x = (float) 1.0;
        scale_y = (float) 1.0;
}

/* create a new page such that the scaled image exactly fits, and place the image */
PDF_begin_page(p, PDF_get_value(p, "imagewidth", image) * scale_x,
                  PDF_get_value(p, "imageheight", image) * scale_y);
PDF_scale(p, scale_x, scale_y);
PDF_place_image(p, image, 0.0, 0.0, 1.0);
PDF_close_image(p, image);
PDF_end_page(p);
```

In order to ignore any dpi value present in the image, and use a fixed dpi value instead (e.g. 300) replace the first two lines in the above code fragment with

```
dpi_x = 300;
dpi_y = 300;        /* or whatever you want */
```

Forcing printed image size. In order to place an image on a PDF page such that it results in a specified target *width* and *height* (as opposed to specifying the resolution values as in the previous algorithm) with a lower left corner at *(x, y)* (all coordinates in points) the following algorithm may be used:

```
scale_x = width/PDF_get_value(p, "imagewidth", image);
scale_y = height/PDF_get_value(p, "imageheight", image);

PDF_save(p);

/* scale the coordinate system to match the image size to the given rectangle */
PDF_scale(p, scale_x, scale_y);

/* in the positioning coordinates we must compensate for the above scaling */
PDF_place_image(p, image, x/scale_x, y/scale_y, 1);
PDF_close_image(p, image);
PDF_restore(p);
```

Non-proportional image scaling. Since in most cases images will be scaled proportionally (i.e., using the same scaling factor in both dimensions), *PDF_place_image( )* supports only a single scaling parameters which is applied to both dimensions. Non-proportional scaling can easily be achieved by scaling the coordinate system, bracketed with save/restore in order to not disturb other graphics operations. The following sequence will place an image, scaled to 50 percent horizontally and 75 percent vertically:

```
PDF_save(p);                    /* save the original coordinate system */
PDF_scale(p, 0.5, 0.75);        /* scale the coordinates, and therefore the image */
PDF_place_image(p, image, 0.0, 0.0, 1.0);
PDF_restore(p);                 /* restore the original coordinate system */
```

Remember that the *x* and *y* positions supplied to *PDF_place_image( )* will also be subject to the *PDF_scale( )* call, and must be adjusted by dividing by the scaling factors.

A code fragment for placing images in a top-down coordinate system can be found in Section 3.2.1, »Coordinate Systems«.

3.4.3 Re-using Image Data

It should be emphasized that PDFlib supports an important PDF optimization technique for using repeated raster images.

Consider a layout with a constant logo or background on several pages. In this situation it is possible to include the image data only once in the PDF, and generate only a reference on each of the pages where the image is used. Simply open the image file and call *PDF_place_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.

3.4.4 Memory Images and External Image References

While the majority of image data for use with PDFlib will be pulled from some disk file on the local file system, other image data sources are also supported. For performance reasons supplying existing image data directly in memory may be preferable over opening a disk file. PDFlib supports in-core image data for certain image file formats.

PDFlib also supports an experimental feature which isn't recommended for general-use PDF files, but may offer advantages in certain environments. While almost all PDF documents are completely self-contained (the only exception being non-embedded fonts), it is also possible to store only a reference to some external data source in the PDF file instead of the actual image data, and rely on Acrobat to fetch the required image data when needed. This mechanism works similar to the well-known image references in HTML documents. Usable external image sources include data files in the local file system, and URLs. It is important to note that while file references work in Acrobat 3 and 4, URL references only work in Acrobat 4 or above (full product). PDF documents which include image URLs are neither usable in Acrobat 3 nor Acrobat Reader 4!

The *PDF_open_image()* interface can be used for both in-memory image data and external references.

3.4.5 Image Masks and Transparency

Transparency in PDF. Transparency has been missing from PostScript and PDF for quite a long time. Only with PDF 1.3 (and PostScript 3) Adobe integrated some limited support for transparency into languages and applications. While image masks (painting solid color through a bitmap mask) are an old feature of both PostScript and PDF, Acrobat 4 added the feature of masking particular pixels of an image. This offers the following opportunities:

- Masking by position: an image may carry the intrinsic information »print the foreground only, but not the background«. This is often used in catalog images.
- Masking by color value: pixels of a certain color (or from a color range – but not arbitrary sets of colors) are not painted, but the previously painted part of the page shines through instead. 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.

It is important to note that PDF supports binary transparency only: there is no alpha channel or variable opacity (»blend this image with the background«) but only a binary decision (»print either the image pixel, or the background pixel«). Binary transparency may be considered »poor man's alpha channel«. Another important restriction is that in PDF the mask is always attached to the image; it's not possible to use an image first with a mask, and the same image a second time without a mask, or with a different mask.

Viewing and printing PDF files with transparency. Equally important as PDF's intrinsic limitations with respect to transparency are the practical limitations when it comes to using PDF files with transparency in the viewer application. The following restrictions should be noted:

- Transparency only works in PDF 1.3/Acrobat 4 and above – older viewers will completely ignore transparency information, and display or print the whole image (overpainting the background).
- Printing transparent images to PostScript Level 1 or 2 doesn't work, even with Acrobat 4 (since transparency support only appeared in PostScript 3, and can't easily be emulated). Acrobat prints the base image without the mask.
- If an image is masked by position Acrobat 4 viewers will only honour the clipping up to a certain image size, and display the whole image otherwise. It appears from experimentation that the following limit applies to Acrobat 4 (Acrobat 5 is not affected by this limit):

  ```
  width x height x components < 1024 K
  ```

  Images above this limit are displayed without applying the mask. The limit in a typical PostScript 3 printer seems to be lower, resulting in PostScript errors when trying to print PDF documents with large masked images.

Transparency support in PDFlib. PDFlib supports both masking by position and by color value (only single color values, but no ranges). Transparency information can be applied implicitly or explicitly. Masked images are not supported in Acrobat 3 compatibility mode.

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 tries to preserve as much as possible from this information: single transparent color values are retained; if multiple color values 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.

The explicit case requires two steps, both of which involve image operations. First, an image must be prepared for later use as a binary transparency mask. This is accomplished by using the standard image file function with an additional parameter:

```
mask = PDF_open_image_file(p, "png", maskfilename, "mask", 0);
```

In order to be usable as a mask, an image must have only a single color component and a bit depth of 1, i.e., only plain bitmaps are suitable as a mask. Only PNG and in-memory images are supported for constructing a mask. Pixel values of 0 in the mask will result in the corresponding area of the image being painted, while pixel values of 1 result in the background shining through.

In the second step this mask is applied to another image which itself is acquired through one of the usual image functions:

```
image = PDF_open_image_file(p, type, filename, "masked", mask)
if (image != -1) {
        PDF_place_image(p, image, x, y, scale);
} else {
        ...
}
```

Note the different use of the optional string parameter for *PDF_open_image_file( )*: *mask* for defining a mask, and *masked* for applying a mask to another image. The integer parameter is unused in the first step, and carries the mask descriptor in the second step.

The image and the mask may have different pixel dimensions; the mask will automatically be scaled to the image's size.

PDFlib doesn't make any provisions for painting solid color through a mask (like PostScript's *imagemask* operator), since this is a special case of the general masking mechanism. You can achieve this effect by applying the required mask to an auxiliary image constructed in memory with *PDF_open_image( )* (a solid rectangle of the requested color).

Note Multi-strip TIFF images are converted to multiple PDF images, which would be masked individually by PDFlib. Since this is usually not intended, this kind of images should be avoided as mask target. Also, it is important to not mix the implicit and explicit cases, i.e., don't use images with transparent color values as mask.

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 solve this problem, PDFlib's automatic transparency support can be disabled with the *ignoremask* parameter when opening the file:

```
image = PDF_open_image_file(p, "gif", filename, "ignoremask", 0);
```

3.4.6 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, the call to *PDF_open_image_file()* additional string and numerical parameters are used:

```
image = PDF_open_image_file(p, "tiff", filename, "page", 1);
```

The *page* parameter indicates that a multi-image file is to be used, and is only supported for TIFF images. The last parameter specifies the number of the image to use. The first image is numbered 1. This parameter may be increased until *PDF_open_image_file()* 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++) {
        image = PDF_open_image_file(p, "tiff", filename, "page", frame);
        if (image == -1)
            break;
        PDF_begin_page(p, width, height);
        PDF_place_image(p, image, 0.0, 0.0, 1.0);
        PDF_close_image(p, image);
        PDF_end_page(p);
}
```

3.5 PDF Import with PDI

Note All functions described in this section require the additional PDF import library (PDI) which is not part of the PDFlib source code distribution. Please visit our Web site for more information on obtaining PDI.

3.5.1 PDI Features and Applications

When the optional PDI (PDF import) library is attached to PDFlib, pages from existing PDF documents can be processed with all supported language bindings. The PDI product 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:

- place a PDF background page and populate it with dynamic data (e.g., mail merge, personalized PDF documents on the Web, form filling)
- 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
- 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

3.5.2 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, bookmarks, thumbnails, and notes) which may be present in the imported PDF document. These hypertext features can be generated with the corresponding PDFlib functions. Similarly, 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 pages 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.

