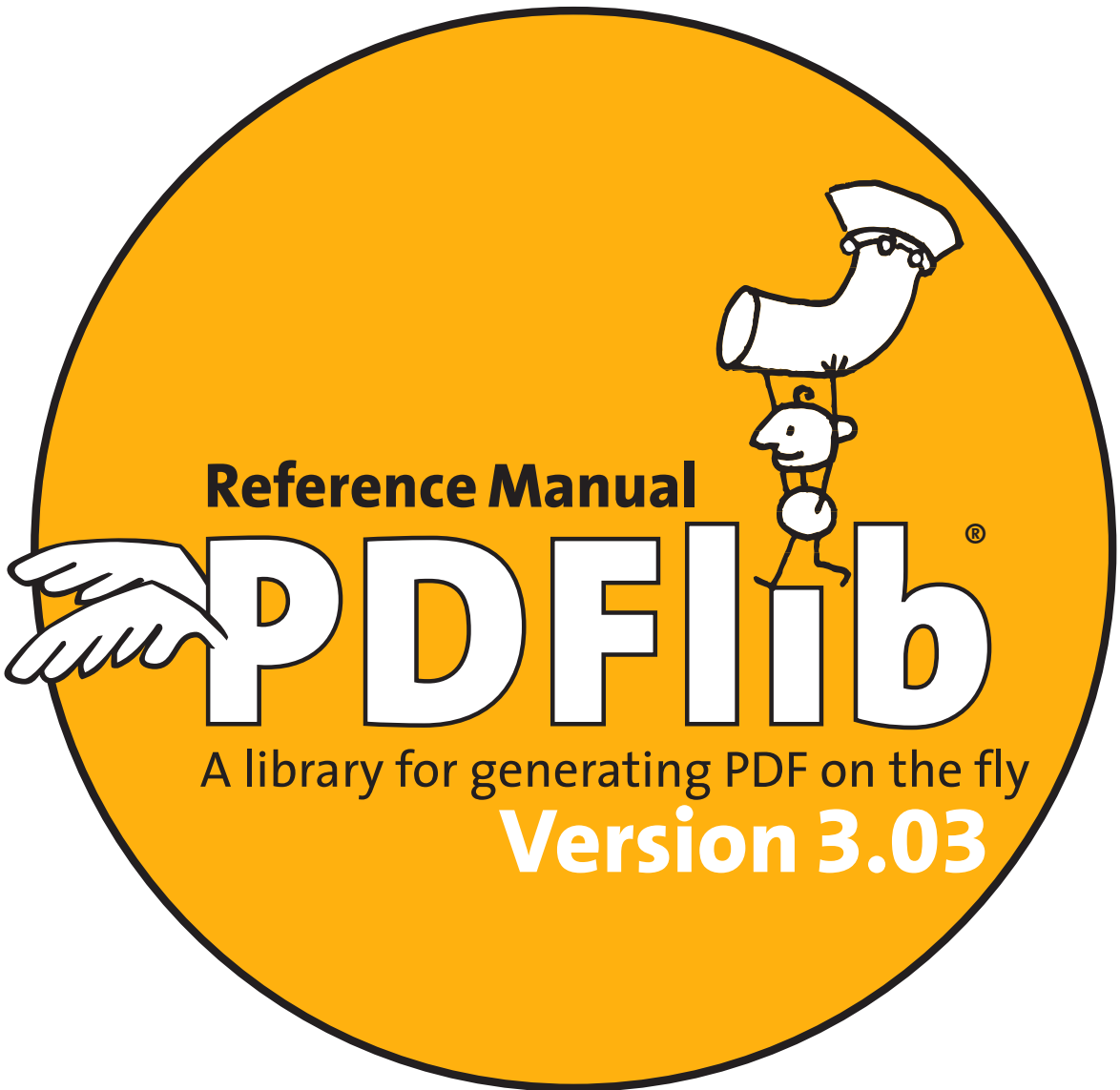
PDFlib GmbH München, Germany



www.pdflib.com

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

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

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

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

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

Revision history of this manual (version information on PDFlib can be found in the source distribution)

Date	***Changes***
December 22, 2000	► *ColdFusion documentation and additions for PDFlib 3.03; separate ActiveX edition of the manual*
August 08, 2000	► *Delphi documentation and minor additions for PDFlib 3.02*
July 01, 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. 01, 1999	► *Minor changes for PDFlib 1.0 (not publicly released)*
Aug. 10, 1998	► *Extensions for PDFlib 0.7 (only for a single customer)*
July 08, 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*

Contents

1 Introduction *7*

1.1 PDFlib Programming *7*

1.2 PDFlib Features *9*

1.3 PDFlib Output and Compatibility *10*

2 PDFlib Language Bindings *12*

2.1 Overview of the PDFlib Language Bindings *12*

2.1.1 What's all the Fuss about Language Bindings? ***12***
2.1.2 Availability and Platforms ***12***
2.1.3 The »Hello world« Example ***14***
2.1.4 Error Handling ***14***
2.1.5 Version Control ***14***
2.1.6 Unicode Support ***14***
2.1.7 Summary of the Language Bindings ***15***

2.2 ActiveX/COM Binding *15*

2.3 C Binding *15*

2.3.1 How does the C Binding work? ***15***
2.3.2 Availability and Special Considerations for C ***15***
2.3.3 The »Hello world« Example in C ***16***
2.3.4 Error Handling in C ***16***
2.3.5 Version Control in C ***18***
2.3.6 Unicode Support in C ***18***

2.4 C++ Binding *18*

2.4.1 How does the C++ Binding work? ***18***
2.4.2 Availability and Special Considerations for C++ ***18***
2.4.3 The »Hello world« Example in C++ ***19***
2.4.4 Error Handling in C++ ***19***
2.4.5 Version Control in C++ ***20***
2.4.6 Unicode Support in C++ ***20***

2.5 Java Binding *20*

2.5.1 How does the Java Binding work? ***20***
2.5.2 Installing the PDFlib Java Edition ***21***
2.5.3 The »Hello world« Example in Java ***22***
2.5.4 Error Handling in Java ***23***
2.5.5 Version Control in Java ***23***
2.5.6 Unicode Support in Java ***24***

2.6 Perl Binding *24*

2.6.1 How does the Perl Binding work? ***24***
2.6.2 Installing the PDFlib Perl Edition ***24***
2.6.3 The »Hello world« Example in Perl ***25***
2.6.4 Error Handling in Perl ***26***

2.6.5 Version Control in Perl *26*
2.6.6 Unicode Support in Perl *26*

2.7 Python Binding *26*
2.7.1 How does the Python Binding work? *26*
2.7.2 Installing the PDFlib Python Edition *26*
2.7.3 The »Hello world« Example in Python *27*
2.7.4 Error Handling in Python *27*
2.7.5 Version Control in Python *28*
2.7.6 Unicode Support in Python *28*

2.8 Tcl Binding *28*
2.8.1 How does the Tcl Binding work? *28*
2.8.2 Installing the PDFlib Tcl Edition *28*
2.8.3 The »Hello world« Example in Tcl *29*
2.8.4 Error Handling in Tcl *30*
2.8.5 Version Control in Tcl *30*
2.8.6 Unicode Support in Tcl *30*

3 PDFlib Programming Concepts *31*

3.1 General Programming Issues *31*
3.1.1 PDFlib Program Structure *31*
3.1.2 Memory Management *31*
3.1.3 Generating PDF Documents directly in Memory *32*
3.1.4 Error Handling *33*

3.2 Page Descriptions *35*
3.2.1 Coordinate Systems *35*
3.2.2 Paths and Color *37*
3.2.3 Ordering constraints *37*

3.3 Text Handling *38*
3.3.1 The PDF Core Fonts *38*
3.3.2 8-Bit Encodings built into PDFlib *38*
3.3.3 Custom Encoding Files for 8-Bit Encodings *40*
3.3.4 Hypertext Encoding *42*
3.3.5 PostScript Fonts *43*
3.3.6 Resource Configuration and the UPR Resource File *45*
3.3.7 CID Font Support for Japanese, Chinese, and Korean Text *47*
3.3.8 Unicode Support *51*
3.3.9 Text Metrics, Text Variations, and Text Box Formatting *55*

3.4 Image Handling *58*
3.4.1 Supported Image File Formats *58*
3.4.2 Code Fragments for Common Image Tasks *60*
3.4.3 Re-using Image Data *62*
3.4.4 Memory Images and External Image References *62*
3.4.5 Image Masks and Transparency *62*
3.4.6 Multi-Page Image Files *64*

4 PDFlib API Reference *66*

4.1 Data Types and Naming Conventions *66*

4.2 General Functions *67*
- 4.2.1 Setup *67*
- 4.2.2 Document and Page *69*
- 4.2.3 Parameter Handling *71*

4.3 Text Functions *71*
- 4.3.1 Font Handling *71*
- 4.3.2 Text Output *73*

4.4 Graphics Functions *76*
- 4.4.1 General Graphics State *76*
- 4.4.2 Special Graphics State *77*
- 4.4.3 Path Segments *79*
- 4.4.4 Path Painting and Clipping *80*

4.5 Color Functions *81*

4.6 Image Functions *82*

4.7 Hypertext Functions *85*
- 4.7.1 Document Open Action and Open Mode *85*
- 4.7.2 Bookmarks *85*
- 4.7.3 Document Information Fields *86*
- 4.7.4 Page Transitions *86*
- 4.7.5 File Attachments *87*
- 4.7.6 Note Annotations *87*
- 4.7.7 Links *88*

4.8 Page Size Formats *89*

5 The PDFlib License *91*

5.1 The »Aladdin Free Public License« *91*

5.2 The Commercial PDFlib License *91*

6 References *92*

A Shared Libraries and DLLs *93*

B Summary of PDFlib Functions *96*

Index *99*

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, MacOS, and EBCDIC-based systems such as IBM AS/400 and S/390. Although PDFlib itself is written in the C language, its functions can be accessed from several other languages and programming environments which are called language bindings. The

