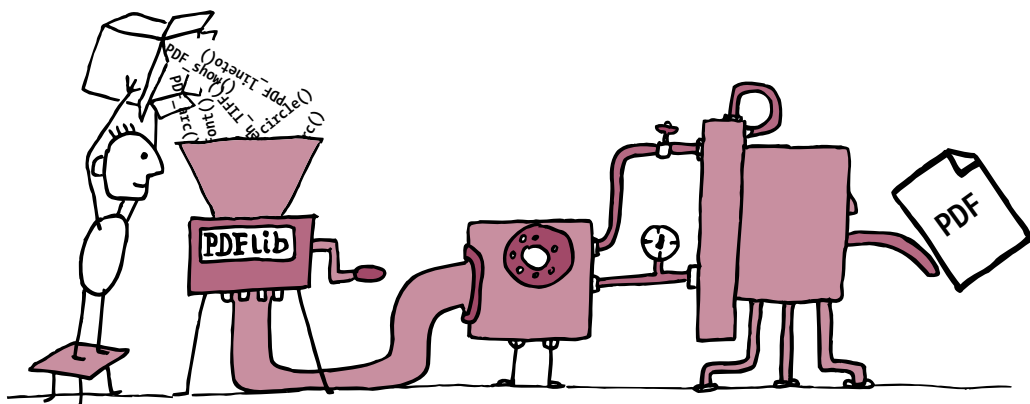


Thomas Merz

# PDFlib Reference Manual

Programming reference for PDFlib 2.01, a library for generating PDF on the fly



**thomasmerz**  
CONSULTING  
PUBLISHING

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

Copyright © 1997 - 1999 Thomas Merz. All rights reserved.

Thomas Merz Consulting & Publishing  
Tal 40, 80331 München, Germany  
<http://www.ifconnection.de/~tm>  
[tm@muc.de](mailto:tm@muc.de)  
fax +49/89/29 16 46 86

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

*Adobe and Acrobat are trademarks of Adobe Systems Incorporated. Microsoft and Windows are registered trademarks of Microsoft Corp. Macintosh is a trademark of Apple Computer. All other products or name brands are trademarks of their respective holders.*



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

*Revision history of this manual*

Date	Changes
Aug. 2, 1999	► Minor changes and additions for PDFlib 2.01
June 29, 1999	► Clarifications in the API description ► Separate sections for the individual language bindings ► Extensions for PDFlib 2.0 ► Enlarged the page size of the manual
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

*Version information on PDFlib itself can be found in the distribution.*

# Contents

## 1 Introduction 7

- 1.1 PDFlib Programming 7
- 1.2 PDFlib Features 8
- 1.3 Features not implemented in PDFlib 10

## 2 PDFlib Language Bindings 11

- 2.1 Overview of the PDFlib Language Bindings 11
  - 2.1.1 What's all the Fuss about Language Bindings? 11
  - 2.1.2 Availability and Special Considerations 12
  - 2.1.3 The »Hello world« Example 13
  - 2.1.4 Error Handling 13
  - 2.1.5 Memory Management 14
  - 2.1.6 Version Control 14
  - 2.1.7 Summary of the Language Bindings 15
- 2.2 C Binding 15
  - 2.2.1 How does the C Binding work? 15
  - 2.2.2 Availability and Special Considerations for C 15
  - 2.2.3 The »Hello world« Example in C 15
  - 2.2.4 Error Handling in C 16
  - 2.2.5 Memory Management in C 16
  - 2.2.6 Version Control in C 17
- 2.3 C++ Binding 17
  - 2.3.1 How does the C++ Binding work? 17
  - 2.3.2 Availability and Special Considerations for C++ 17
  - 2.3.3 The »Hello world« Example in C++ 18
  - 2.3.4 Error Handling in C++ 19
  - 2.3.5 Memory Management in C++ 19
  - 2.3.6 Version Control in C++ 19
- 2.4 Java Binding 19
  - 2.4.1 How does the Java Binding work? 19
  - 2.4.2 Availability and Special Considerations for Java 20
  - 2.4.3 The »Hello world« Example in Java 20
  - 2.4.4 Error Handling in Java 21
  - 2.4.5 Memory Management in Java 21
  - 2.4.6 Version Control in Java 21
- 2.5 Perl Binding 22
  - 2.5.1 How does the Perl Binding work? 22
  - 2.5.2 Availability and Special Considerations for Perl 22
  - 2.5.3 The »Hello world« Example in Perl 23
  - 2.5.4 Error Handling in Perl 23
  - 2.5.5 Memory Management in Perl 24