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 template feature, restrictions in certain environments other than Acrobat may apply (see Section 3.2.4, »Templates«).

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;
float   sheetwidth, sheetheight;
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);
}
sheetwidth = PDF_get_pdi_value(p, "width", doc, page, 0);
sheetheight = PDF_get_pdi_value(p, "height", doc, page, 0);

PDF_begin_page(p, sheetwidth, sheetheight);
PDF_place_pdi_page(p, page, 0.0, 0.0, 1.0, 1.0);
PDF_close_pdi_page(p, page);
...add more content to the page using PDFlib functions...
PDF_end_page(p);
```

The PDFlib distribution contains PDI examples for all supported language bindings which demonstrate various applications of PDI features:

- The personalization demo pulls a page from an existing PDF document, and places additional text on the page.
- The quick reference demo extracts several pages from an existing PDF document, scales down the pages, and places multiple pages on an output sheet.
- The imposition demo (which is only available in C code) is a generalization of the quick reference demo. It processes an arbitrary number of PDF documents, and places *n x m* pages on an output sheet. In addition, lines are drawn around the scaled-down pages.

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_place_pdi_page( )*. 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 and Coordinate Limits«);
- rotation which has been applied to the page will be retained.

Many important properties, such as the page size of an imported PDF page and the number of pages in a document, can be queried via PDFlib's parameter mechanism. The relevant parameters are listed in Table 4.15 and Table 4.15. These properties can be useful in making decisions about the placement of imported PDF pages on the output page. The algorithms presented in Section 3.4.2, »Code Fragments for Common Image Tasks« for images can be used for scaling imported PDF pages as well.

Imported PDF pages and the graphics state. PDFlib treats imported PDF pages as templates. For this reason the comments in Section 3.2.4, »Templates« also apply to imported PDF pages which are placed on the output page. In particular, imported pages may change their appearance if the surrounding page changes some graphics state parameter which the imported page doesn't explicitly set. You can use *PDF_initgraphics( )* to avoid this behavior.

Dealing with Acrobat 5/PDF 1.4 files. PDI is fully compatible to PDF 1.4 files generated with Acrobat 5. However, you should be aware of the following: Imported PDF documents must not have a higher PDF version number than the generated PDF output. Since the default output mode in PDFlib 4 is Acrobat 4/PDF 1.3, imported PDF 1.4 files will be rejected by default. You can modify this behavior by changing PDFlib's PDF output version as follows:

```
PDF_set_parameter(p, "compatibility", "1.4")
```

This will result in Acrobat 5/PDF 1.4 compatible output, which in turn allows you to also import files according to PDF 1.4.

3.5.3 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. However, in rare cases a PDF document or a particular page of a document may be rejected by PDI. 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.
- Encrypted PDF documents (i.e., any security settings applied). Solution: remove all security settings in Acrobat and resave the document. Obviously, you will need the document's password to do so.

- Since PDFlib/PDI do not contain any implementation of the LZW algorithm, certain PDF pages which use LZW compression (more specifically, LZW-compressed pages with multiple content streams) will be rejected *(unsupported filter)*. Solution: resave the document in Acrobat 4.0 or above, with the *Optimize* button checked. Note that Acrobat 4.0 and above will never generate this kind of offended file, but only Acrobat 3 under certain circumstances. For this reason you are unlikely to run into this restriction. If you have a large number of such files which must be converted you should look into Acrobat's batch optimization feature.

Depending on the *pdiwarning* parameter, unacceptable PDF files will simply result in an error return value, or a nonfatal exception with a detailed explanation.

4 PDFlib API Reference

The API reference documents all supported PDFlib functions. A few functions are not supported in certain language bindings since they are not necessary. These cases are mentioned in appropriate notes.

4.1 Data Types, Naming Conventions, and Scope

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 4.1 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 4.1. 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 or const char ** [2]	*char **
Java	*no*	*no*	*String*	*byte[ ]*
Perl	*yes*	*yes*	*string*	*string*
PHP	*yes*	*yes*	*string*	*string*
Python	*yes*	*yes*	*string*	*string*
Tcl	*yes*	*yes*	*string*	*byte array*

[1] C language NULL string values and empty strings are considered equivalent.
*[2] ANSI C++ strings or C-style char * are used according to a compiler-dependent preprocessor definition. NULL string values must not be used in the C++ binding.*

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);
```

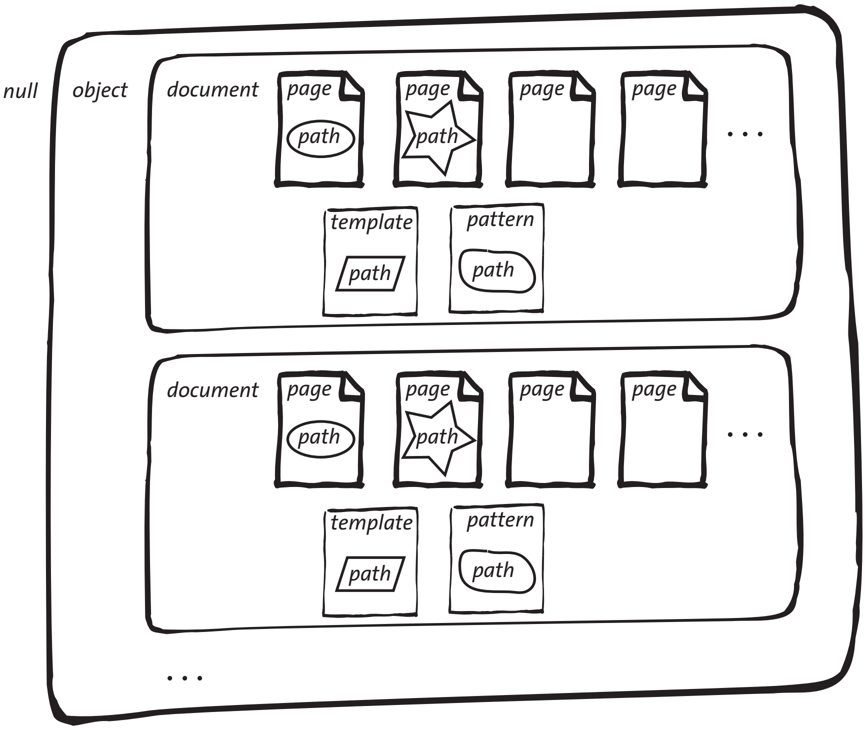


Fig. 4.1.
Relationship of scopes

Function scopes. Most PDFlib functions are subject to certain ordering and nesting constraints which are derived from their contribution to the generated document. Most of these constraints are rather obvious. For example, you must begin a page before you can close it. In the same spirit, the functions for opening a PDF document and closing it must always be paired. PDFlib uses a strict scoping system for defining and verifying the correct ordering of functions used by client programs. The function descriptions reference these scopes; the scope definitions can be found in Table 4.2. Figure 4.1 depicts the relationship of scopes. PDFlib will throw an exception if a function is called outside the allowed scope.

Table 4.2. 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 4.4.4, »Path Painting and Clipping«*
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*
document	*between PDF_open_*( ) and PDF_close( ), but outside of page, template, and pattern 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*

4.2 General Functions

4.2.1 Setup

Table 4.3 lists relevant parameters and values for this section.

Table 4.3. Parameters and values for the setup functions

function	*key*	*explanation*
set_parameter	*compatibility*	*Set PDFlib's compatibility mode to one of the strings »1.2«, »1.3«, or »1.4« for Acrobat 3, 4, or 5. The default is »1.3«. This parameter must be set before the first call to PDF_open_*( ). Setting compatibility to »1.2« will make Acrobat 4 features unavailable. Strict Acrobat 3 compatibility mode is not required for generating Acrobat 3 compatible files, but only under very specific circumstances related to PDF-enabled RIPs (see Section 1.3, »PDFlib Output and Compatibility«).*
set_parameter	*flush*	*Set PDFlib's flushing strategy to none, page, content, or heavy. The default is page. See Section 3.1.3, »Generating PDF Documents directly in Memory« for details. This parameter is only available in the C and C++ language bindings.*
set_parameter	*prefix*	*Resource file name prefix as used in a UPR file (see Section 3.3.6, »Resource Configuration and the UPR Resource File«). The prefix can only be set once. It contains a slash character plus a path name, which in turn may start with a slash.*
set_parameter	*resourcefile*	*Relative or absolute file name of the PDFlib UPR resource file. The resource file will be loaded at the next attempt to access resources. The resource file name can only be set once. This call should occur before the first page.*
set_parameter	*serial*	*Set the PDFlib and/or PDI serial string. The serial string can only be set once before the first call to PDF_begin_page( )*
set_parameter	*warning*	*Enable or suppress warnings (nonfatal exceptions). Possible values are true and false, default value is true.*
set_value	*compress*	*Set the compression parameter to a value from 0–9. This parameter does not affect precompressed image data which is handled in pass-through mode.* *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.*
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.*

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.

int PDF_get_majorversion(void)
int PDF_get_minorversion(void)

Return the PDFlib major and minor version number, respectively.