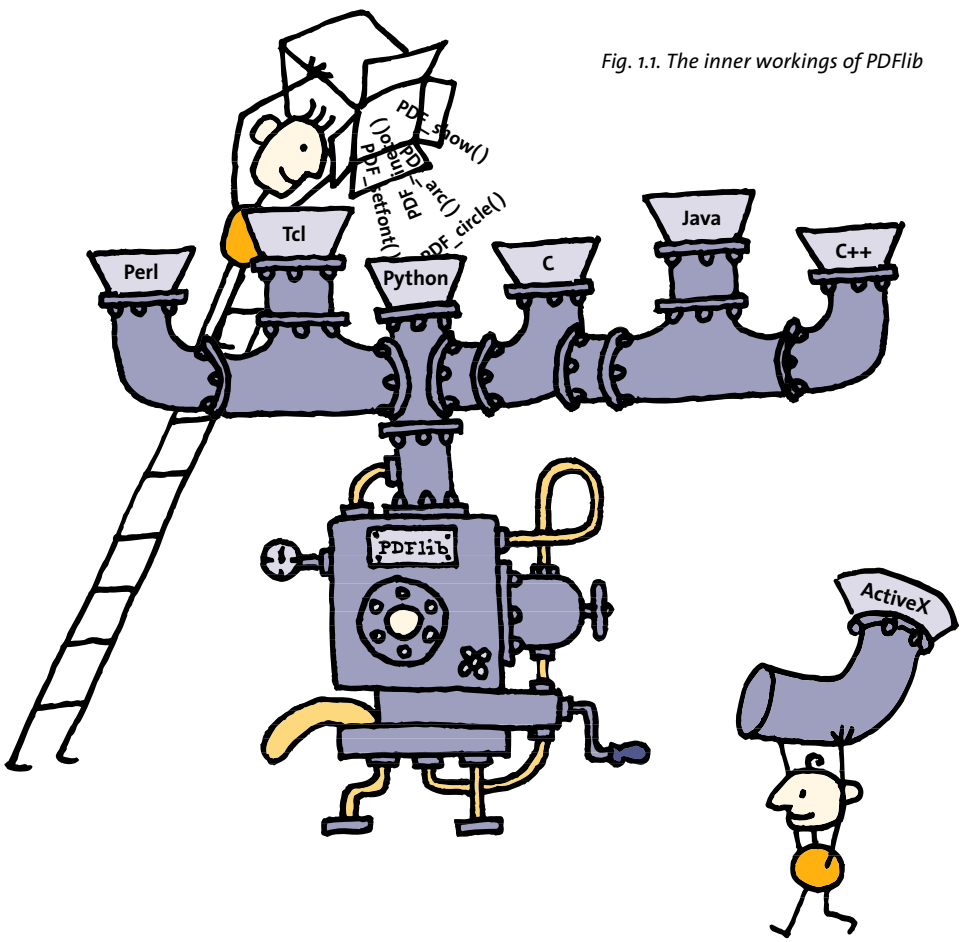


Fig. 1.1. The inner workings of PDFlib

PDFlib 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, Windows Script Host, Active Server Pages with VBScript or JScript, Allaire ColdFusion, Borland Delphi, and many other environments
- ANSI C
- ANSI C++
- Java, including servlets
- Python
- Perl
- 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 500+ 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 on specific platforms. 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/PDF features or internals. 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 Documents	▸ 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 ▸ strict Acrobat 3 / PDF 1.2 mode optionally available
Vector graphics	▸ common vector graphics primitives: lines, curves, arcs, rectangles, etc. ▸ vector paths for stroking, filling, and clipping ▸ RGB color for stroking and filling objects
Fonts	▸ text output in different fonts ▸ text column formatting ▸ underlined, overlined, and strikeout text ▸ built-in font metrics for PDF's 14 base fonts ▸ PostScript Type 1 (PFB and PFA file formats) font support with or without font embedding ▸ support for AFM and PFM font metrics files ▸ library clients can retrieve character metrics for exact formatting ▸ flexible font and metrics file configuration
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, e.g., ISO 8859-2
Unicode	▸ Unicode support for hypertext features: bookmarks, contents and title of text annotations, document information fields, attachment description, and author name ▸ Unicode encoding for Japanese, Chinese, and Korean text
Images	▸ embed images in GIF (non-interlaced), PNG, TIFF, JPEG, or CCITT file formats ▸ 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, 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 ▸ supports both ASCII- and EBCDIC-based platforms

Table 1.2 lists PDF features which are currently not implemented in PDFlib.

Table 1.2. Features which are currently not implemented in PDFlib

topic	remarks
dealing with existing PDFs	PDFlib generates new PDF documents, but doesn't integrate or manipulate existing PDF content. Embedding existing PDF contents will be supported when PDFlib 4.0 is available in Spring 2001 (check our Web site).
encryption	Encryption requires all page contents to be cryptographically processed.
thumbnails	Thumbnails require a rasterizer for the page contents.
linearization	Linearization (Web optimization) requires a complex rewrite of the PDF file.
EPS embedding	Embedding EPS graphics requires a PostScript interpreter which would slow down operation. Instead, we wil support PDF embedding in the future.
font subsetting	Font subsetting requires extended font processing.
TrueType fonts	PDFlib currently doesn't support TrueType fonts. The TrueType font format will be supported when PDFlib 4.0 is available in Spring 2001 (check our Web site).

1.3 PDFlib Output and Compatibility

PDFlib output. PDFlib generates binary PDF output (although most items can also be generated in ASCII mode for debugging purposes). If the Zlib compression library is available (this is the case for all binary PDFlib distributions), the PDF output will be compressed with the Flate (also known as ZIP) compression algorithm. The 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 1.3 lists those features. More details can be found at the respective function descriptions.

Table 1.3. 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 ▸ CID fonts for Chinese, Japanese, and Korean require Acrobat 3J or Acrobat 4
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 (Reader and Exchange) 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 dual-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.

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

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 and others)	Windows	MacOS	S/390, AS/400 (zSeries and iSeries)
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	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, and 5.7	ActivePerl 5.005 and 5.6	MacPerl 5.2.0r4, based on Perl 5.004	–
Python	Python 1.5.2 and 2.0	Python 1.5.2 and 2.0	Python 1.5.2	–
Tcl	Tcl 8.3.1 and 8.4a2	Tcl 8.3.1 and 8.4a2	Tcl 8.3.0	–

Using PDFlib on MacOS. As shown in Table 2.1, all relevant language bindings are supported on MacOS. The following MacOS-specific differences in PDFlib 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 texts.
- 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.

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 zSeries and iSeries, previously known as IBM S/390 and AS/400. 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
- document information (if not Unicode, see function descriptions)
- string parameters to PDFlib functions
- input and output file names
- environment variables (if supported by the 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 automatic conversion must be avoided):

- PDF output files

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

The following restrictions restrictions must be observed on EBCDIC-based platforms:

- Not all PDFlib language bindings are EBCDIC-safe (see Table 2.2 for details). Depending on demand other bindings may be adapted to EBCDIC platforms in the future.
- Due to restrictions in PDF, text box formatting *(PDF_show_boxed( ))* is not supported for EBCDIC encoding.

2.1.3 The »Hello world« Example

Being a well-known classic, the »Hello, world!« example will be used for the first PDFlib program. 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.

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 and minor version number at the interface level. These may be used by the client in order to decide whether the given PDFlib implementation can be accepted, 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 the Language Bindings

For easy reference, Table 2.2 summarizes important features of the PDFlib language bindings. More details can be found in the respective section 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	–	–
Python	Python exceptions	–	manually	–	–
Tcl	Tcl exceptions	yes (Tcl 8.2 or 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

```
/* hello.c
 *
 * PDFlib client: hello 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" */

    exit(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 error 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«. Macro definitions for the error types can be found in *pdflib.h*. These are constructed by prefixing the error name with *PDF_* (e.g., *PDF_MemoryError)*. 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.

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( )* 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 the *setjmp( )/longjmp( )* facility.

The following code may be used as a starting point for developing a custom error handler:

```
void custom_errorhandler(PDF *p, int type, const char* shortmsg)
{
    char msg[256];

    sprintf(msg, "Application error: %s\n", shortmsg);

    (void) fprintf(stderr, msg);        /* Issue a warning message in all cases */

    switch (type) {
        case PDF_NonfatalError:
            return;

        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:

            /*  if p == NULL the exception was thrown by the initial malloc()
             *  for the PDF object, and we must not call PDF_delete().
             */
            if (p != NULL)                  /* first allocation? */
                PDF_delete(p);              /* clean up PDFlib */

            exit(99);                       /* brutal way of saying good-bye */
    }
}
```

Obviously, the appropriate action when an error happens is completely application specific. The above sample doesn't even attempt to handle the error, but simply exits. A custom error handler can be installed in PDFlib by using *PDF_new2( )*.

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.

2.4 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.

In most environments there are inherent issues related to C++ deployment with shared libraries which adversely affect portability. For this reason it is suggested to statically bind *pdflib.cpp* to your application, and use the generic PDFlib C library as a shared library (if shared libraries are to be used at all).

2.4.3 The »Hello world« Example in C++

```
// hello.cpp
//
// PDFlib client: hello 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 not be a class method

(unless it is a static class method) since it will be called indirectly through a function pointer without any class association.

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 simple 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 (rather private) data type name *PDF_c* although the PDFlib API reference calls for the *PDF* data type.

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«).

2.5 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)[2]. 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 simply 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 loading the shared library.

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 (at least on the server side), remember that PDFlib runs on many more platforms than the Java VM!

1. See http://java.sun.com
2. See http://java.sun.com/products/jdk/1.2/docs/guide/jni/index.html

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)[1], 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, the auxiliary libraries (Unix only), 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.

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

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

Unix. On Unix systems the library name supplied in the PDFlib Java class file will be decorated according to the system's naming conventions for the names of shared libraries (usually by prepending *lib* and appending *.so*). The library 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. On Windows systems the file name of the PDFlib wrapper is *pdf_java.dll*. This DLL must be placed in the Windows system directory, or a directory which is listed in the PATH environment variable.

Macintosh. On the Mac the library name supplied in the PDFlib Java class file is used without any change (i.e. *pdf_java*). The shared library is 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). Note that the above naming not only relates to the file name, but also to the fragment name of the library which must be correctly set by the linker. Also, you may want to increase the amount of memory allocated to JBindery, especially if you want to process images with PDFlib.

1. See http://devworld.apple.com/java

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

```
/* hello.java
 *
 * PDFlib client: hello 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. 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.

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*

2.5.5 Version Control in Java

Version control for the Java binding is done transparently when loading the shared library. The wrapper code for loading the PDFlib shared library relies on the major and minor version numbers (which are queried from the PDFlib library). An exact match of both minor and major version number is required. If a PDFlib library with a non-matching version number is found, the wrapper code will raise a *java.lang.InternalError* exception.

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 is loaded at runtime by the Perl interpreter, with some help from the package file. The shared library module is referred to from the Perl script 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 have access to the PDFlib Perl wrapper, the auxiliary libraries (Unix only), 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* search path using a command similar to the following:

```
use lib qw(/home/tm/pdflib-3.03/bind/perl/.libs);
use pdflib_pl 3.03;
```

before the *use pdflib_pl* line. However, Perl will only use the path given in this command for the PDFlib wrapper library (e.g., *pdflib_pl.so.o)*, but not any required auxiliary libraries (these are only required on Unix). These must either be made available via operating system specific methods (see Appendix A, »Shared Libraries and DLLs«), or built into the PDFlib wrapper library.

Unix. On Unix systems both *pdflib_pl.so* and *pdflib_pl.pm* will be found if placed in the current directory, or the directory printed by the following Perl command:

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

Perl will also search the subdirectory *auto/pdflib_pl*. 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
```

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

Windows. On the Windows platform 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 found if placed 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 exchanged between ActivePerl 5.6 and older versions. The PDFlib binary distribution contains DLLs for both.*

Macintosh. PDFlib supports the Macintosh port of Perl known as MacPerl[2]. Both the shared library *pdflib_pl* and *pdflib_pl.pm* will be found if placed in the current folder, or in one of the following folders:

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

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). You may want to increase the memory allocated to the Perl interpreter, especially if you want to process images with PDFlib.

2.6.3 The »Hello world« Example in Perl

```
#!/usr/bin/perl
# hello.pl
#
# PDFlib client: hello example in Perl
#

use pdflib_pl 3.03;

$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!");
```

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

```
PDF_continue_text($p, "(says Perl)");
PDF_end_page($p);
PDF_close($p);

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 3.03;
```

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 Python Binding

2.7.1 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.