- 2.5.6 Version Control in Perl 24
- 2.6 Python Binding 24
  - 2.6.1 How does the Python Binding work? 24
  - 2.6.2 Availability and Special Considerations for Python 24
  - 2.6.3 The »Hello world« Example in Python 25
  - 2.6.4 Error Handling in Python 25
  - 2.6.5 Memory Management in Python 25
  - 2.6.6 Version Control in Python 26
- 2.7 Tcl Binding 26
  - 2.7.1 How does the Tcl Binding work? 26
  - 2.7.2 Availability and Special Considerations for Tcl 26
  - 2.7.3 The »Hello world« Example in Tcl 27
  - 2.7.4 Error Handling in Tcl 28
  - 2.7.5 Memory Management in Tcl 28
  - 2.7.6 Version Control in Tcl 28
- 2.8 Visual Basic Binding 28
  - 2.8.1 How does the Visual Basic Binding work? 28
  - 2.8.2 Availability and Special Considerations for Visual Basic 29
  - 2.8.3 The »Hello world« Example in Visual Basic 29
  - 2.8.4 Error Handling in Visual Basic 30
  - 2.8.5 Memory Management in Visual Basic 30
  - 2.8.6 Version Control in Visual Basic 30

## **3 Programming Concepts 31**

- 3.1 General Programming Issues 31
- 3.2 Coordinate Systems 31
- 3.3 Graphics and Text Handling 32
- 3.4 Font Handling 33
  - 3.4.1 The PDF Core Fonts 33
  - 3.4.2 Character Sets and 8-Bit Encoding 33
  - 3.4.3 Font Outline and Metrics Files 35
  - 3.4.4 Resource Configuration and the UPR Resource File 36
  - 3.4.5 Unicode Support 39
- 3.5 Image Handling 40
  - 3.5.1 Image File Formats 40
  - 3.5.2 Embedding Images in PDF 41
  - 3.5.3 Re-using Image Data 41
  - 3.5.4 Memory Images and External Image References 41
- 3.6 Error Handling 42

## **4 PDFlib API Reference 44**

- 4.1 General Functions 44
- 4.2 Text Functions 46
  - 4.2.1 Font Handling Functions 46
  - 4.2.2 Text Output Functions 47
- 4.3 Graphics Functions 49
  - 4.3.1 General Graphics State Functions 49
  - 4.3.2 Special Graphics State Functions 50
  - 4.3.3 Path Segment Functions 51
  - 4.3.4 Path Painting and Clipping Functions 51
- 4.4 Color Functions 52
- 4.5 Image Functions 52
- 4.6 Hypertext Functions 54
  - 4.6.1 Bookmarks 55
  - 4.6.2 Document Information Fields 55
  - 4.6.3 Page Transitions 55
  - 4.6.4 File Attachments 56
  - 4.6.5 Note Annotations 56
  - 4.6.6 Links 57
- 4.7 Convenience Stuff 58

## **5 The PDFlib License 59**

## **6 References 60**

### **Index 61**



# 1 Introduction

## 1.1 PDFlib Programming

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

**How can I use PDFlib?** PDFlib is available on a variety of platforms, including Unix, Windows NT, and MacOS. Although PDFlib itself is programmed in C, its functions can be accessed from several other languages and programming environments which we will call language bindings. The Application Programming Interface (API) is easy to learn, and is the same for all environments. Currently the following bindings are supported:

- ▶ ANSI C library (static or dynamic)
- ▶ ANSI C++ class via an object wrapper
- ▶ Perl
- ▶ Tcl
- ▶ Python
- ▶ Java
- ▶ Visual Basic

Besides, there are a number of additional PDFlib bindings which are supported by other people, most notably PHP3.