Scope any

Params *major, minor, revision, version*

Bindings Java, Perl, and Tcl: the functions are not available; use *PDF_get_value(p, "major", 0)* instead.

PDF *PDF_new(void)

Create a new PDF object with default settings.

Details This function creates a new PDF object, using PDFlib's internal default error handling and memory allocation routines.

Returns A handle to a PDF object which is to be used in subsequent PDFlib calls. The contents of the PDF structure are considered private to PDFlib; only pointers to the PDF structure are used at the API level.

This function does not return any error code. If it doesn't succeed due to unavailable memory, a PDFlib exception is raised.

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 PDF 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++: this function is not available since it is hidden in the PDF constructor.
ActiveX, Java: this function is automatically called by the wrapper code, and therefore not available.

PDF *PDF_new2(
void (*errorhandler)(PDF *p, int type, 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 PDF object with client-supplied error handling and memory allocation routines.

errorhandler Pointer to a user-supplied error-handling function.

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 pointer to the opaque PDF data type which is required as the *p* argument for all other functions.

Details This function creates a new PDF 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.

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.

Other bindings: this function is automatically called by the wrapper code, and therefore not available.

void PDF_delete(PDF *p)

Delete a PDF 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 from client-supplied error handlers for cleanup. 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 *object* or user-defined error handler (C and C++ only); this function terminates object scope, and must always be paired with a matching call to one of the *PDF_new( )* or *PDF_new2 ( )* functions.

Bindings C++: this function is indirectly available via the PDF destructor.

Java: this function is automatically called by the wrapper code, and therefore not available.

void *PDF_get_opaque(PDF *p)

Fetch 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 PDF object.

Scope *any*

Bindings Only available in the C and C++ bindings.

4.2.2 Document and Page

Table 4.4 lists relevant parameters and values for this section.

Table 4.4. Parameters and values for the document and page functions

function	***key***	***explanation***
set_value	*pagewidth* *pageheight*	*Change the page size dimensions of the current page. These parameters must only be used within a page description.*
set_value	*CropBox,* *BleedBox,* *ArtBox,* *TrimBox*	*Change one of the box parameters of the current page. These parameters must only be used within a page description. 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 and Coordinate Limits« for details)*

int PDF_open_file(PDF *p, const char *filename)

Create a new PDF file using the supplied file name.

filename Name of the PDF output file to be generated. If *filename* is empty the PDF document will be generated in memory instead of on file. The result 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 MacOS).

Returns -1 (in PHP: 0) on error, and 1 otherwise.

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.

Bindings C++: this function is hidden in the overloaded open( ) call.

int PDF_open_fp(PDF *p, FILE *fp)

Open a new PDF file associated with p, using the supplied file handle.

fp Pointer to a C-style FILE structure to which the generated output PDF will be written. On Mac, Windows, and AS/400 the *fp* file handle must have been opened in binary mode, which is necessary for PDF output. On Windows PDFlib changes the output mode of the supplied file handle to binary mode itself.

Returns -1 on error, and 1 otherwise.

Scope *object;* this function starts document scope if *fp* is not NULL, and must always be paired with a matching *PDF_close( )* call.

Bindings This function is only available in the C and C++ bindings.

C++: this function is hidden in the overloaded *open( )* call.

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 3.1.3, »Generating PDF Documents directly in Memory« 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++: this function is hidden in the overloaded open( ) call. *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 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 4.1. The returned buffer can be used until the end of the surrounding *object* scope.

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«. A list of commonly used page formats can be found in Table 4.24 in Section 4.9, »Page Size Formats«. 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 4.3).

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.

4.2.3 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.

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.

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*.

4.3 Text Functions

4.3.1 Font Handling

Table 4.5 lists relevant parameters and values for this section.

Table 4.5. Parameters and values for the font functions (see Section 4.2.3, »Parameter Handling«)

function	*key*	*explanation*
set_parameter	*FontAFM* *FontPFM* *FontOutline* *Encoding*	*The corresponding resource file line as it would appear for the respective category in a UPR file (see Section 3.3.6, »Resource Configuration and the UPR Resource File«). Multiple calls add new entries to the internal list. (See also prefix and resourcefile in Table 4.3)*
get_value	*font*	*Return the identifier of the current font which must have been set with PDF_setfont( ). Scope: page, pattern, template.*
get_parameter	*fontname*	*The name of the current font which must have been previously set with PDF_setfont( ). Scope: page, pattern, template.*
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.*
get_value	*fontsize*	*Return the size of the current font which must have been previously set with PDF_setfont( ). Scope: page, pattern, template.*
get_value	*capheight* *ascender* *descender*	*Return metrics information for the font identified by the modifier. See Section 3.3.9, »Text Metrics, Text Variations, and Text Box Formatting« for more details. The values are measured in fractions of the font size, and must therefore be multiplied by the desired font size.*
set_parameter	*fontwarning*	*If set to false, PDF_findfont( ) returns -1 (in PHP: 0) if the font/encoding combination cannot be loaded (instead of throwing an exception). Default is true.*

int PDF_findfont(PDF *p, const char *fontname, const char *encoding, int embed)

Search for a font, and prepare it for later use.

fontname The name of the font as configured in PDFlib.

encoding For 8-bit fonts, *encoding* is one of *builtin*, *macroman*, *winansi*, *ebcdic*, or *host* (see Section 3.3.2, »8-Bit Encodings built into PDFlib«), or the name of an external PDFlib-supplied or user-defined encoding (see Section 3.3.3, »Custom Encoding and Code Page Files for 8-Bit Encodings«). Note that in order to use arbitrary encodings, you will need metrics information for the font (see Section 3.3.5, »PostScript and TrueType Fonts«).

Alternatively, *encoding* can be the name of one of the built-in CMaps if *fontname* describes a CID font (see Section 3.3.7, »CID Font Support for Japanese, Chinese, and Korean Text«). In this case metrics information is not required. Case is significant for both *fontname* and *encoding*.

embed If 0 (zero), only general font information (a font descriptor) is included in the PDF output. If *1*, the font outline file must be available in addition to the metrics information, and the actual font definition will be embedded in the PDF output. However, the font file will only be checked when this function is called, but not yet used, since font embedding is done at the end of the generated PDF file. The *embed* parameter must be 0 for CID fonts.

Returns A font handle for later use with *PDF_setfont()*. The behavior of this function changes when the *fontwarning* parameter is set to *true*. In this case *PDF_findfont()* 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 passed.

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.

Details This function prepares a font for later use with *PDF_setfont()*. The metrics will be loaded from memory or from an external 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 of type *PDF_RuntimeError* will be raised. Otherwise, the value returned by this function can be used as font argument to other font-related functions.

CID fonts are not supported in Acrobat 3 compatibility mode.

Scope *document, page, pattern, template*

Params See Table 4.5.

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()*.

fontsize Size of the font, measured in units of the current user coordinate system.

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*

Params See Table 4.5.

4.3.2 Text Output

Note *All text supplied to the functions in this section must match the encoding selected with PDF_findfont(). This applies to 8-bit text as well as Unicode or other encodings selected via a CMap.*

Table 4.5 lists relevant parameters and values for this section. The scope for all parameters is *page, pattern,* and *template* unless otherwise noted.

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 position.

text The text to be printed.

Table 4.6. Parameters and values for the text functions (see Section 4.2.3, »Parameter Handling«)

function	*key*	*explanation*
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 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. However, ascenders and descenders of adjacent lines will generally not overlap.*
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 baseline 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.*
set_value *get_value*	*horizscaling*	*Set or get the horizontal text scaling to the given percentage, which must be greater than 0. Text scaling shrinks or expands the text by a given percentage. The text scaling is set to the default of 100 at the beginning of each page. Text scaling always relates to the horizontal coordinate.*
set_value *get_value*	*textrendering*	*Set or get the current text rendering mode to one of the values given in Table 3.11. The text rendering parameter is set to the default of 0 (= solid fill) at the beginning of each page.*
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 text space units. It is reset to the default of 0 at the beginning of each page. In order to spread the characters apart use positive values for horizontal writing mode, and negative values for vertical writing mode.*
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 fonts with multi-byte encodings don't have an ASCII space character they are not affected by the word spacing. The spacing value is given in text space units. It is reset to the default of 0 at the beginning of each page.*
get_value	*textx* *texty*	*Get the x or y coordinate, respectively, of the current text position. These parameters are currently not supported for CID fonts.*
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. Theses modes can be set independently from each other, and are reset to false at the beginning of each page (see Section 3.3.9, »Text Metrics, Text Variations, and Text Box Formatting«).* *true* *underline/overline/strikeout text (does not work for CID fonts)* *false* *do not underline/overline/strikeout text*
set_parameter	*native-unicode*	*If true, enable native Unicode text processing for language bindings with Unicode support; disable if false. Default value is false (see Section 3.3.8, »Unicode Support«). Scope: any.*

len (Only for *PDF_show2( ).)* Length of *text* (in bytes) for strings which may contain null characters. If *len = 0* a null-terminated string is assumed as in *PDF_show( )*.

Details Both font (via *PDF_setfont( ))* and current text position (via *PDF_set_text_pos( )* or some text output function*)* must have been set before. The current point is moved to the end of the printed text.