2.7.2 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. On Unix systems the PDFlib shared library for Python *pdflib_py.so* and the auxiliary libraries will be searched in the directories listed in the PYTHONPATH environment variable.

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

Windows. On Windows systems the PDFlib shared library *pdflib_py.dll* will be searched in the directories listed in the PYTHONPATH environment variable.

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

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

2.7.3 The »Hello world« Example in Python

```
#!/usr/bin/python
# hello.py
#
# PDFlib client: hello 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"
    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);
```

2.7.4 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
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

2.7.5 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.

2.7.6 Unicode Support in Python

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

2.8 Tcl Binding

2.8.1 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.

In addition, the PDFlib Tcl binding leverages the idea of extension packages introduced in Tcl 7.5. All PDFlib functions are packed into a single Tcl extension package. The shared library module is referred to from the Tcl script via a *package* statement.

2.8.2 Installing the PDFlib Tcl Edition

The Tcl extension mechanism works by loading shared libraries at runtime. For extending the Tcl interpreter with PDFlib, Tcl 8.0 or higher is required (because of its support for binary strings). Unicode support requires Tcl 8.2 or higher. The PDFlib wrapper code for Tcl may also be compiled for older versions of Tcl (down to 8.0). The supplied binaries, however, require Tcl 8.2 or higher.

For the PDFlib binding to work, the Tcl shell must have access to the PDFlib Tcl wrapper (the supplied test programs use *auto_path* to make the library available from the current directory; this facilitates testing) and the package index file *pkgIndex.tcl*:

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

Unix. On Unix systems the library name *pdflib_tcl.so* supplied in the *pkgIndex.tcl* file, and the auxiliary libraries 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. Unfortunately, Tcl doesn't itself produce a platform-specific decoration of the library name. If you compile the Tcl binding from source code, you must change the library name *pdflib_tcl.so* supplied in the *pkgIndex.tcl* file to the appropriate name *pdflib_tcl.dll* (this has already been done in the binary distribution). A library by this name will be searched in the Tcl shell's directory, the current directory, the *Windows* and *Windows\system32* directories, and the directories listed in the PATH environment variable. The files *pkgIndex.tcl* and *pdflib_tcl.dll* will be searched for in the directories

```
C:\Program Files\Tcl 8.3\lib\tcl8.3\
C:\Program Files\Tcl 8.3\lib\tcl8.3\pdflib
```

Mac. On the Mac the library *pdflib_tcl.so* and *pkgIndex.tcl* will be searched in the Tcl shell's folder, and in the folders

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

2.8.3 The »Hello world« Example in Tcl

```
#!/bin/sh
#
# hello.tcl
#
# PDFlib client: hello example in Tcl
#

# Hide the exec to TCL but not to the shell by appending a backslash\
exec tclsh "$0" ${1+"$@"}

# The lappend line is unnecessary if PDFlib has been installed
# in the Tcl package directory
lappend auto_path .

package require pdflib 3.03

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
```

2.8.4 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
}
```

2.8.5 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 3.03
```

in order to make sure they will get the required library version (or a newer one, which is ok for PDFlib).

2.8.6 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 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 checks for several conditions in the ordering of API calls, but doesn't attempt to trap all kinds of illegal function call combinations. In the development phase it will be helpful to take a look at all warning messages generated by PDFlib, since these usually point to problems in the client's ordering of function calls. PDFlib will throw an exception if bad parameters are supplied by a library client.

3.1.2 Memory Management

Note This section applies to C and C++ PDFlib clients only. All other language bindings leverage the internal memory management of the language environment.

PDFlib dynamically allocates and frees lots of small and large memory chunks. The general strategy is to strictly separate PDFlib memory from client memory. In order to achieve this, data supplied by the client to PDFlib functions is copied into PDFlib memory space if the data is still needed after the call is finished. Consequently, PDFlib is responsible for freeing such memory when the data is no longer needed.

In order to allow for maximum flexibility, PDFlib's internal memory management routines (which are based on standard C *malloc/free*) may be replaced by external procedures provided by the client. These procedures will be called for all PDFlib-internal memory allocation or deallocation.

It is not reasonable to provide custom memory management routines from the scripting language bindings (since freeing the programmer from memory management chores is a major advantage of scripting languages). For this reason, custom memory management routines are only available for the C and C++ programmer. For all other language bindings memory management is handled in the respective wrapper code.

Memory Management in C. Memory management routines can be installed with a call to *PDF_new2()*, and will be used in lieu of PDFlib's internal memory management 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.

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 alloca-

tion 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.

The signatures of the memory management routines can be found in Section 4.2, »General Functions«.

Memory Management in C++. 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.

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.

Note User-supplied memory management routines are used (besides PDFlib) in the Zlib compression library, but not in the TIFF and PNG libraries.

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 buf...
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 neither touch nor 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. However, 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

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

- running out of virtual memory
- not adhering to programming restrictions (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. The implementation of raising exceptions is obviously language-specific, and part of the wrapper code.

For C and C++ clients which chose to 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. More information on implementing a custom error handler (for C and C++) and catching exceptions (for other language bindings) can be found in Chapter 2.

Runtime errors in PDFlib applications 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 as arguments, 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	*decimal value*	*explanation*
MemoryError	*1*	*not enough memory*
IOError	*2*	*input/output error, e.g. disk full*
RuntimeError	*3*	*wrong order of PDFlib function calls*
IndexError	*4*	*array index error*
TypeError	*5*	*argument type error*
DivisionByZero	*6*	*division by zero*
OverflowError	*7*	*arithmetic overflow*
SyntaxError	*8*	*syntactical error*
ValueError	*9*	*a value supplied as argument to PDFlib is invalid*
SystemError	*10*	*PDFlib internal error, or configuration problem (e.g., version mismatch)*
NonfatalError	*11*	*A non-fatal problem was detected. Non-fatal errors (warnings) can be suppressed using PDF_set_parameter( ).*
UnknownError	*12*	*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* 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.

The freely available Acrobat plugins *callas pdfMeasure*[1] or *Enfocus Measure*[2] may help in determining object sizes and distances in PDF files.

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.

1. See http://www.callas.de/download.htm
2. See http://www.enfocus.com/plugins.htm

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);
```

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 compatibility mode (the default) and the Acrobat 4 limits are exceeded PDFlib will only issue a non-fatal warning message. Common standard page size dimensions can be found in Table 4.21..

Table 3.3. Minimum and maximum page sizes supported by Acrobat 3 and 4

PDF viewer	*minimum page size*	*maximum page size*
Acrobat 3	*1" = 72 pt = 2.54 cm*	*45" = 3240 pt = 114.3 cm*
Acrobat 4	*1/24" = 3 pt = 0.106 cm*	*200" = 14400 pt = 508 cm*

3.2.2 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. Paths may be stroked or filled, or used for 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. Clipping reduces the imageable area by replacing the current clipping area (which is the page size by default) with the intersection of the current clipping area and the path.

It's important to understand that 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 and 1
- RGB triples, i.e., three values between 0 and 1 specifying the percentage of red, green, and blue.

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

3.2.3 Ordering constraints

Ordering constraints for path functions. For the sake of efficiency, PDF page descriptions must obey certain restrictions related to the ordering of path description, building, and using the path. In particular, none of the following functions must be used between the beginning of a path (i.e., one of the functions listed in Section 4.4.3, »Path Segments«) and its natural demise (i.e., one of the functions listed in Section 4.4.4, »Path Painting and Clipping«):

- all functions listed in Section 4.4.1, »General Graphics State« (e.g., changing line width or linecap)
- all functions listed in Section 4.4.2, »Special Graphics State«
- all functions listed in Section 4.5, »Color Functions« (e.g., changing the fill or stroke color)

These rules may easily be summarized as »don't change the appearance within a path description«.

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 3.4 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 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 3.4. 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
```

while Latin text fonts will usually have the entry

```
EncodingScheme AdobeStandardEncoding
```

Fonts with the Adobe StandardEncoding can be reencoded to *winansi*, *macroman*, or *ebcdic* encodings, while fonts with *FontSpecific* encoding can't, and must use *builtin* encoding instead. PDFlib issues a warning message when an attempt is made to reencode symbol fonts.

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 labeled with Adobe StandardEncoding.*

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 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 code pages different from the one supported internally in PDFlib. The following steps must be followed 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«).
- 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) encoding:

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

More formally, the contents of an encoding 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 a PostScript character name, followed by whitespace and a decimal character code in the decimal range 0–255.
- Character codes which are not mentioned in the encoding file are assumed to be undefined.

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 (see Section 3.3.6, »Resource Configuration and the UPR Resource File«). Sample encoding files are supplied with PDFlib.

Distributed encoding files. The PDFlib distribution contains several encoding files 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«).

Table 3.5. Additional external character encodings distributed with PDFlib

encoding	*description*
iso8859-2	*Latin-2: this character set 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.*
iso8859-9	*Latin-5 (yes, that's 5, not 9!): this character set 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.*
iso8859-15	*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.*

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[1] 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).[2]

For the convenience of PDFlib users, the PostScript program *print_glyphs.ps* in the distribution fileset can be used to find the names of all characters contained in a PostScript font. In order to use it, enter the name of the font at the end of the PostScript file and send it (along with the font) to a PostScript Level 2 or 3 printer, or view it with a Level-2-compatible PostScript viewer such as Ghostscript[3]. 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

1. See http://www.fontlab.com
2. 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).
3. See http://www.cs.wisc.edu/~ghost

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.

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«.

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 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 3.6 lists different situations with respect to font usage, each of which poses different requirements on the necessary font and metrics files.

Table 3.6. 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 fonts without embedding	*yes*	*no*
Non-core fonts with embedding	*yes*	*yes*
Additional font/encoding combinations for which the metrics have been compiled into PDFlib (see below)	*no*	*yes, if font embedding is requested*
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.

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.

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.

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

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:

- 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 in-core 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.

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.

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 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. 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.[1] There is an Adobe-supplied utility called *makepsres* floating around the Internet which can be used to automatically generate UPR files from PostScript font outline and metrics files.

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 directory 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 followed by the directory prefix. Note that the initial slash character is required on all platforms, and is not part of the directory 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 ab-

1. For those interested, 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

solute 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 currently supported in PDFlib are listed in Table 3.7. Other resource categories may be present in the UPR file for compatibility with Display PostScript installations, but they will silently be ignored.

Table 3.7. 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 font outline file in PFA or PFB format*
Encoding	*Text file containing an 8-bit encoding table*

Redundant resource entries should be avoided. For example, do not include multiple entries for a certain font's metrics data.