Scope *page, pattern, template*

Params See Table 4.6.

Bindings C and C++: for *PDF_show()* *text* must not contain null characters, since it is assumed to be null-terminated; use *PDF_show2()* for strings which may contain null characters.

Other bindings: *PDF_show2()* is not available since 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.

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 is assumed as in *PDF_show_xy()*.

Details The font must have been set before with *PDF_setfont()*. The current point is moved to the end of the printed text.

Scope *page, pattern, template*

Params See Table 4.6.

Bindings C and C++: for *PDF_show_xy()* *text* must not contain null characters, since it is assumed to be null-terminated; use *PDF_show_xy2()* for strings which may contain null characters.

Other bindings: *PDF_show_xy2()* is not available since 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.

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 is assumed as in *PDF_continue_text()*.

Details The positioning of text 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 is moved to the end of the printed text.

Scope *page, pattern, template;* this function should not be used in vertical writing mode.

Params See Table 4.6.

Bindings C and C++: for *PDF_continue_text()* *text* must not contain null characters, since it is assumed to be null-terminated; use *PDF_continue_text2()* for strings which may contain null characters.

Other bindings: *PDF_continue_text2( )* is not available since arbitrary string contents can be supplied with *PDF_continue_text( )*.

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.

x, y The coordinates of a 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)* (but see the description of top-down coordinates in Section 3.2.1, »Coordinate Systems«) 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, horizontal spacing, and leading are used for the text.

Scope *page, pattern, template;* this function cannot be used with CID fonts or *ebcdic* encoding.

Params See Table 4.6.

See also Restrictions of this functions are listed in Section 3.3.9, »Text Metrics, Text Variations, and Text Box Formatting«.

float PDF_stringwidth(PDF *p, const char *text, int font, float size)
float PDF_stringwidth2(PDF *p, const char *text, int len, int font, float size)

Return the width of *text* in an arbitrary font.

text The text for which the width will be queried.

len (Only for *PDF_stringwidth2( ).)* Length of *text* (in bytes) for strings which may contain null characters. If *len = 0* a null-terminated string is assumed as in *PDF_stringwidth( )*.

font A font handle returned by *PDF_findfont( )*.

size Text size, measured in units of the user coordinate system.

Details This function returns the width of *text* in an arbitrary font and size which has been selected with *PDF_findfont( )*. The width calculation takes the current values of the following text parameters into account: horizontal scaling, character spacing, and word spacing.

Scope *page, pattern, template, path, document;* this function cannot be used with CID fonts. If the current font is a CID font, it returns 0 regardless of the *text* and *size* arguments.

Params See Table 4.6.

Bindings C and C++: For *PDF_stringwidth( ) text* must not contain null characters, since *text* is assumed to be null-terminated; use *PDF_stringwidth2( )* for strings which may contain null characters.

Other bindings: *PDF_stringwidth2( )* is not available since arbitrary string contents can be supplied with *PDF_stringwidth( )*.

void PDF_set_text_pos(PDF *p, float x, float y)

Set the text output position.

x, y The current text position to be set.

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*

Params See Table 4.6.

4.4 Graphics Functions

4.4.1 General Graphics State

Note *None of the the general graphics state functions must be used during path scope (see Section 3.2, »Page Descriptions«).*

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*

void PDF_setpolydash(PDF *p, float *darray, int length)

Set a more complicated dash pattern defined by an array.

darray An array which contains alternating values for black and white dash lengths. The array values must be non-negative, and not all zero.

length The number of elements in the dash array.

Details In order to produce a solid line, choose *length* = *0* and *darray* = *NULL* or an empty array. The array length must be less than or equal to 8; otherwise the array will be truncated. The dash parameter is set to a solid line at the beginning of each page.

Scope *page, pattern, template*

Bindings C and C++: The length parameter is required.

Other bindings: Other language bindings simply supply an array as argument, and the language wrapper will automatically determine its length.

void PDF_setflat(PDF *p, float flatness)

Set the flatness parameter.

flatness 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 0 at the beginning of each page, which means that the device's default flatness is used.

Scope *page, pattern, template*

void PDF_setlinejoin(PDF *p, int linejoin)

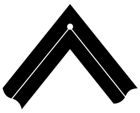
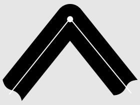
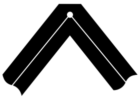
Set the linejoin parameter.

linejoin Specifies the shape at the corners of paths that are stroked, see Table 4.7.

Details The *linejoin* parameter is set to the default value of 0 at the beginning of each page.

Scope *page, pattern, template*

Table 4.7. 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)

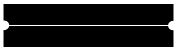
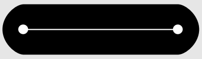
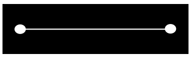
Set the linecap parameter.

linecap Controls the shape at the end of a path with respect to stroking, see Table 4.8.

Details The linecap parameter is set to the default value of 0 at the beginning of each page.

Scope *page, pattern, template*

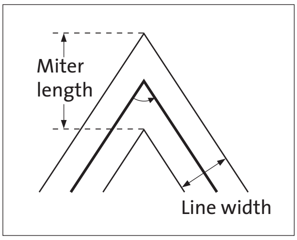
Table 4.8. Values of the linecap parameter

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.



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*

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*

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.

This function may be useful in situations where the program flow doesn't allow for easy use of *PDF_save( )/PDF_restore( ),* or for preparing the graphics state for a subsequent template or imported PDF.

Scope *page, pattern, template*

4.4.2 Special Graphics State

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 4.3, »Text 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_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;
- 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;* must always be paired with a matching *PDF_restore( )* call. *PDF_save( )* and *PDF_restore( )* calls must be balanced on each page, pattern, and template.

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*, *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;* must always be paired with a matching *PDF_save( )* call. *PDF_save( )* and *PDF_restore( )* calls must be balanced on each page, pattern, and template.

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*

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*

void PDF_rotate(PDF *p, float phi)

Rotate the user 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*

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* axes. 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*

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*

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*

4.4.3 Path Construction

Table 4.9 lists relevant parameters and values for this section.

Table 4.9. Parameters and values for the path segment functions (see Section 4.2.3, »Parameter Handling«)

function	*key*	*explanation*
get_value	*currentx* *currenty*	*The x or y coordinate, respectively, of the current point.*

Note Make sure to call one of the functions in Section 4.4.4, »Path Painting and Clipping« 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;* 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;* 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 functions, 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;* 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;* 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*

4.4.4 Path Painting and Clipping

Table 4.10 lists relevant parameters and values for this section.

Table 4.10. Parameters and values for the path painting and clipping functions (see Section 4.2.3, »Parameter Handling«)

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.*

Note *All functions in this section clear the path, and leave the current point undefined. Subsequent drawing operations must 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 and clear it.

Details This function strokes (draws) the current path with the current line width and the current stroke color.

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 *PDF_setfillrule( ))*. 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.

Details This function fills and strokes the current path with the current fill and stroke color, respectively.

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.

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 and will only rarely be useful. It generates an invisible path on the page.

Scope *path;* this function terminates *path* scope.

4.5 Color Functions

void PDF_setcolor(PDF *p,
const char *type, const char *colorspace, float c1, float c2, float c3, float c4)

Set the current color space and color.

type One of *stroke, fill*, or *both* to specify that the color is set for filling, stroking, or both filling and stroking.

colorspace One of *gray, rgb, cmyk, spot,* or *pattern* to specify the color space.

c1, c2, c3, c4 Color components for the chosen color space:
- If the *colorspace* is *gray, c1* specifies a gray value;
- If the *colorspace* is *rgb, c1, c2, c3* specify red, green, and blue values;
- If the *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( ).*

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.

Scope *page, pattern (only if the pattern's paint type is 1), template;* a pattern color can not be used within its own definition.

---

int PDF_makespotcolor(PDF *p, const char *spotname, int len)

Make a named spot color from the current fill color.

spotname An arbitrary name for the spot color to be defined. This name may contain arbitrary characters, but is restricted to a maximum length of 126 bytes.

len The length of *spotname* in bytes. If len = 0, *spotname* is assumed to be null-terminated. This parameter is only required for C and C++.

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* 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;* the current fill color must not be a spot color or pattern;

Bindings C, C++: the *len* parameter is required, but can be 0 for null-terminated strings.
Other bindings do not require the *len* parameter but simply supply a string argument.

---

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( )* throughout the document.

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.

void PDF_setgray_fill(PDF *p, float g)
void PDF_setgray_stroke(PDF *p, float g)
void PDF_setgray(PDF *p, float g)

Deprecated, use *PDF_setcolor(p, type, "gray", g, 0, 0, 0)* with a type parameter of *fill*, *stroke*, or *both* instead.

void PDF_setrgbcolor_fill(PDF *p, float red, float green, float blue)
void PDF_setrgbcolor_stroke(PDF *p, float red, float green, float blue)
void PDF_setrgbcolor(PDF *p, float red, float green, float blue)

Deprecated, use *PDF_setcolor(p, type, "rgb", red, green, blue, 0)* with a type parameter of *fill*, *stroke*, or *both* instead.

4.6 Image Functions

The functions for opening images described below can be called within or outside of page descriptions. Opening images outside a *PDF_begin_page( ) / PDF_end_page( )* context actually offers slight output size advantages.

Table 4.11 lists relevant parameters and values for this section.

Table 4.11. Parameters and values for the image functions (see Section 4.2.3, »Parameter Handling«)

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.*
get_value	*resx* *resy*	*Get the horizontal or vertical resolution of an image, respectively. The modifier is the integer handle of the selected image.* *If the value is positive, the return value is the image resolution in pixels per inch (dpi). If the return value is negative, it can be used to find the aspect ratio of non-square pixels, but doesn't have any absolute meaning. If the return value is zero, the resolution of the image is unknown.*
set_parameter	*image-warning*	*This parameter can be used in order to obtain more detailed information about why an image couldn't be opened successfully with PDF_open_image_file( ) or PDF_open_CCITT( ):* *true* *Raise a Nonfatal exception when the image function fails. The message supplied with the exception may be useful in debugging.* *false* *Do not raise an exception when the image function fails. Instead, the function returns -1 (in PHP: 0) on error. This is the default.*

int PDF_open_image_file(PDF *p,
const char *type, const char *filename, const char *stringparam, int intparam)

Open an image file.

type Specifies the format type of the image: *png*, *gif*, *jpeg*, or *tiff* (see Section 3.4.1, »Supported Image File Formats«). Case is significant for all parameters.

filename The name of the image file to be opened.

stringparam, intparam The *stringparam* and *intparam* parameters are used for additional image attributes according to Table 4.12. If *stringparam* is unused, it must be an empty string, and *intparam* must be 0.

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 to *true* (see Table 4.11). 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 file formats as determined by the *type* parameter. PDFlib will open the image file with the given name, process the contents, and close it before returning from this call. Although images can be placed multiply within a document (see *PDF_place_image( )*), the actual image file will not be kept open after this call.

Scope *document, page;* must always be paired with a matching *PDF_close_image( )* call.

Params *imagewidth, imageheight, resx, resy, imagewarning*

Table 4.12. The stringparam and intparam parameters of PDF_open_image_file( )

stringparam	*explanation and possible intparam values*
mask	*Create a mask from this image. The returned image handle may be used in subsequent calls for opening another image and can be supplied for the »masked« parameter. The intparam parameter is ignored in this case, and must be 0.*
masked	*Use the image descriptor given in intparam as a mask for this image. The intparam parameter is an image handle which has been retrieved with a previous call to PDF_open_image( ) with the »mask« parameter, and has not yet been closed.*
ignoremask	*Ignore any transparency information which may be present in the image file.*
invert	*Invert black and white for 1-bit TIFF images. This is mainly intended as a workaround for certain TIFF images which are interpreted differently by different applications.*
page	*Extract the image with the number given in intparam from a multi-page file. The first image has the number 1. This is only supported for multi-image TIFF files.*

int PDF_open_CCITT(PDF *p,
const char *filename, int width, int height, int BitReverse, int K, int BlackIs1)

Open a raw CCITT image.

filename The name of the CCITT file to be opened.

width, height The dimensions of the image in pixels.

BitReverse If 1, do a bitwise reversal of all bytes in the compressed data.

K CCITT compression parameter for encoding scheme selection.
- -1 indicates G4 encoding;
- 0 indicates one-dimensional G3 encoding (G3-1D);
- 1 indicates mixed one- and two-dimensional encoding (G3, 2-D) as supported by PDF.

BlackIs1 If this parameter has the value 1, 1-bits are interpreted as black and 0-bits as white. Most CCITT images don't use such a black-and-white reversal, i.e., most images use *BlackIs1 = 0*.

Returns An image handle which may be used in subsequent image-related calls if not -1 (in PHP: 0). Since PDFlib is unable to analyze CCITT images, all relevant image parameters have to be passed to *PDF_open_CCITT( )* by the client.

Details This function opens an image file with raw CCITT G3 or G4 compressed bitmap data (this is different from a TIFF file which contains CCITT-compressed image data!).

Scope *document, page;* must always be paired with a matching *PDF_close_image( )* call.

Params *imagewidth, imageheight, resx, resy, imagewarning*

int PDF_open_image(PDF *p, const char *type, const char *source, const char *data, long length, int width, int height, int components, int bpc, const char *params)

Use image data from a variety of data sources.

type Specifies the kind of image data or compression: *jpeg*, *ccitt*, or *raw* (see Section 3.4.1, »Supported Image File Formats«).

source, data, length The *source* parameter denotes where the image data comes from, and can attain the values *fileref*, *url*, or *memory* (see Section 3.4.4, »Memory Images and External Image References«). The relationship among the *source*, *data*, and *length* parameters is explained in Table 4.13. The *data* parameter has binary data type according to Table 4.1.

width, height The dimensions of the image in pixels.

components The number of color components must be 1, 3, or 4 corresponding to grayscale, RGB, or CMYK image data.

bpc The number of bits per component must be 1, 2, 4, or 8.

params If *components = 1* and *bpc = 1*, *params* may be *mask* in order to use this image as an image mask. Alternatively, additional CCITT parameters can be supplied (see below).

Returns An image handle which may be used in subsequent image-related calls if not -1 (in PHP: 0).

Details This versatile interface can be used to work with image data in several formats and from several data sources. Unlike *PDF_open_image_file( )* which analyzes an image file, the user must supply the *length, width, height, components,* and *bpc* parameters. *PDF_open_image( )* does not analyze the image data, and the user is responsible for supplying parameters which actually match the image properties. Otherwise corrupt PDF output

Table 4.13. Values of the source, data, and length parameters of PDF_open_image( )

source	*data*	*length*
fileref[1]	*string*[2] *with a platform-independent file name (see [1])*	*unused, should be 0*
url[1]	*string*[2] *with an image URL conforming to RFC 1738*[3]*. The URL will not be resolved by PDFlib, but by Acrobat when the PDF is opened (see Section 3.4.4, »Memory Images and External Image References«). This experimental feature is not recommended for production use.*	*unused, should be 0*
memory	*Binary bytes containing image data; the image data is compressed according to the type parameter. Exactly »length« bytes must be supplied.*	*length of (compressed) image data in bytes.*

[1] *Not supported in Acrobat 3 compatibility mode.*
[2] *data is not a string in Java and C++, which makes it a little bit clumsy to pass filenames or URLs.*
[3] *The URL must not contain any additional parameter, query string, access scheme, network login, or fragment identifier.*

may be generated, and Acrobat may respond with the *Image in Form, Type 3 font or pattern too big* error message.

If *type* is *raw*, length must be equal to *[width x components x bpc/ 8] x height* bytes, with the bracketed term adjusted upwards to the next integer, and this exact amount of data must be supplied. 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). 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.

If *type* is *ccitt,* CCITT-compressed image data is expected. In this case, *params* is examined. For CCITT images two parameters as described for *PDF_open_CCITT( )* can be supplied in the *params* string as follows:

```
/K -1 /BlackIs1 true
```

Supported values for */K* are -1, 0, or 1, the default value is 0. Supported values for */BlackIs1* are *true* and *false*; the default value is *false*. The default values will be used if an empty *params* string is supplied. BitReverse cannot be supplied in this string. Instead, a special notion is used: if *length* is negative, the image data will be reversed.

If *params* is not used, it must be empty. The client is responsible for the memory pointed to by the *data* argument. The memory may be freed by the client immediately after this call.

Don't use Photoshop-generated CMYK JPEG images with this function since these will appear in the PDF with inverted colors.

Scope *document, page;* must always be paired with a matching *PDF_close_image( )* call.

Params *imagewidth, imageheight, resx, resy, imagewarning*

void PDF_close_image(PDF *p, int image)

Close an image.

image A valid image handle retrieved with one of the *PDF_open_image*( )* functions.

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_open_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. This function must not be called for image handles which have been opened with *PDF_open_pdi_page( )* (use *PDF_close_pdi_page( )* instead).

Scope *document, page;* must always be paired with a matching call to one of the *PDF_open_image_file( ), PDF_open_CCITT( ), PDF_open_image( )* functions.

void PDF_place_image(PDF *p, int image, float x, float y, float scale)

Place an image or template, with the lower left corner at *(x, y),* and scale it.

image A valid image handle retrieved with one of the *PDF_open_image*( ) or PDF_begin_template( )* functions.

x, y The coordinates in the user coordinate system where the lower left corner of the placed image will be located.

scale The scaling factor which will be applied to the image in *x* and *y* direction.

Details See Section 3.4.2, »Code Fragments for Common Image Tasks« for more information on scaling and dpi calculations, including non-uniform scaling (different scaling factors in *x* and *y* dimensions).

Scope *page, pattern, template;* 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( )*.

See also *PDF_initgraphics( )* may be useful for initializing the graphics state before placing 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 An image handle which can be used in subsequent image-related calls. 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.

Params *imagewidth, imageheight*

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.

4.7 PDF Import Functions

Note All functions described in this chapter require the additional PDF import library (PDI) which is not part of the general PDFlib distribution. Please visit our Web site for more information on obtaining PDI.