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
```

```
.
Encoding
iso8859-2=iso8859-2.enc
.
```

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 (which may be set using *PDF_set_parameter( ))* is examined and used as a resource file name, if set.
- 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 4 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 4 product CD-ROM.

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.

- If you use Acrobat Reader 4, install one of the Asian Font Packs which are available on the Acrobat 4 product CD-ROM, or on the Web.[1]

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 3.8. 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 are supported. Tables with all supported characters are available from Adobe[2]; CMap descriptions can be found in Table 3.9.

Table 3.8. Acrobat's standard fonts for Japanese, Chinese, and Korean text

locale	*available standard fonts in Acrobat*	*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*

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 3.8). The *HeiseiKakuGo* sample in Table 3.8 has been generated with the following code:

```
font = PDF_findfont(p, "HeiseiKakuGo-W5", "Ext-RKSJ-H", 0);
PDF_setfont(p, font, 24);
```

1. See http://www.adobe.com/prodindex/acrobat/cjkfontpack.html
2. 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. Predefined CMaps for Japanese, Chinese, and Korean text (from the PDF Reference [1])

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

```
PDF_set_text_pos(p, x, y);
PDF_show(p, "\x93\xFA\x96\x7B\x8C\xEA");
```

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 3.8). 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 appropriate 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 fully supports the Unicode standard. 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.[1] 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«)

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

- Unicode support for the actual page descriptions is only available for CJK fonts.

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

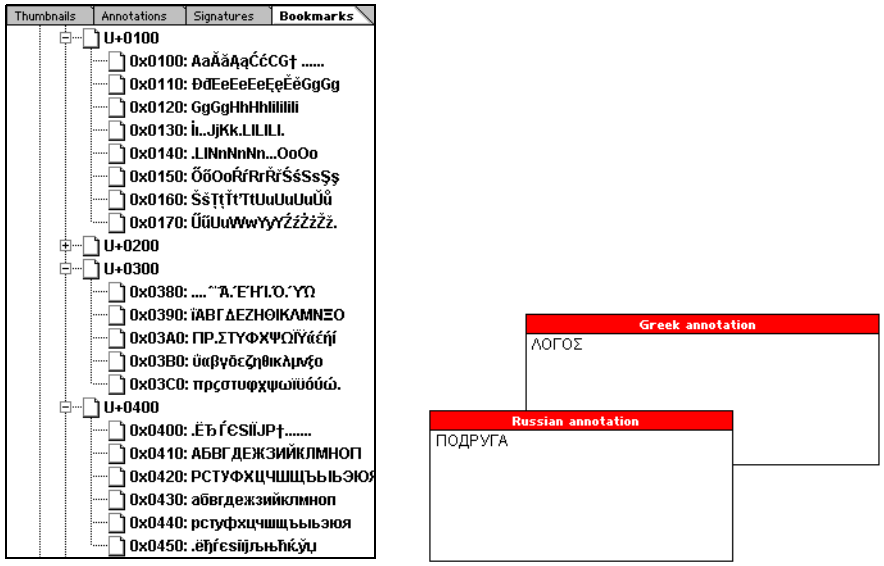


Fig. 3.2. Unicode bookmarks (left) and Unicode text annotations (right)

- 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 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 functions. 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.

Unicode encoding for page descriptions. 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 3.9). 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.

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 3.8).

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");
```

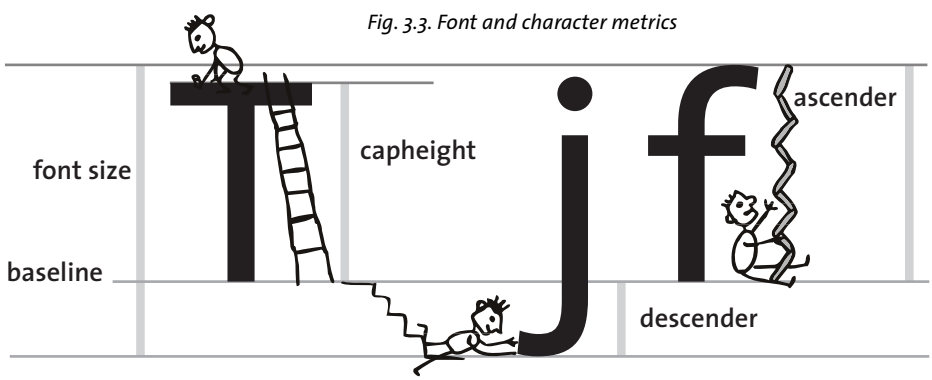
This will pass the intended character code 0x96 to PDFlib, which will correctly interpret it according to the chosen encoding vector (although the Java compiler will be fooled into believing it deals with the *Unicode* character 0x0096, which doesn't actually exist).

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.

Fig. 3.3. Font and character metrics



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
```

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

Table 3.10. 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 box formatting. While PDFlib offers the *PDF_stringwidth( )* function for performing text width calculations, many clients need easy access to text box formatting and justifying, e.g. to fit a certain amount of text into a given column. Although PDFlib offers such features, you shouldn't think of PDFlib as a full-featured text and graphics layout engine. The *PDF_show_boxed( )* function is an easy-to-use method for text box formatting with a number of formatting options. Text may be laid out in a rectangular box either left-aligned, right-aligned, centered, or fully justified. The first line of text starts at a baseline with a vertical position which equals the top edge of the supplied box minus the leading. The bottom edge of the box serves as the last baseline used. For this reason, descenders of the last text line may appear outside the specified box (see Figure 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 *(oxoA)* in the supplied text are recognized, and force a new paragraph. CR/NL combinations are treated like a single newline character. Other formatting characters (especially tab characters) are not supported.

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

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

font = PDF_findfont(p, "Helvetica", "host", 0);
PDF_setfont(p, font, fontsize);

x = 50;
y = 650;
w = 357;
```

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.

```
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* 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. If the freely available PNG reference library (which in turn requires the Zlib compression library) is installed, PDFlib supports all flavors of PNG images (»Portable Network Graphics«).[1] PNG images are currently not handled in pass-through mode. If PNG images contain 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 PDF, and therefore PDFlib.

Note All binary distributions of PDFlib support the PNG image file format.

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 available JPEG images.
- Progressive JPEG compression which is supported by Acrobat 4/PDF 1.3. If run in Acrobat 3 compatibility mode, however, PDFlib will refuse to import progressive JPEGs.

PDFlib applies a workaround which is necessary for correctly processing 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.

1. See http://www.w3.org/Graphics/PNG and http://www.libpng.org/pub/png for more information on the PNG image file format and the libpng library

- 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 in Acrobat's print dialog.

TIFF images. Sam Leffler's TIFFlib[1] can be plugged into PDFlib in order to support many TIFF compression and encoding flavors. In most cases PDFlib will handle TIFF images in pass-through mode. PDFlib supports the following flavors of TIFF images:

- compression schemes: uncompressed, CCITT (group 3, group 4, and RLE), LZW, and PackBits (Macintosh RunLength encoding) 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-in-TIFF) and certain combinations of features (e.g., LZW compression and alpha channel) 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«. Since the problem does not exist in Ghostscript and Acrobat displays the image just fine, we consider this a bug in Acrobat. You may be able to work around it by choosing a different TIFF compression scheme.

Note All binary distributions of PDFlib support the TIFF image file format.

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 descrip-

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

tor. 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) {
        scale_x = ((float) 72.0) / dpi_x;
        scale_y = ((float) 72.0) / dpi_y;
} else if (dpi_x < 0 && dpi_y < 0) {
        scale_x = (float) 1.0;
        scale_y = dpi_y / dpi_x;
} else {
        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 like */
```

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 (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 – 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:

  ```
  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.
- Ghostscript 6.0 does not support masked images in PDF.

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.

Note 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 as 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);
}
```

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 and Naming Conventions

PDFlib data types. The exact syntax to be used for a particular language binding may actually vary slightly from the C syntax shown here in the reference. This especially holds true for the PDF document parameter *(PDF *** in the API reference) which has to be supplied as the first argument to almost all PDFlib functions in the C binding, but not those bindings which hide the PDF document parameter in an object created by the language wrapper.

Table 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	*PDF document parameter required?*	*function names use 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*
Python	*yes*	*yes*	*string*	*string*
Tcl	*yes*	*yes*	*string*	*byte array[3]*

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.*
3. Byte arrays were introduced in Tcl 8.1.

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 into the following when the function is used from Java:

```
pdflib p;
...
p.open_file(String filename);
```

4.2 General Functions

4.2.1 Setup

Note *Users of the Java binding can ignore the functions in this section.*

Table 4.2 lists relevant parameters and values for this section.

Table 4.2. Parameters and values for the setup functions

function	*key*	*explanation*
set_parameter	*compatibility*	*Set PDFlib's compatibility mode to one of the strings »1.2« or »1.3« for Acrobat 3 or 4. 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. Note that strict Acrobat 3 compatibility mode is not required for generating Acrobat 3 compatible files, but only in 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 a discussion of flushing strategies. 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.*
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	*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. Default value is 6. This parameter does not affect precompressed image data which is handled in pass-through mode.* *0 no compression* *1 best speed* *9 best compression*

void PDF_boot(void)
void PDF_shutdown(void)
Boot and shut down PDFlib, respectively. Recommended for the C language binding, although currently not required. 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.

Note *Both functions are not available in the ActiveX, Java, Perl, and Tcl bindings because these supply their own versioning schemes.*

PDF *PDF_new(void)

Create a new PDF object, using PDFlib's internal default error handling and memory allocation routines. *PDF_new( )* 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 the library; only pointers to the PDF structure are used at the API level.

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.

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

PDF_new( ) must always be paired with a matching *PDF_delete( )* call.

Note *This function is not available in the C++ binding since it is hidden in the PDF constructor. In the ActiveX and Java bindings this function is automatically called by the wrapper code, and therefore also 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. Returns a pointer to the opaque PDF data type which is required as the *p* argument for all other functions. 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.

PDF_new2( ) must always be paired with a matching *PDF_delete( )* call.

Note *In the C++ binding this function is indirectly available via the PDF constructor. Not all function arguments must be given since default values of NULL are supplied. In all bindings other than C and C++ 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 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.

PDF_delete( ) must always be paired with a matching call to one of the *PDF_new( )* or *PDF_new2 ( ) functions.*

Note *In the C++ binding this function is indirectly available via the PDF destructor. In the ActiveX and Java bindings this function is automatically called by the wrapper code, and therefore not available.*

void *PDF_get_opaque(PDF *p)
Return 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.

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

4.2.2 Document and Page

Table 4.3 lists relevant parameters and values for this section.

Table 4.3. 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. The parameters must be given as strings.*

int PDF_open_file(PDF *p, const char *filename)
Open a new PDF file associated with *p*, 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. This function returns -1 on error, and 1 otherwise.

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).

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 error value.

PDF_open_file( ) must always be paired with a matching *PDF_close( )* call.

Note In the C++ binding 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. The function returns -1 on error, and 1 otherwise.

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.

PDF_open_fp( ) must always be paired with a matching *PDF_close( )* call.

Note This function is only available in the C and C++ bindings. In the C++ binding, it 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 document in memory, without writing to a disk file. The user-supplied *writeproc* callback function will be called by PDFlib in order to submit (portions of) the generated PDF data. 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).

PDF_open_mem( ) must always be paired with a matching *PDF_close( )* call.

Note This function is only available in the C and C++ bindings. In the C++ binding it is hidden in the overloaded open( ) call. Other language bindings can 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)
Fetch the full or partial buffer containing the generated PDF data. This function must only be called between page descriptions (i.e., after *PDF_end_page( )* and before *PDF_begin_page( ))*, or after *PDF_close( )* and before *PDF_delete( )* (the latter is not required by all language bindings). This function must only be called if an empty filename has been supplied to *PDF_open_file( )*. It returns a buffer full of binary PDF data for consumption by the client. The *size* parameter is only used for C and C++ clients, and points to a memory location where the length of the returned data in bytes will be stored. In all other language bindings an object of appropriate length will be returned, and the *size* parameter must be omitted.

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.

This function returns a language-specific data type for binary data according to Table 4.1.

void PDF_close(PDF *p)
Finish 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( )*.