4.7.1 Document and Page

int PDF_open_pdi(PDF *p, const char *filename , const char *stringparam, int intparam)

Open an existing PDF document and prepare it for later use.

filename The name of the PDF file.

stringparam, intparam Reserved for later use; must currently be NULL (or empty) and 0, respectively.

Returns A document descriptor 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-related problem (wrong PDF file name, unsupported format, bad PDF data, etc.), set the *pdiwarning* parameter to *true*.

Scope *document, page*

Params See Table 4.15 and Table 4.14.

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.

Scope *document, page*

Params See Table 4.14 and Table 4.15.

int PDF_open_pdi_page(PDF *p, int doc, int pagenumber, const char* pagelabel)

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.

pagelabel Reserved; must currently be NULL or empty.

Returns A page descriptor which can be used for placing pages with *PDF_place_image( ).* 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 4.14 and Table 4.15.

See also *PDF_place_pdi_page( )*

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 in the PDF document identified by *doc*, and releases all related resources. *page* must not be used after this call.

Scope *document, page*

Params See Table 4.14 and Table 4.15.

void PDF_place_pdi_page(PDF *p, int page, float x, float y, float sx, float sy)

Place a PDF page with the lower left corner at (x, y), and scale it.

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 in the user coordinate system where the lower left corner of the placed page will be located.

sx, sy The horizontal and vertical scaling factors which will be applied to the page.

Details This function is similar to *PDF_place_image( ),* but operates on imported PDF pages instead. Another difference is that *PDF_place_image( )* provides only a single scaling factor, while this function provides separate scaling factors in *x* and *y* direction.

Scope *page, pattern, template*

Params See Table 4.14 and Table 4.15.

4.7.2 Parameter Handling

float PDF_get_pdi_value(PDF *p, const char *key, int doc, int page, int index)

Get some PDI document parameter with numerical type.

key Specifies the name of the parameter to be retrieved, see Table 4.15 and Table 4.14.

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.

index Currently unused, must be 0.

Scope *any*

const char * PDF_get_pdi_parameter(
PDF *p, const char *key, int doc, int page, int index, int *len)

Get some PDI document parameter with string type.

key Specifies the name of the parameter to be retrieved, see Table 4.14 and Table 4.15.

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.

index Currently unused, must be 0.

len A C-style pointer to an integer which will receive the length of the returned string. This parameter is only required for C and C++.

Details This function gets some string parameter related to an imported PDF documented, in some cases further specified by *page* and *index*. Table 4.16 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 4.14. Page-related parameters and 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)*

Table 4.15. Document-related parameters and values for PDF import (the page parameter must be -1)

function	***key***	***explanation***
get_parameter	*pdi*	*Returns the string true, if the PDI library is attached (and not restricted to demo mode), and false otherwise.*
get_pdi_value	*/Root/Pages/Count*	*total number of pages in the imported document*
get_pdi_parameter	*filename*	*name of the PDF file*
get_pdi_value	*version*	*PDF version number multiplied by 10, e.g. 13 for PDF 1.3*
set_parameter	*pdiwarning*	*This parameter can be used to obtain more detailed information about why a PDF or page couldn't be opened:* *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. This is the default.*
set_parameter	*pdistrict*	*This parameter can be used to control PDI's behavior with respect to damaged PDF files:* *true* *Raise a nonfatal exception for non-conforming PDFs unless the warning parameter is set to false.* *false* *Accept certain kinds of damaged PDFs. This is the default.*

4.8 Hypertext Functions

4.8.1 Document Open Action and Open Mode

Table 4.16 lists relevant parameters and values for this section. These parameters can be set at an arbitrary time before calling *PDF_close( )*.

Table 4.16. Parameters for document open action and open mode (see Section 4.2.3, »Parameter Handling«)

function	***key***	***explanation***
set_parameter	*openaction*	*Set the open action, i.e., the zoom factor for the first page of the document. Possible values are retain, fitpage, fitwidth, fitheight, fitbbox (see Table 4.23). The default is retain. This parameter can be set once at an arbitrary time before PDF_close( ).*
set_parameter	*openmode*	*Set the appearance when the document is opened. The default value is 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 the document in fullscreen mode.*

4.8.2 Bookmarks

Table 4.17 lists relevant parameters for this section.

Table 4.17. Parameters for bookmarks (see Section 4.2.3, »Parameter Handling«)

function	*key*	*explanation*
set_parameter	*bookmarkdest*	*Set the target zoom for subsequently generated bookmark. Possible values are retain, fitpage, fitwidth, fitheight, fitbbox (see Table 4.23). This parameter can be changed an arbitrary number of times. The default is retain.*

Note *Adding bookmarks sets the open mode (see Section 4.8.1, »Document Open Action and Open Mode«) to bookmarks unless another mode has explicitly been set.*

int PDF_add_bookmark(PDF *p, const char *text, 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 be encoded with PDFDocEncoding or Unicode. The maximum length of *text* is 255 characters *(PDFDocEncoding),* or 126 Unicode characters. However, a practical limit of 32 characters for *text* is advised.

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. The bookmark target will be viewed at the current bookmark zoom factor which can be set via the *bookmarkdest* parameter (see Table 4.17).

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* that points to the current page. The zoom factor can be controlled with the *bookmarkdest* parameter.

Scope *page*

Params *openmode, bookmarkdest*

4.8.3 Document Information Fields

void PDF_set_info(PDF *p, const char *key, const char *value)

Fill document information field *key* with *value*.

key The *key* parameter must be encoded with PDFDocEncoding. *key* may be any of the five standard information field names, or an arbitrarily named custom field (see Table 4.18). 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]

[1] See http://purl.org/DC

Table 4.18. Values for the document information field key

key	*explanation*
Subject	*Subject of the document*
Title	*Title of the document*
Creator	*Creator of the document*
Author	*Author of the document*
Keywords	*Keywords describing the contents of the document*
any name other than CreationDate and Producer	*User-defined field. PDFlib supports an arbitrary number of fields. Names consist of printable characters except the following: blank ’ ’, %, (,), <, >, [,], {, }, /, and #.*

value The string to which the *key* parameter will be set. It can be encoded with PDF-DocEncoding or Unicode. Acrobat imposes a maximum length of *value* of 255 bytes.

Scope *document, page*

4.8.4 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 4.19 lists relevant parameters and values for this section.

Table 4.19. Parameters and values for page transitions (see Section 4.2.3, »Parameter Handling«)

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: replace.* *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: one second.*

4.8.5 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)

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 user space coordinates.

filename The name of the file which will be attached to the PDF document.

description A string with some explanation of the attachment. It may be encoded in PDFDocEncoding or Unicode.

author A string with the author's name or function. It may be encoded in PDFDocEncoding or Unicode.

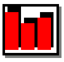
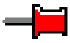
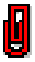
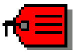
mimetype The MIME type of the file. It will be used by Acrobat for launching the appropriate program when the file attachment annotation is activated.

icon Controls the display of the unopened file attachment in Acrobat (see Table 4.20).

Details This function adds a file attachment annotation at the specified rectangle. PDF file attachments are only supported in Acrobat 4, and are therefore not supported in PDFlib's Acrobat 3 compatibility mode. Moreover, 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.

Scope *page*

Table 4.20. Icon names for file attachments

icon name	*icon appearance*	*icon name*	*icon appearance*
graph		*pushpin*	
paperclip		*tag*	

4.8.6 Note Annotations

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)

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 user space coordinates.

contents Text content of the note. It may be encoded with PDFDocEncoding or Unicode. The maximum length of *contents* is 65535 bytes.

title Heading text of the note. It may be encoded with PDFDocEncoding or Unicode. The maximum length of *title* is 255 characters *(PDFDocEncoding),* or 126 Unicode characters. However, a practical limit of 32 characters for *title* is advised.

icon Controls the display of the unopened note attachment in Acrobat (see Table 4.21).

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. Different note icons are only available in Acrobat 4, and are not supported in Acrobat 3 compatibility mode (the icon parameter must be empty in this case). Acrobat 3 viewers (and apparently Unix versions of Acrobat 4) will display the »note« type icon regardless of the supplied icon parameter.

Scope *page*

Table 4.21. Icon names for note annotations

icon name	*icon appearance*	*icon name*	*icon appearance*
comment	💬	*newparagraph*	▲ NP
insert	▲	*key*	🔑
note	📝	*help*	?
paragraph	¶		

4.8.7 Links

Table 4.22 lists relevant parameters for this section.

Note *PDF doesn't support links with shapes other than rectangles.*

Table 4.22. Parameters for links (see Section 4.2.3, »Parameter Handling«)

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.*

void PDF_add_pdflink(PDF *p, float llx, float lly, float urx, float ury,
const char *filename, int page, const char *dest)

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 user space coordinates.

filename The name of the target PDF file.

page The physical page number of the target page.

dest The destination zoom. It can attain one of the values specified in Table 4.23.

Scope *page*

void PDF_add_locallink(PDF *p, float llx, float lly, float urx, float ury,
int page, const char *dest)

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 user space coordinates.

page The physical page number of the target page. 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.

dest Specifies the destination zoom, which is one of the values specified in Table 4.23.