PDF_close( ) 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)
Start a new page in the PDF file. 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 4.21 in Section 4.8, »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*.

PDF_begin_page( ) must always be paired with a matching *PDF_end_page( )* call.

void PDF_end_page(PDF *p)
Must be used to finish a page description. *PDF_end_page( )* 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 numerical value of some internal PDFlib parameter *key*, in some cases characterized by the *modifier*. For parameters where the description doesn't mention *modifier*, it is ignored and must be 0.

void PDF_set_value(PDF *p, const char *key, float value)
Set some numerical PDFlib parameter *key* to *value*.

const char * PDF_get_parameter(PDF *p, const char *key, float modifier)
Get the string value of some PDFlib parameter *key*, in some cases further characterized by *modifier*. For parameters where the description doesn't mention *modifier*, it is ignored and must be 0.

Note C and C++ clients must neither touch nor free the returned string.

void PDF_set_parameter(PDF *p, const char *key, const char *value)
Set the string value of some PDFlib parameter *key* to *value*.

4.3 Text Functions

4.3.1 Font Handling

Table 4.4 lists relevant parameters and values for this section.

Table 4.4. 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«)*
get_value	*font*	*Return the identifier of the current font which must have been set with PDF_setfont( ). Must only be called within a page description.*
get_parameter	*fontname*	*The name of the current font which must have been previously set with PDF_setfont( ). This function must only be called within a page description.*
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( ). This function must only be called within a page description. Note that the returned encoding name may not be literally identical to the encoding parameter supplied to PDF_findfont( ) because host encoding has been resolved, or PDFlib forced a different encoding because the requested encoding couldn't be used.*
get_value	*fontsize*	*Return the size of the current font which must have been previously set with PDF_setfont( ). This function must only be called within a page description.*

Table 4.4. Parameters and values for the font functions (see Section 4.2.3, »Parameter Handling«)

function	*key*	*explanation*
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 if the font/encoding combination cannot be loaded (instead of throwing an exception). Default value is true.*

int PDF_findfont(PDF *p, const char *fontname, const char *encoding, int embed)
Prepare the font *fontname* for later use with *PDF_setfont( )*. The metrics will be loaded from memory or from an external metrics file.

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 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 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*.

Note CID fonts are not supported in Acrobat 3 compatibility mode.

If the *embed* parameter has the value 0, only general font information is included in the PDF output. If *embed = 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.

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. *PDF_findfont( )* can safely be called outside of page descriptions.

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 if the requested font/encoding combination cannot be loaded, and does not throw an exception. However, exceptions will still be thrown for when bad parameters are passed.

Note 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.

void PDF_setfont(PDF *p, int font, float fontsize)
Set the current font in the given *fontsize*. The font descriptor must have been retrieved with *PDF_findfont( )*. This function must only be called within a page description. 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.

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 4.4 lists relevant parameters and values for this section.

Table 4.5. 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 3.10. 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«).

void PDF_show(PDF *p, const char *text)
Print *text* in the current font and font size at the current text position. 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. In the C and C++ bindings *text* must not contain null characters.

void PDF_show2(PDF *p, const char *text, int len)
Same as *PDF_show( ),* but with explicit string length in bytes for strings which may contain null characters. If *len* = *0* a null-terminated string is assumed as in *PDF_show( )*.

Note This function is only available for the C and C++ bindings, and is not required for the other language bindings.

void PDF_show_xy(PDF *p, const char *text, float x, float y)
Print *text* in the current font at position *(x, y)*. The font must have been set before. The current point is moved to the end of the printed text. In the C and C++ bindings *text* must not contain null characters.

void PDF_show_xy2(PDF *p, const char *text, int len, float x, float y)
Same as *PDF_show_xy( ),* but with explicit string length in bytes for strings which may contain null characters. If *len* = *0* a null-terminated string is assumed as in *PDF_show_xy( )*.

Note This function is only available for the C and C++ bindings, and is not required for the other language bindings.

void PDF_continue_text(PDF *p, const char *text)
Move to the next line and print *text*. The start of the next line is determined by the leading parameter and the most recent call to *PDF_show_xy( )* or *PDF_set_text_pos( )*. The current point is moved to the end of the printed text. In the C and C++ bindings *text* must not contain null characters. This function should not be used in vertical writing mode.

void PDF_continue_text2(PDF *p, const char *text, int len)
Same as *PDF_continue_text( ),* but with explicit string length in bytes for strings which may contain null characters. If *len* = *0* a null-terminated string is assumed as in *continue_text( )*.

Note This function is only available for the C and C++ bindings, and is not required for the other language bindings.

int PDF_show_boxed(PDF *p, const char *text, float x, float y, float width, float height, const char *mode, const char *feature)
Format the supplied *text* into a rectangular column. *mode* selects the horizontal alignment mode as discussed below.

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.

This function 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*, this function always returns 0 in this case.

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.

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.

This function cannot be used with CID fonts. It is safe to use *PDF_continue_text( )* after this function if *mode* = *left* or *justify*.

float PDF_stringwidth(PDF *p, const char *text, int font, float size)
Return 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. In the C and C++ bindings *text* must not contain null characters.

This function cannot be used with CID fonts. If the current font is a CID font, this function returns 0 regardless of the *text* and *size* arguments.

This function must only be called within a page description.

float PDF_stringwidth2(PDF *p, const char *text, int len, int font, float size)
Same as *PDF_stringwidth( ),* but with explicit string length in bytes for strings which may contain null characters. If *len* = *0* a null-terminated string is assumed as in *PDF_stringwidth( )*.

This function must only be called within a page description.

Note This function is only available for the C and C++ bindings, and is not required for the other language bindings.

void PDF_set_text_pos(PDF *p, float x, float y)
Set the current text position to *(x, y)*. The text position is set to the default value of *(0, 0)* at the beginning of each page.

Note The current point for graphics output and the current text position are maintained separately.

4.4 Graphics Functions

All functions in this section must only be called within page descriptions.

4.4.1 General Graphics State

Note *Don't use general graphics state functions within a path description (see Section 3.2, »Page Descriptions«), but use one of the fill, stroke, or clip functions before you call one of the general graphics state functions. If you don't obey this rule, Acrobat will issue a »drawing error occurred« error message.*

void PDF_setdash(PDF *p, float b, float w)
Set the current dash pattern to *b* black and *w* white units. *b* and *w* must be non-negative numbers. In order to produce a solid line, set *b* = *w* = 0. The dash parameter is set to solid at the beginning of each page.

void PDF_setpolydash(PDF *p, float *darray, int length)
Set a more complicated dash pattern. The array of the given *length* contains alternating values for black and white dash lengths. The array values must be non-negative, and not all zero. 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.

Note *The length parameter is only required for the C and C++ language bindings. Other language bindings simply supply the array as argument, and the language wrapper will automatically determine its length.*

void PDF_setflat(PDF *p, float flatness)
Set the flatness to a value between 0 and 100 inclusive. The *flatness* parameter describes the maximum distance (in device pixels) between the path and an approximation constructed from straight line segments. 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.

void PDF_setlinejoin(PDF *p, int linejoin)
Set the linejoin parameter to a value of 0, 1, or 2. The *linejoin* parameter specifies the shape at the corners of paths that are stroked, as shown in Table 4.6. The linejoin parameter is set to the default value of 0 at the beginning of each page.

void PDF_setlinecap(PDF *p, int linecap)
Set the linecap parameter to a value 0, 1, or 2. The linecap parameter controls the shape at the ends of open paths with respect to stroking, as shown in Table 4.7. The linecap parameter is set to the default value of 0 at the beginning of each page.

void PDF_setmiterlimit(PDF *p, float miter)
Set the miter limit to a value greater than or equal to 1. The *miterlimit* parameter is set to the default value of 10 at the beginning of each page.

Table 4.6. Values of the linejoin parameter

value	description (from the PDF specification)	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.	

Table 4.7. Values of the linecap parameter

value	description (from the PDF specification)	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_setlinewidth(PDF *p, float width)
Set the current line width to *width* units in the user coordinate system. The linewidth parameter is set to the default value of 1 at the beginning of each page.

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( ),* and *PDF_concat( ))* change the coordinate system used for drawing future objects. They do not affect existing objects on the page at all.

void PDF_save(PDF *p)
Save the current graphics state. The graphics state contains parameters that control all types of graphics objects. Saving the graphics state is not required by PDF; it is only necessary if the application wishes to return to some specific graphics state later (e.g., a custom coordinate system) without setting all relevant parameters explicitly again. The following items are subject to save/restore:

- graphics parameters: clipping path, coordinate system, current point, flatness, line cap style, dash pattern, line join style, line width, miter limit;
- color parameters: fill and stroke colors;
- text parameters: character spacing, word spacing, horizontal scaling, leading, font, font size, rendering mode, text rise;

The saved graphics state does not include the values of the *fillrule*, *underline*, *overline*, and *strikeout* parameters.

This function must not be called within a path construction sequence. *PDF_save( )* must always be paired with a matching *PDF_restore( )* call. *PDF_save( )* and *PDF_restore( )* calls must be balanced on each page.

void PDF_restore(PDF *p)
Restore the most recently saved graphics state. The corresponding graphics state must have been saved on the same page. Pairs of *PDF_save( )* and *PDF_restore( )* may be nested.

This function must not be called within a path construction sequence. *PDF_restore( )* must always be paired with a matching *PDF_save( )* call. *PDF_save( )* and *PDF_restore( )* calls must be balanced on each page.

Note 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.

void PDF_translate(PDF *p, float tx, float ty)
Translate the origin of the coordinate system to *(tx, ty)*. The new origin of the coordinate system is the point *(tx, ty)*, measured in the old coordinate system.

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

Note Due to limitations in the Acrobat viewers, PDFlib must output coordinates with absolute values above 32.767 as integers. This may affect output accuracy in rare cases (when very small scaling factors and very large coordinates are used).

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

void PDF_skew(PDF *p, float alpha, float beta)
Skew the coordinate system by the angles of *alpha* and *beta* degrees. Skewing (or shearing) distorts the coordinate system by the given angles in x and y direction. Angles are measured counterclockwise from the positive x axis of the current coordinate system. Both *alpha* and *beta* must be different from 90° and 270°.

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 (CTM) for text and graphics. This function 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. The six floating point

values make up the 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*.

4.4.3 Path Segments

Table 4.8 lists relevant parameters and values for this section.

Table 4.8. 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.

void PDF_moveto(PDF *p, float x, float y)
Set the current point to *(x, y)*. The current point is set to the default value of *undefined* at the beginning of each page.

Note The current point for graphics output and the current text position are maintained separately.

void PDF_lineto(PDF *p, float x, float y)
Add 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.

void PDF_curveto(PDF *p, float x1, float y1, float x2, float y2, float x3, float y3)
Add a Bézier curve from the current point to *(x3, y3)* to the current path, 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.

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

void PDF_arc(PDF *p, float x, float y, float r, float start, float end)
Add a counterclockwise circular arc segment with center *(x, y)* and nonnegative radius *r* to the current path, extending from *start* to *end* degrees. 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.

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

void PDF_closepath(PDF *p)
Close the current subpath, i.e., add a line from the current point to the starting point of the subpath. If the current point is not set this function will have no effect.

4.4.4 Path Painting and Clipping

Table 4.9 lists relevant parameters and values for this section.