Scope *page*

Table 4.23. Values for the dest parameter of PDF_add_pdflink( ) and PDF_add_locallink( ). The same values are also used for the openaction (see Section 4.8.1, »Document Open Action and Open Mode«) and bookmarkdest parameters (see Section 4.8.2, »Bookmarks«).

dest	*explanation*
retain	*Retain the zoom factor which was in effect when the link was activated.*
fitpage	*Fit the complete page to the window.*
fitwidth	*Fit the page width to the window.*
fitheight	*Fit the page height to the window.*
fitbbox	*Fit the page's bounding box (the smallest rectangle enclosing all objects) to the window.*

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 user space coordinates.

filename The name of the file which will be launched upon clicking the link.

Scope *page*

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 user space coordinates.

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*

void PDF_set_border_style(PDF *p, const char *style, float width)

Set the border style for all kinds of annotations.

style Specifies the annotation border style, and must be one of *solid* or *dashed*.

width Specifies the annotation border width in points. If *width = 0* the annotation borders will be invisible.

Details The settings made by this function are used for all annotations until a new style is set. At the beginning of a document the annotation 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 all kinds of annotations.

red, green, blue The RGB color values for annotation borders.

Details 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 annotations.

b, w Specify the border dash style (see *PDF_setdash( )*).

Details At the beginning of a document the annotation 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*

4.8.8 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 one of the *PDF_open_image*( )* functions, but not a handle to a PDF page or template.

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 3.4.1, »Supported Image File Formats«).

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 (including the free Reader) generates thumbnails on the fly, 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 must have thumbnails attached to them.

Params *openmode*

4.9 Page Size Formats

For the convenience of PDFlib users, Table 4.24 lists common standard page sizes[1].

Table 4.24. Common standard page size dimensions in points

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

<format>_width, <format>_height, *where format is one of*
a0, a1, a2, a3, a4, a5, a6, b5, letter, legal, ledger, p11x17;

These macro definitions provide page width and height values for the most common page formats which may be used in calls to *PDF_begin_page( )*.

Bindings C and C++: macro definitions for these formats are available in *pdflib.h*

Other bindings may use the values provided in Table 4.24.

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

5 The PDFlib License

PDFlib is available under two separate licensing terms which are substantially different, and meet the needs of different developer groups. Please take the time to read the short summaries below in order to decide which one applies to your development.

5.1 The »Aladdin Free Public License«

This license applies to the main PDFlib package, but not to the ActiveX edition, any EBCDIC edition, and the PDI library (all of which are only available under the terms of the commercial PDFlib license). The complete text of the license agreement can be found in the file *aladdin-license.pdf*. In short and non-legal terms:

- you may develop free software with PDFlib, provided you make your source code available
- you may develop software for your own use with PDFlib as long as you don't sell it
- you may redistribute PDFlib non-commercially
- you may redistribute PDFlib on digital media for a fee if the complete contents of the media are freely redistributable.

Note that only the text in the file *aladdin-license.pdf* is considered to completely describe the licensing conditions. Project managers please note: using PDFlib in your commercial projects is not covered by the Aladdin license, and effectively means jeopardizing your project through unlicensed software.

5.2 The Commercial PDFlib License

A commercial PDFlib license is required for all uses of the software which are not explicitly covered by the Aladdin Free Public License, for example:

- applications which use the PDF import library (PDI)
- shipping a commercial product which contains PDFlib
- distributing (free or commercial) software based on PDFlib when the source code is not made available
- implementing commercial Web services with PDFlib

Different licensing options are available for PDFlib use on one or more servers, and for redistributing PDFlib with your own products. We also offer maintenance and support 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
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 (include your license number)
For other inquiries check our mailing list at http://www.egroups.com/group/pdflib.

6 References

[1] Adobe Systems Incorporated: PDF Reference, Second Edition: Version 1.3. Published by Addison-Wesley 2000, ISBN 0-201-61588-6; also available as PDF from
http://partners.adobe.com/asn/developer/technotes.html

[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/developer/technotes.html

[3] The following book by the principal author of PDFlib is available in English, German, and Japanese editions. It describes all aspects of integrating PDF in the Web:

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

German edition: Thomas Merz, Mit PDF ins Web.
Integration, Formulare, Sicherheit, E-Books.
Zweite Auflage. Mit CD-ROM. ISBN 3-935320-00-0, PDFlib Edition 2001
PDFlib GmbH, 80331 München, Tal 40, fax +49 • 89 • 29 16 46 86
http://www.pdflib.com, books@pdflib.com

Japanese edition: Tokyo Denki Daigaku 1999, ISBN 4-501-53020-0
http://plaza4.mbn.or.jp/~unit

A Shared Libraries and DLLs

Most PDFlib language bindings require the use of shared libraries, also known as shared objects or dynamic link libraries (DLLs). For your convenience we collected some general information about shared libraries in this appendix.

Shared Libraries on Unix Systems

The many faces of shared libraries. Most problems with shared libraries are related to the variety of methods invented by Unix vendors for implementing shared library support. In order to facilitate the use of shared libraries PDFlib leverages GNU libtool[1]. This is a collection of macros and shell scripts for building and using shared libraries on Unix systems. While libtool support is integrated in the PDFlib configuration, it is suggested to take a look at libtool and the corresponding documentation.

Building shared libraries. Although we do not cover all details of shared libraries here, the hints given below may be helpful for PDFlib users.

- On many systems a compiler flag (Linux: *-fPIC*) must be used for modules which are intended to be linked into a shared library (position-independent code, or PIC).
- Most systems require a special linker flag for shared libraries (Linux: *-shared*).
- The naming conventions for shared libraries vary (Linux and many others: *.so,* HP-UX: *.sl,* MacOS X: *.dylib*)
- Some systems require a version number to be included in the shared library file name, others at least tolerate it. Still others refuse to load libraries with version numbers in their names. The version number is often appended to the file name suffix (Linux: *lib<name>.so.5*, BSDI: *lib<name>.so5*).

The PDFlib *configure* script and GNU libtool take care of all these issues by constructing suitable Makefiles. In case of problems try to locate as much information as possible regarding the above issues, and compare with the generated Makefiles.

PDFlib's *configure* mechanism will build static versions of the library on systems where a shared library cannot be built. This implies that only the C and C++ language bindings will be available.

Using shared libraries. Once you managed to correctly build your shared library, you are not yet done – you must make sure that the run-time linker (which loads and runs your program) can access the library:

- In order to actually find shared libraries, a variety of mechanisms is deployed. The most common is an environment variable (Linux, Solaris, and many others: *LD_LIBRARY_PATH*, HP-UX: *SHLIB_PATH*, AIX: *LIBPATH,* MacOS X: *DYLD_LIBRARY_PATH)*. It contains a colon-separated list of directories which are searched for shared libraries. Failing that, a cache file (see below) is consulted, and then some default directories (Linux: */usr/lib* and */lib)*. Setting an environment variable doesn't require *root* privilege, and can be useful for testing. Library paths can also be hard-coded in the executable file using a special linker option (Solaris: *-R)*.
- In order to prepare the cache consulted by the run-time linker, a special program (Linux: *ldconfig*) must be invoked. This program scans all relevant locations for

[1] *See http://www.gnu.org/software/libtool/libtool.html*

shared libraries and sets up a cache with the known libraries (Linux: */etc/ld.so.cache)*. Usually this program is invoked at boot time, and requires *root* privilege. This technique is useful for permanently installing a shared library on a system.

The PDFlib *configure* script and GNU libtool emit some instructions explaining the required steps for using a shared library after the build process is completed. You may recognize some of the above information in these instructions.

In order to find out the shared libraries required by a program a special utility (Linux: *ldd)* can be invoked. It informs about the libraries which are required for running a given program, and tries to locate these on the system. This is convenient for the analysis of shared library related problems.

If you find yourself fiddling with shared library related problems because you cannot install the libraries due to a lack of administrator privileges, take a look at the *.libs* subdirectory and the library wrapper scripts created by libtool, as well as the test and install targets in the generated makefiles.

Library versioning scheme used by libtool. If the operating system supports a versioning scheme for shared libraries libtool will use it, and create versioned libraries. Library version numbers are different from software version numbers – don't expect PDFlib's major and minor version numbers to show up in library file names! Library versions rather identify the binary programming interface exposed by the library. A table with the PDFlib version numbers and the corresponding interface (libtool) numbers can be found in the distribution.

Windows DLLs

DLLs (Dynamic Link Libraries) on Windows generally don't pose any problems, the major exception being the cluttering of the Windows directory with all kinds of DLLs installed by every vendor and his dog. The PDFlib ActiveX component tries to avoid this issue by installing the required DLL into a private directory and custom registry entries. If you want to move the PDFlib DLL around your system, it may be useful to know the order in which Windows searches for DLLs:

- The current directory (this may actually be difficult to determine, e.g. if you are using a script interpreter).
- Windows 95/98: the Windows system directory
- Windows NT/2000: the 32-bit Windows system directory (*system32*)
- Windows NT/2000: the 16-bit Windows system directory (*system*)
- The Windows directory
- The directories listed in the *PATH* environment variable

Shared Libraries on the Macintosh

Shared libraries on the MacOS are fully supported on PowerPC machines via the Code Fragment Manager (*CFM*). A file type of *shlb* is generally used for shared libraries. The system looks for shared libraries in the following locations:

- The application folder
- The *Extensions* folder in the active system folder

MacOS X supports Unix-style shared libraries as detailed above.