Table 4.9. 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 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 (draw) the current path with the current line width and the current stroke color. This operation clears the path.

void PDF_closepath_stroke(PDF *p)
Close the current subpath (add a straight line segment from the current point to the starting point of the path), and stroke the complete current path with the current line width and the current stroke color. This operation clears the path.

void PDF_fill(PDF *p)
Fill the interior of the current path with the current fill color. The interior of the path is determined by one of two algorithms (see *PDF_setfillrule( )*). Open paths are implicitly closed before being filled. This operation clears the path.

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

void PDF_closepath_fill_stroke(PDF *p)
Close the current subpath (add a straight line segment from the current point to the starting point of the path), and fill and stroke the complete current path. This operation clears the path.

void PDF_clip(PDF *p)
Use the intersection of the current path and the current clipping path as the clipping path for future operations. The clipping path is set to the default value of the page size at the beginning of each page. This operation clears the path. The clipping path is subject to *PDF_save( )/PDF_restore( )*. The clipping path can only be enlarged by means of *PDF_save( )/PDF_restore( )*. This operation clears the path.

void PDF_endpath(PDF *p)
Terminate the current path without filling or stroking it. This function is deprecated, and its use is discouraged. Use one of the above stroke, fill, or clip functions instead.

4.5 Color Functions

All color functions expect values in the inclusive range 0–1. Grayscale and color values 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)* means black; a gray value of 1 and RGB values with *(r, g, b) = (1, 1, 1)* means white. Color values in the range 0–255 must be scaled to the range 0–1 by dividing by 255.

All functions in this section must only be called within page descriptions.

Note Don't use color functions within a path description (see Section 3.2, »Page Descriptions«).

void PDF_setgray_fill(PDF *p, float gray)
Set the current fill color to the *gray* value. The gray fill parameter is set to the default value of 0 at the beginning of each page.

void PDF_setgray_stroke(PDF *p, float gray)
Set the current stroke color to the *gray* value. The gray stroke parameter is set to the default value of 0 at the beginning of each page.

void PDF_setgray(PDF *p, float gray)
Set the current fill and stroke color to the *gray* value. The gray parameter is set to the default value of 0 at the beginning of each page.

void PDF_setrgbcolor_fill(PDF *p, float red, float green, float blue)
Set the current fill color to the supplied RGB values. The rgbcolor fill parameter is set to the default value of *(0, 0, 0)* at the beginning of each page.

void PDF_setrgbcolor_stroke(PDF *p, float red, float green, float blue)
Set the current stroke color to the supplied RGB values. The rgbcolor stroke parameter is set to the default value of *(0, 0, 0)* at the beginning of each page.

void PDF_setrgbcolor(PDF *p, float red, float green, float blue)
Set the current fill and stroke color to the supplied RGB values. The rgbcolor parameter is set to the default value of *(0, 0, 0)* at the beginning of each page.

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 4.10 lists relevant parameters and values for this section.

Table 4.10. 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 sucessfully with PDF_open_image_file( ) or PDF_open_CCITT( ):* *true* — *Raise a Nonfatal exception when the image function fails. The information string supplied with the exception may be useful in debugging image-related problems.* *false* — *Do not raise an exception when the image function fails. Instead, the function returns -1 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 and analyze a raster graphics file in one of the supported file formats as determined by the *type* parameter. The *type* parameter may attain the following values: *png*, *gif*, *jpeg*, *tiff* (case is significant for all parameters). The returned image handle, if not -1, may be used in subsequent image-related calls. In order to get more detailed information about the nature of an image-related problem (wrong image file name, bad image data, etc.), set the *imagewarning* parameter to *true* (see Table 4.10).
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 is not kept open after this call.

The *stringparam* and *intparam* parameters are used for additional image attributes according to Table 4.11. If *stringparam* is unused, it must be an empty string, and *intparam* must be 0.

PDF_open_image_file( ) must always be paired with a matching *PDF_close_image( )* call.

Note The returned image handle cannot be reused across multiple PDF documents.

Table 4.11. 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 supplied this image 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.*
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 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!). The returned image handle, if not -1, may be used in subsequent image-related calls. However, since PDFlib is unable to analyze CCITT images, all relevant parameters have to be passed to *PDF_open_CCITT( )* by the client. The parameters have the following meaning (apart from *filename*, *width*, and *height*, which are obvious):

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

K: CCITT compression parameter for encoding scheme selection. It has to be set as follows: -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*.

PDF_open_CCITT( ) must always be paired with a matching *PDF_close_image( )* call.

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)

This versatile interface can be used to work with image data in several formats and from several data sources. The returned image handle, if not -1, may be used in subsequent image-related calls.

The *type* parameter denotes the kind of image data or compression. It can attain the values *jpeg*, *ccitt*, or *raw* (see Section 3.4.1, »Supported Image File Formats«); 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 4.12. The *data* parameter has binary data type according to Table 4.1.

Note Images referenced via external files or URLs are not supported in Acrobat 3 compatibility mode.

The *width* and *height* parameters describe the dimensions of the image. The number of color *components* must be 1, 3, or 4 corresponding to grayscale, RGB, or CMYK image data.

Table 4.12. Values of the source, data, and length parameters of PDF_open_image( )

source	*data*	*length*
fileref	*string[1] with a platform-independent file name (see [1])*	*unused, should be 0*
url	*string[1] with an image URL conforming to RFC 1738[2]. 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. data is not a string in Java and C++, which makes it a little bit clumsy to pass filenames or URLs.
2. The URL must not contain any additional parameter, query string, access scheme, network login, or fragment identifier.

The number of bits per component *bpc* must be 1, 2, 4, or 8. *width*, *height*, *components*, and *bpc* must always be supplied.

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 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 *components = 1* and *bpc = 1*, *params* may be *mask* in order to use this image as an image mask.

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.

PDF_open_image( ) must always be paired with a matching *PDF_close_image( )* call.

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

void PDF_close_image(PDF *p, int image)

Close the image. This 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_file( )* call. An image handle cannot be used any more after having been closed with this function, since it cuts PDFlib's internal association with the image.

PDF_close_image( ) must always be paired with a matching call to one of the *PDF_open_image_file( )*, *PDF_open_CCITT( )*, or *PDF_open_image( )* functions.

void PDF_place_image(PDF *p, int image, float x, float y, float scale)
Place the supplied image (which must have been retrieved with one of the *PDF_open_*( )* functions) on the current page. The lower left corner of the image is placed at *(x, y)* on the page, and the image is scaled by the supplied scaling factor. 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).

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( )*. *PDF_place_image( )* must only be used within page descriptions, i.e., between *PDF_begin_page( )* and *PDF_end_page( )*.

4.7 Hypertext Functions

4.7.1 Document Open Action and Open Mode

Table 4.13 lists relevant parameters and values for this section. These parameters can be set at an arbitrary time before calling *PDF_close( )*.

Table 4.13. 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. The meaning of these values is explained in Table 4.20. 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.

4.7.2 Bookmarks

Table 4.14 lists relevant parameters for this section.

Table 4.14. Parameters for bookmarks (see Section 4.2.3, »Parameter Handling«)

function	key	explanation
set_parameter	bookmark-dest	Set the target zoom for bookmarks generated in the future. Possible values are retain, fitpage, fitwidth, fitheight, fitbbox. The meaning of these values is explained in Table 4.20. This parameter can be changed an arbitrary number of times. The default is retain.

Note Adding bookmarks sets the open mode (see Section 4.7.1, »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 PDF bookmark with the supplied *text* that points to the current page. The text may be encoded with PDFDocEncoding or Unicode. This function must not be called before starting the first page of the document with *PDF_begin_page( )*.

This function returns an identifier for the bookmark just generated. This identifier may be used as the *parent* parameter in subsequent calls. In this case, 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. If the *open* parameter has a value of 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 4.14).

The maximum length of *text* is 255 characters *(PDFDocEncoding),* or 126 Unicode characters. However, a practical limit of 32 characters for *text* is advised.

This function must only be called within page descriptions.

4.7.3 Document Information Fields

void PDF_set_info(PDF *p, const char *key, const char *value)
Fill document information field *key* with *value*. The value can be encoded with PDFDocEncoding or Unicode, while the *key* must be encoded with PDFDocEncoding. *key* may be any of the five standard information field names, or an arbitrarily named custom field (see Table 4.15). There is no limit for the number of custom fields. Acrobat imposes a maximum length of *value* of 255 bytes.

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]

Table 4.15. 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 custom field name other than CreationDate and Producer	*User-defined field name. PDFlib supports an arbitrary number of custom fields. Custom field names must consist of printable characters except the following: blank ' ', %, (,), <, >, [,], {, }, /, and #.*

4.7.4 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 4.16 lists relevant parameters and values for this section.

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

Table 4.16. 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 any subsequent pages until until the transition is changed again. The transition type strings givenbelow are supported. type may also be empty to reset the transition effect. The default transition is replace, i.e., no special effect. 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. The default duration is one second.

4.7.5 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 at the rectangle specified by its lower left and upper right corners in default user space coordinates. *description* and *author* may be encoded in PDFDocEncoding or Unicode. *mimetype* is the MIME type of the file and will be used by Acrobat for launching the appropriate program when the file attachment annotation is activated. The *icon* parameter controls the display of the unopened file attachment in Acrobat, as shown in Table 4.17.

Table 4.17. Icon names for file attachments

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

Note *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.*

4.7.6 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 at the rectangle specified by its lower left and upper right corners in default user space coordinates. *contents* and *title* may be encoded with PDFDocEncoding or Unicode. The *icon* parameter controls the display of the unopened note attachment in Acrobat, as shown in Table 4.18. The annotation will be opened if *open = 1*, and closed if *open = 0*.

The maximum length of *contents* is 65535 bytes. The maximum length of *title* is 255 characters *(PDFDocEncoding)*, or 126 Unicode characters. However, a practical limit of 32 characters for *title* is advised.

Table 4.18. Icon names for note annotations

icon name	*icon appearance*	*icon name*	*icon appearance*
comment		*newparagraph*	
insert		*key*	
note		*help*	
paragraph			

Note *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.*

4.7.7 Links

Table 4.19 lists relevant parameters for this section.

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

Table 4.19. 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 the PDF file *filename* at the rectangle specified by its lower left and upper right corners in default user space coordinates. *page* is the physical page number of the target page. *dest* specifies the destination zoom. It can attain one of the values specified in Table 4.20.

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

Add a link annotation with a target page in the current document at the rectangle specified by its lower left and upper right corners in default user space coordinates. *page* is the physical page number of the target page, and 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 generat-

ed; PDFlib will issue a warning message otherwise. *dest* specifies the destination zoom. It can attain one of the values specified in Table 4.20.

Table 4.20. Values for the dest parameter of PDF_add_pdflink( ) and PDF_add_locallink( ). The same values are also used for the openaction (see Section 4.7.1, »Document Open Action and Open Mode«) and bookmarkdest parameters (see Section 4.7.2, »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 (arbitrary file type) at the rectangle specified by its lower left and upper right corners in default user space coordinates. *filename* is the name of the file which will be launched upon clicking the link.

void PDF_add_weblink(PDF *p, float llx, float lly, float urx, float ury, const char *url)
Add a weblink annotation at the rectangle specified by its lower left and upper right corners in default user space coordinates. *url* is a Uniform Resource Identifier encoded in 7-bit ASCII specifying the link target. It can point to an arbitrary (Web or local) resource.

void PDF_set_border_style(PDF *p, const char *style, float width)
Set the border style for all kinds of annotations. These settings 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. Possible values of the style parameter are *solid* and *dashed*. If *width = 0* the links will be invisible.

void PDF_set_border_color(PDF *p, float red, float green, float blue)
Set the border color for all kinds of annotations. At the beginning of a document the annotation border color is set to *(0, 0, 0)*.

void PDF_set_border_dash(PDF *p, float b, float w)
Set the border dash style for all kinds of annotations (see *PDF_setdash( )*). 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*.