B PDFlib Quick Reference

General Functions

Function prototype	*page*
void PDF_boot(void)	*78*
void PDF_shutdown(void)	*78*
int PDF_get_majorversion(void)	*79*
int PDF_get_minorversion(void)	*79*
*PDF *PDF_new(void)*	*79*
*PDF *PDF_new2(void (*errorhandler)(PDF *p, int type, 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)*	*79*
*void PDF_delete(PDF *p)*	*80*
*void *PDF_get_opaque(PDF *p)*	*80*
*int PDF_open_file(PDF *p, const char *filename)*	*81*
*int PDF_open_fp(PDF *p, FILE *fp)*	*81*
*void PDF_open_mem(PDF *p, size_t (*writeproc)(PDF *p, void *data, size_t size))*	*82*
*const char * PDF_get_buffer(PDF *p, long *size)*	*82*
*void PDF_close(PDF *p)*	*83*
*void PDF_begin_page(PDF *p, float width, float height)*	*83*
*void PDF_end_page(PDF *p)*	*83*
*float PDF_get_value(PDF *p, const char *key, float modifier)*	*84*
*void PDF_set_value(PDF *p, const char *key, float value)*	*84*
*const char * PDF_get_parameter(PDF *p, const char *key, float modifier)*	*84*
*void PDF_set_parameter(PDF *p, const char *key, const char *value)*	*84*

Text Functions

Function prototype	*page*
*int PDF_findfont(PDF *p, const char *fontname, const char *encoding, int embed)*	*85*
*void PDF_setfont(PDF *p, int font, float fontsize)*	*86*
*void PDF_show(PDF *p, const char *text)*	*86*
*void PDF_show2(PDF *p, const char *text, int len)*	*86*
*void PDF_show_xy(PDF *p, const char *text, float x, float y)*	*88*
*void PDF_show_xy2(PDF *p, const char *text, int len, float x, float y)*	*88*
*void PDF_continue_text(PDF *p, const char *text)*	*88*
*void PDF_continue_text2(PDF *p, const char *text, int len)*	*88*
*int PDF_show_boxed(PDF *p, const char *text, float x, float y, float width, float height, const char *mode, const char *feature)*	*89*
*float PDF_stringwidth(PDF *p, const char *text, int font, float size)*	*90*
*float PDF_stringwidth2(PDF *p, const char *text, int len, int font, float size)*	*90*
*void PDF_set_text_pos(PDF *p, float x, float y)*	*90*

Graphics Functions

Function prototype	***page***
*void PDF_setdash(PDF *p, float b, float w)*	*91*
*void PDF_setpolydash(PDF *p, float *darray, int length)*	*91*
*void PDF_setflat(PDF *p, float flatness)*	*91*
*void PDF_setlinejoin(PDF *p, int linejoin)*	*92*
*void PDF_setlinecap(PDF *p, int linecap)*	*92*
*void PDF_setmiterlimit(PDF *p, float miter)*	*93*
*void PDF_setlinewidth(PDF *p, float width)*	*93*
*void PDF_save(PDF *p)*	*94*
*void PDF_restore(PDF *p)*	*94*
*void PDF_translate(PDF *p, float tx, float ty)*	*94*
*void PDF_scale(PDF *p, float sx, float sy)*	*94*
*void PDF_rotate(PDF *p, float phi)*	*95*
*void PDF_skew(PDF *p, float alpha, float beta)*	*95*
*void PDF_concat(PDF *p, float a, float b, float c, float d, float e, float f)*	*95*
*void PDF_moveto(PDF *p, float x, float y)*	*96*
*void PDF_lineto(PDF *p, float x, float y)*	*96*
*void PDF_curveto(PDF *p, float x1, float y1, float x2, float y2, float x3, float y3)*	*97*
*void PDF_circle(PDF *p, float x, float y, float r)*	*97*
*void PDF_arc(PDF *p, float x, float y, float r, float alpha, float beta)*	*97*
*void PDF_arcn(PDF *p, float x, float y, float r, float alpha, float beta)*	*98*
*void PDF_rect(PDF *p, float x, float y, float width, float height)*	*98*
*void PDF_closepath(PDF *p)*	*98*
*void PDF_stroke(PDF *p)*	*99*
*void PDF_closepath_stroke(PDF *p)*	*99*
*void PDF_fill(PDF *p)*	*99*
*void PDF_fill_stroke(PDF *p)*	*99*
*void PDF_closepath_fill_stroke(PDF *p)*	*99*
*void PDF_clip(PDF *p)*	*100*
*void PDF_endpath(PDF *p)*	*100*

Color Functions

Function prototype	***page***
*void PDF_setcolor(PDF *p, const char *type, const char *colorspace, float c1, float c2, float c3, float c4)*	*100*
*int PDF_makespotcolor(PDF *p, const char *spotname, int len)*	*101*
*int PDF_begin_pattern(PDF *p, float width, float height, float xstep, float ystep, int painttype)*	*101*
*void PDF_end_pattern(PDF *p)*	*102*

Image Functions

Function prototype	***page***
*int PDF_open_image_file(PDF *p, const char *type, const char *filename, const char *stringparam, int intparam)*	*103*
*int PDF_open_CCITT(PDF *p, const char *filename, int width, int height, int BitReverse, int K, int BlackIs1)*	*103*
*int PDF_open_image(PDF *p, const char *type, const char *source, const char *data, long length, int width, int height, int components, int bpc, const char *params)*	*104*
*void PDF_close_image(PDF *p, int image)*	*105*
*void PDF_place_image(PDF *p, int image, float x, float y, float scale)*	*106*
*int PDF_begin_template(PDF *p, float width, float height)*	*106*
*void PDF_end_template(PDF *p)*	*106*

PDF Import (PDI) Functions

Function prototype	***page***
*int PDF_open_pdi(PDF *p, const char *filename , const char *stringparam, int intparam)*	*107*
*void PDF_close_pdi(PDF *p, int doc)*	*107*
*int PDF_open_pdi_page(PDF *p, int doc, int pagenumber, const char* pagelabel)*	*107*
*void PDF_close_pdi_page(PDF *p, int page)*	*108*
*void PDF_place_pdi_page(PDF *p, int page, float x, float y, float sx, float sy)*	*108*
*float PDF_get_pdi_value(PDF *p, const char *key, int doc, int page, int index)*	*109*
*const char * PDF_get_pdi_parameter(PDF *p, const char *key, int doc, int page, int index, int *len)*	*109*

Hypertext Functions

Function prototype	***page***
*int PDF_add_bookmark(PDF *p, const char *text, int parent, int open)*	*111*
*void PDF_set_info(PDF *p, const char *key, const char *value)*	*111*
*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)*	*112*
*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)*	*113*
*void PDF_add_pdflink(PDF *p, float llx, float lly, float urx, float ury, const char *filename, int page, const char *dest)*	*114*
*void PDF_add_locallink(PDF *p, float llx, float lly, float urx, float ury, int page, const char *dest)*	*114*
*void PDF_add_launchlink(PDF *p, float llx, float lly, float urx, float ury, const char *filename)*	*115*
*void PDF_add_weblink(PDF *p, float llx, float lly, float urx, float ury, const char *url)*	*115*
*void PDF_set_border_style(PDF *p, const char *style, float width)*	*115*
*void PDF_set_border_color(PDF *p, float red, float green, float blue)*	*116*
*void PDF_set_border_dash(PDF *p, float b, float w)*	*116*
*void PDF_add_thumbnail(PDF *p, int image)*	*116*

Parameters and Values

category	*function*	*keys*
setup	*set_parameter*	*prefix, resourcefile, compatibility, warning, flush*
	set_value	*compress, floatdigits*
	get_value	*major, minor, revision*
	get_parameter	*version*
document	*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*
font	*set_parameter*	*FontAFM, FontPFM, FontOutline, Encoding, fontwarning*
text	*set_value*	*leading, textrise, horizscaling, textrendering, charspacing, wordspacing*
	get_value	*leading, textrise, horizscaling, textrendering, charspacing, wordspacing, textx, texty, font, fontsize, capheight, ascender, descender*
	set_parameter	*underline, overline, strikeout, nativeunicode*
	get_parameter	*underline, overline, strikeout, fontname, fontencoding*
graphics	*set_parameter*	*fillrule*
	get_value	*currentx, currenty*
image	*get_value*	*imagewidth, imageheight, resx, resy*
	set_parameter	*imagewarning*
PDI	*get_parameter*	*pdi*
	set_parameter	*pdiwarning, pdistrict*
	get_pdi_value	*/Root/Pages/Count, /Rotate, version, width, height*
	get_pdi_ parameter	*filename*
hypertext	*set_parameter*	*openaction, openmode, bookmarkdest, transition, base*
	set_value	*duration*

C Revision History

Version information on PDFlib can be found in the source distribution.

Revision history of this manual

Date	*Changes*
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 ActiveX 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