4.8 Page Size Formats

For the convenience of PDFlib users, Table 4.21 lists common standard page sizes[1].

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

Table 4.21. 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( )*. C macro definitions for these formats are available in *pdflib.h*

Note Page size definitions are only supplied for the C and C++ bindings. Other language clients may use the values provided in Table 4.21.

5 The PDFlib License

PDFlib is available under two different 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 and any EBCDIC editions (both 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:

- 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.

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 the PDFlib 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

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

The details of building and using shared libraries, also known as shared objects or dynamic link libraries (DLLs), are among the most frequently asked questions of PDFlib users. Most PDFlib language bindings require the use of shared libraries and may rely on additional shared libraries (these are zlib, libpng, and TIFFlib). 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 on Unix. Most problems with shared libraries are related to the variety of methods, options, and calls invented by Unix system vendors for implementing shared library support. In order to facilitate building and using shared libraries on a wealth of Unix systems, PDFlib leverages GNU libtool[1]. This is a collection of macros and shell scripts which attempt to »do the right thing« in order to build and use shared libraries on Unix systems.

While libtool support is completely integrated into the PDFlib configuration machinery, it is suggested to take a look at libtool and the corresponding documentation if you want to learn more about shared libraries.

Building shared libraries. Although we do not even attempt to completely cover the intricate details of shared libraries here, the hints given below may be helpful for PDFlib users. Examples for Linux and other Unix systems are shown in brackets.

- On many systems a compiler flag (Linux: *-fPIC*) must be used for modules which are intended to be linked into a shared library (so-called position-independent code, or PIC).
- Similarly, most systems require a special linker flag for shared libraries (Linux: *-shared*).
- The naming conventions for shared libraries vary (Linux and most others: *.so*, HP-UX: *.sl*)
- The system may or may not support a versioning system for shared libraries. 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 with or without an additional dot (Linux: *lib<name>.so.5*, BSDI: *lib<name>.so5*). The system may or may not consider version numbers when loading shared libraries.

The PDFlib configure script and GNU libtool try to 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.

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) is able to 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*). It contains a colon-separated list of

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

directories which are searched for shared libraries. Failing that, a cache file (see below) is consulted, and then some set of default system 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 shared libraries and sets up a cache file 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. Of course, the details vary among systems.

In order to find out the shared libraries required by a program or another shared library, 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.

Note *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. These items, along with the commands issued for the test and install targets will give you an idea of libtool's library deployment.*

To share or not to share? Note that while most Unix systems support shared libraries, not all do. According to the libtool documentation, building shared libraries is currently not supported on the following systems:

```
alpha-dec-osf2.1
i*86-*-bsdi3.1
i*86-*-bsdi3.0
i*86-*-bsdi2.1
i*86-pc-cygwin
m68k-next-nextstep3
m68k-sun-sunos4.1.1
mips-sgi-irix5.3
powerpc-ibm-aix4.1.5.0
```

PDFlib's *configure* mechanism will therefore build static versions of the library on those systems. This implies that only the C and C++ language bindings will be available.

Library versioning scheme used by libtool. If the operating system supports a versioning scheme for shared libraries libtool will make use of it, and create versioned libraries for PDFlib. It is very important to note that 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 version numbers 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) form one of the cornerstones of the Windows architecture. Building and using DLLs is very well understood, and generally doesn't pose any problems. The major exception to this rule is 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 all required DLLs into a single application-specific directory. If you want to move PDFlib DLLs 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

Note The PDFlib ActiveX edition takes care of these issues through a private installation directory and custom registry entries.

Shared Libraries on the Macintosh

Shared libraries on the MacOS are fully supported on PowerPC machines via the Code Fragment Manager (*CFM*). 68K Macs require an extension called *CFM68K* which will not be further discussed here. 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

B Summary of PDFlib Functions

General Functions

Function prototype	*page*
void PDF_boot(void)	*67*
void PDF_shutdown(void)	*67*
int PDF_get_majorversion(void)	*67*
int PDF_get_minorversion(void)	*67*
*PDF *PDF_new(void)*	*68*
*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)*	*68*
*void PDF_delete(PDF *p)*	*68*
*void *PDF_get_opaque(PDF *p)*	*69*
*int PDF_open_file(PDF *p, const char *filename)*	*69*
*int PDF_open_fp(PDF *p, FILE *fp)*	*69*
*void PDF_open_mem(PDF *p, size_t (*writeproc)(PDF *p, void *data, size_t size))*	*69*
*const char * PDF_get_buffer(PDF *p, long *size)*	*70*
*void PDF_close(PDF *p)*	*70*
*void PDF_begin_page(PDF *p, float width, float height)*	*70*
*void PDF_end_page(PDF *p)*	*70*
*float PDF_get_value(PDF *p, const char *key, float modifier)*	*71*
*void PDF_set_value(PDF *p, const char *key, float value)*	*71*
*const char * PDF_get_parameter(PDF *p, const char *key, float modifier)*	*71*
*void PDF_set_parameter(PDF *p, const char *key, const char *value)*	*71*

Text Functions

Function prototype	*page*
*int PDF_findfont(PDF *p, const char *fontname, const char *encoding, int embed)*	*72*
*void PDF_setfont(PDF *p, int font, float fontsize)*	*72*
*void PDF_show(PDF *p, const char *text)*	*74*
*void PDF_show2(PDF *p, const char *text, int len)*	*74*
*void PDF_show_xy(PDF *p, const char *text, float x, float y)*	*74*
*void PDF_show_xy2(PDF *p, const char *text, int len, float x, float y)*	*74*
*void PDF_continue_text(PDF *p, const char *text)*	*74*
*void PDF_continue_text2(PDF *p, const char *text, int len)*	*74*
*int PDF_show_boxed(PDF *p, const char *text, float x, float y, float width, float height, const char *mode, const char *feature)*	*74*
*float PDF_stringwidth(PDF *p, const char *text, int font, float size)*	*75*
*float PDF_stringwidth2(PDF *p, const char *text, int len, int font, float size)*	*75*
*void PDF_set_text_pos(PDF *p, float x, float y)*	*75*

Graphics Functions

Function prototype	page
void PDF_setdash(PDF *p, float b, float w)	76
void PDF_setpolydash(PDF *p, float *darray, int length)	76
void PDF_setflat(PDF *p, float flatness)	76
void PDF_setlinejoin(PDF *p, int linejoin)	76
void PDF_setlinecap(PDF *p, int linecap)	76
void PDF_setmiterlimit(PDF *p, float miter)	76
void PDF_setlinewidth(PDF *p, float width)	77
void PDF_save(PDF *p)	77
void PDF_restore(PDF *p)	78
void PDF_translate(PDF *p, float tx, float ty)	78
void PDF_scale(PDF *p, float sx, float sy)	78
void PDF_rotate(PDF *p, float phi)	78
void PDF_skew(PDF *p, float alpha, float beta)	78
void PDF_concat(PDF *p, float a, float b, float c, float d, float e, float f)	78
void PDF_moveto(PDF *p, float x, float y)	79
void PDF_lineto(PDF *p, float x, float y)	79
void PDF_curveto(PDF *p, float x1, float y1, float x2, float y2, float x3, float y3)	79
void PDF_circle(PDF *p, float x, float y, float r)	79
void PDF_arc(PDF *p, float x, float y, float r, float start, float end)	79
void PDF_rect(PDF *p, float x, float y, float width, float height)	80
void PDF_closepath(PDF *p)	80
void PDF_stroke(PDF *p)	80
void PDF_closepath_stroke(PDF *p)	80
void PDF_fill(PDF *p)	80
void PDF_fill_stroke(PDF *p)	80
void PDF_closepath_fill_stroke(PDF *p)	81
void PDF_endpath(PDF *p)	81
void PDF_clip(PDF *p)	81

Color Functions

Function prototype	page
void PDF_setgray_fill(PDF *p, float gray)	81
void PDF_setgray_stroke(PDF *p, float gray)	81
void PDF_setgray(PDF *p, float gray)	81
void PDF_setrgbcolor_fill(PDF *p, float red, float green, float blue)	81
void PDF_setrgbcolor_stroke(PDF *p, float red, float green, float blue)	81
void PDF_setrgbcolor(PDF *p, float red, float green, float blue)	82

Image Functions

Function prototype	page
int PDF_open_image_file(PDF *p, const char *type, const char *filename, const char *stringparam, int intparam)	82
int PDF_open_CCITT(PDF *p, const char *filename, int width, int height, int BitReverse, int K, int BlackIs1)	83
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)	83
void PDF_close_image(PDF *p, int image)	84
void PDF_place_image(PDF *p, int image, float x, float y, float scale)	85

Hypertext Functions

Function prototype	page
int PDF_add_bookmark(PDF *p, const char *text, int parent, int open)	86
void PDF_set_info(PDF *p, const char *key, const char *value)	86
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)	87
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)	87
void PDF_add_pdflink(PDF *p, float llx, float lly, float urx, float ury, const char *filename, int page, const char *dest)	88
void PDF_add_locallink(PDF *p, float llx, float lly, float urx, float ury, int page, const char *dest)	88
void PDF_add_launchlink(PDF *p, float llx, float lly, float urx, float ury, const char *filename)	89
void PDF_add_weblink(PDF *p, float llx, float lly, float urx, float ury, const char *url)	89
void PDF_set_border_style(PDF *p, const char *style, float width)	89
void PDF_set_border_color(PDF *p, float red, float green, float blue)	89
void PDF_set_border_dash(PDF *p, float b, float w)	89

Parameters and Values

category	function	keys
setup	set_parameter	prefix, resourcefile, compatibility, warning, flush
	set_value	compress
document	set_value	pagewidth, pageheight
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
hypertext	set_parameter	openaction, openmode, bookmarkdest, transition, base
	set_value	duration

Index

0-9

16-bit encoding 51
8-bit encodings 38

A

Acrobat 3 compatibility 11
Acrobat 4 compatibility 10
ActiveX binding
    general 15
Adobe Font Metrics (AFM) 43
AdobeStandardEncoding 39
AFM (Adobe Font Metrics) 43
Aladdin free public license 91
alpha channel 63
annotations 51, 87
API (Application Programming Interface)
    reference 66
AS/400 13, 39
ascender 55
ascender parameter 72
Asian FontPack 48
attachments 51, 87
Author field 86
availability of PDFlib 12

B

base parameter 88
baseline compression 58
Bézier curve 79
big-endian 52
bindings 12
BitReverse 83
BlackIs1 83
blind mode 57, 75
bold CJK text 50
BOM (Byte Order Mark) 52
bookmarkdest parameter 85
bookmarks 51, 85
builtin encoding 39
Byte Order Mark 52
byte ordering 52

C

C binding 15
    error handling 16
    memory management 31
    Unicode support 18
    version control 18
C++ binding 18
    error handling 19
    memory management 32
    Unicode support 20
    version control 20
capheight 55
capheight parameter 72
categories of resources 46
CCITT 60, 83
CFM (Code Fragment Manager) 95
character ID (CID) 47
character metrics 55
character names 41
character sets 38
characters per inch 55
charspacing parameter 73
Chinese 47, 49
CID fonts 47
CJK (Chinese, Japanese, Korean) 47
clip 37
CMaps 48, 49
code page
    IBM 1047 39
    Microsoft Windows 1250 41
    Microsoft Windows 1252 38
    Microsoft Windows 1254 41
color 37
color functions 81
COM (Component Object Model): see ActiveX
    binding
commercial license 91
compatibility
    Acrobat 4 10
    Acrobat Reader 10
compatibility parameter 67
compile_metrics utility 44
compress parameter 67
compression 10, 32
coordinate system 35, 77
    metric 35
    top-down 36
core fonts 38
CPI (characters per inch) 55
Creator field 86
current point 37
currentx and currenty parameter 79
custom encoding 40

D

default coordinate system 35

default zoom 85
descender 55
descender parameter 72
descriptor 43
DLL (dynamic link library) 93
document and page functions 69
document information fields 51, 86
document open action 85
downsampling 60
dpi calculations 60
drawing error 76
Dublin Core 86
duration parameter 87

E

EBCDIC 13, 15, 39
ebcdic encoding 39
EJB (Enterprise Java Beans) 22
embedding fonts 43
encoding 38
  CJK 51
  custom 40
  for hypertext 42
  Unicode 53
Encoding parameter 71
environment variable PDFLIBRESOURCE 47
error handling 14, 33
  API 68
  error names 34
  in C 16
  in C++ 19
  in Java 23
  in Perl 26
  in Python 27
  in Tcl 30
eServer zSeries and iSeries 13
Euro character 38, 42
exception: see error handling
external image references 62

F

features of PDFlib 9
file attachments 51, 87
fill 37
fillrule parameter 80
Flate compression 10
flush parameter 33, 67
font metrics 55
font parameter 71
FontAFM parameter 71
fontencoding parameter 71
fontname parameter 71
FontOutline parameter 71
FontPFM parameter 71
fonts
  AFM files 43
  Asian fontpack 48
  CID fonts 47
  CJK fonts 47
  descriptor 43
  embedding 43
  general 38
  glyph names 41
  legal aspects of embedding 44
  mono-spaced 55
  PDF core set 38
  PFA files 43
  PFB files 43
  PFM files 43
  PostScript 43
  resource configuration 45
  type 1 43
  type 3 43
  Unicode support 51
fontsize parameter 71
FontSpecific encoding 39
fontwarning parameter 72

G

general graphics state 76
get_value() 71
GIF 59, 82
graphics functions 76
graphics state 76, 77
grid.pdf 35

H

hello world example
  general 14
  in C 16
  in C++ 19
  in Java 22
  in Perl 25
  in Python 27
  in Tcl 29
horizontal writing mode 48, 50
horizscaling parameter 73
host encoding 39, 40
hypertext
  encoding 42
  functions 85

I

icons
  for file attachments 87
  for notes 88
ignoremask 64, 83
image data, re-using 62
image file formats 58
image functions 82
image mask 62
image references 62
image scaling 60

imagewarning parameter 58, 82
imagewidth and imageheight parameters 61, 82
inch 35
in-core PDF generation 32
invert 83
invisible text 56
iSeries 13
ISO 8859-1 38, 42
ISO 8859-15 41
ISO 8859-2 40, 41
ISO 8859-9 41

J

Japanese 47, 49
Java application servers 22
Java binding 20
  EJB 22
  error handling 23
  javadoc 21
  package 21
  servlet 22
  Unicode support 24
  version control 23
JPEG 10, 58, 82, 84

K

K parameter for CCITT images 83
Keywords field 86
Korean 47, 49

L

landscape mode 70
language bindings: see bindings
Latin 1 encoding 38, 42
Latin 2 encoding 40, 41
Latin 5 encoding 41
Latin 9 encoding 41
LD_LIBRARY_PATH 93
leading 55
leading parameter 73
licensing conditions 91
line spacing 55
links 88
little-endian 52
longjmp 17
LZW decompression 59

M

MacOS 13
macroman encoding 39
makepsres utility 45
mask 64, 83
masked 64, 83
masking images 62
memory images 62
memory management
  API 68
  general 31
  in C 31
  in C++ 32
memory, generating PDF documents in 32
metadata 86
metric coordinates 35
metrics 55
millimeters 35
mirroring 78
mono-spaced fonts 55
multi-page image files 64

N

nativeunicode parameter 53, 54, 73
non-proportional image scaling 61
note annotations 51, 87

O

openaction parameter 85
openmode parameter 85
ordering constraints 37
outline text 56
overline parameter 56, 73

P

page 65
page descriptions 35
page parameter 83
page size formats 89, 90
  limitations in Acrobat 37
page transitions 86
pagewidth and pageheight parameters 69
parameter
  ascender 72
  base 88
  bookmarkdest 85
  capheight 72
  charspacing 73
  compatibility 67
  compress 67
  currentx and currenty 79
  descender 72
  duration 87
  Encoding 71
  fillrule 80
  flush 33, 67
  font 71
  FontAFM 71
  fontencoding 71
  fontname 71
  FontOutline 71
  FontPFM 71
  fontsize 71
  fontwarning 72
  horizscaling 73

imagewarning 58, 82
imagewidth and imageheight 61, 82
leading 73
nativeunicode 53, 54, 73
openaction 85
openmode 85
overline 56, 73
pageheight and pagewidth 69
prefix 67
resourcefile 47, 67
resx and resy 82
strikeout 56, 73
textrendering 50, 56, 73
textrise 73
textx and texty 58, 73
transition 87
underline 56, 73
warning 34, 67
wordspacing 73
parameter handling functions 71
path 37
painting and clipping 80
segment functions 79
PDF 1.3 42
PDF_add_bookmark() 86
PDF_add_launchlink() 89
PDF_add_locallink() 88
PDF_add_note() 87
PDF_add_pdflink() 88
PDF_add_weblink() 89
PDF_arc() 79
PDF_attach_file() 87
PDF_begin_page() 70
PDF_boot() 67
PDF_circle() 79
PDF_clip() 81
PDF_close() 70
PDF_close_image() 84
PDF_closepath() 80
PDF_closepath_fill_stroke() 81
PDF_closepath_stroke() 80
PDF_concat() 78
PDF_continue_text() 74
PDF_continue_text2() 74
PDF_curveto() 79
PDF_delete() 68
PDF_end_page() 70
PDF_endpath() 81
PDF_fill() 80
PDF_fill_stroke() 80
PDF_findfont() 72
PDF_get_buffer() 32, 70
PDF_get_majorversion() 67
PDF_get_minorversion() 67
PDF_get_opaque() 69
PDF_get_parameter() 71
PDF_get_value() 71
PDF_lineto() 79
PDF_moveto() 79
PDF_new() 68
PDF_new2() 68
PDF_open_CCITT() 83
PDF_open_file() 69
PDF_open_fp() 69
PDF_open_image() 83
PDF_open_image_file() 82
PDF_open_mem() 69
PDF_place_image() 85
PDF_rect() 80
PDF_restore() 78
PDF_rotate() 78
PDF_save() 77
PDF_scale() 78
PDF_set_border_color() 89
PDF_set_border_dash() 89
PDF_set_border_style() 89
PDF_set_info() 86
PDF_set_parameter() 47
PDF_set_text_pos() 75
PDF_set_value() 71
PDF_setdash() 76
PDF_setflat() 76
PDF_setfont() 72
PDF_setgray() 81
PDF_setgray_fill() 81
PDF_setgray_stroke() 81
PDF_setlinecap() 76
PDF_setlinejoin() 76
PDF_setlinewidth() 77
PDF_setmiterlimit() 76
PDF_setpolydash() 76
PDF_setrgbcolor() 82
PDF_setrgbcolor_fill() 81
PDF_setrgbcolor_stroke() 81
PDF_show() 74
PDF_show_boxed() 56, 74
PDF_show_xy() 74
PDF_show_xy2() 74
PDF_show2() 74
PDF_shutdown() 67
PDF_skew() 78
PDF_stringwidth() 56, 75
PDF_stringwidth2() 75
PDF_stroke() 80
PDF_translate() 78
PDFDocEncoding 42
PDFlib
features 9
program structure 31
thread-safety 9, 15
pdflib.upr 47
PDFLIBRESOURCE environment variable 47
Perl binding 24
error handling 26
Unicode support 26
version control 26

PFA (Printer Font ASCII) 43
PFB (Printer Font Binary) 43
PFM (Printer Font Metrics) 43
Photoshop 84
platforms 12
PNG 58, 64, 82
PNG library 32, 58
Portable Document Format Reference Manual 92
PostScript fonts 43
PostScript Language Reference Manual 92
prefix parameter 67
print_glyphs.ps 41
Printer Font ASCII (PFA) 43
Printer Font Binary (PFB) 43
Printer Font Metrics (PFM) 43
program structure 31
Python binding 26
    error handling 27
    Unicode support 28
    version control 28

R

raster images
    functions 82
    general 58
raw image data 60, 83
references 92
reflection 78
resource category 46
resourcefile parameter 47, 67
resx and resy parameter 82
RGB color 37
rotating objects 35

S

S/390 13, 39
scaling images 60
scripting API 12
servlet 22
set_parameter() 47, 71
setjmp 17
setup functions 67
shared libraries 93
show_boxed() 74
skewing 78
special graphics state 77
standard output 69
standard page sizes 90
stdout channel 69
strikeout parameter 56, 73
stroke 37
structure of PDFlib programs 31
Subject field 86
subpath 37
subscript 55, 73
superscript 55, 73
SWIG 12
Symbol font 39

T

T1lib 44
Tcl binding 28
    error handling 30
    Unicode support 30
    version control 30
text box formatting 55
text functions 71
text handling 38
text metrics 55
text rendering modes 56
text variations 55
textrendering parameter 50, 56, 73
textrise parameter 73
textx and texty parameter 50, 58, 73
thread-safety 9, 15
thumbnails 85
TIFF 59, 82
    multi-page 64
TIFFlib 32, 59
Title field 86
top-down coordinates 36
transition parameter 87
transparency 62
    problems with 64
type 1 fonts 43
type 3 fonts 43

U

underline parameter 56, 73
Unicode 15, 51
    in C 18
    in C++ 20
    in Java 24
    in Perl 26
    in Python 28
    in Tcl 30
    problems with language bindings 53
units 35
UPR (Unix PostScript Resource) 45
    file format 45
    file searching 47
URL 62, 89
user space 35
UTF-16 52

V

value: see parameter
version control
    for shared libraries 94
    general 14
    in C 18
    in C++ 20
    in Java 23

in libtool 94
in Perl 26
in Python 28
in Tcl 30
vertical writing mode 48, 50

W

warning 31, 37, 39, 89
suppress 34
warning parameter 34, 67
weblink 89
winansi encoding 38, 39
wordspacing parameter 73
writing modes 48, 50

Z

ZapfDingbats font 39
ZIP compression 10
Zlib compression 10, 32, 58
zoom factor 85
zSeries 13