**What can I use PDFlib for?** PDFlib's primary target is creating dynamic PDF on the World Wide Web. Similar to HTML pages dynamically generated with a CGI script on the Web server, you may use a PDFlib program for dynamically generating PDF reflecting

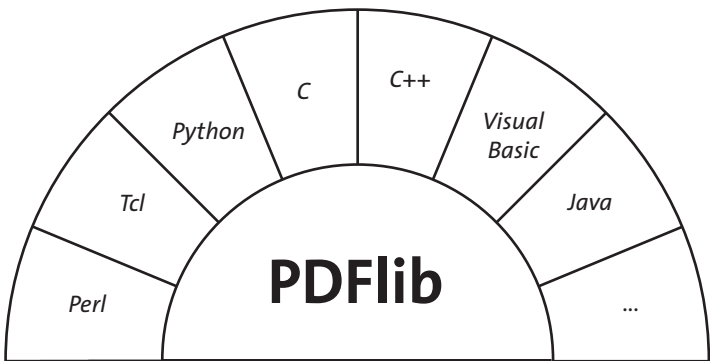


Fig. 1.1. PDFlib language bindings

user input or some other dynamic data, e.g. data retrieved from the Web server's database. The PDFlib approach offers several advantages as opposed to creating PDF from PostScript files with Acrobat Distiller:

- ▶ The PDFlib »driver« can be integrated directly in the application generating 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.
- ▶ PDFs need not be created ahead of time and stored on the server, but can be generated if needed. This is a big win not only if you want to deal with dynamic data which do not exist prior to the Web interaction, but also if large amounts of data have to be handled which make it impractical to pre-generate all the necessary PDF.
- ▶ PDFlib's thread-safety as well as its robust memory and error handling support the implementation of high-performance server applications.

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

**Requirements for using PDFlib.** PDFlib tries to make possible PDF generation 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 of her application data shouldn't have much trouble adapting to the PDFlib API which is 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 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 doesn't even attempt to explain Acrobat/PDF features or internals. Please refer to the Acrobat product literature, and the material cited at the end of the manual for further reference.

## 1.2 PDFlib Features

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

**About PDFlib-generated documents.** Generally, we strive to produce PDF documents which may be used with a wide variety of PDF consumers. According to common PDF practise, PDFlib generates binary compressed output compatible with Acrobat 3 and higher (although compression may be deactivated by the client).



Table 1.1. PDFlib features for generating PDF

Topic	Features
PDF Documents	<ul style="list-style-type: none"><li>▶ PDF documents of arbitrary length</li><li>▶ Arbitrary page size—each page may have a different size</li><li>▶ Compression for image data and file attachments</li></ul>
Vector graphics	<ul style="list-style-type: none"><li>▶ Common vector graphics primitives: lines, curves, arcs, rectangles, etc.</li><li>▶ Use vector paths for stroking, filling, and clipping</li><li>▶ RGB color for stroking and filling objects</li></ul>
Fonts	<ul style="list-style-type: none"><li>▶ Text output in different fonts and different encodings (character sets)</li><li>▶ Built-in font metrics for PDF's 14 base fonts</li><li>▶ Font embedding</li><li>▶ Use AFM files for font metrics</li><li>▶ Support for the Euro symbol</li><li>▶ Library clients can retrieve character widths for exact formatting</li><li>▶ Flexible font and metrics file configuration via UPR file</li></ul>
Hypertext	<ul style="list-style-type: none"><li>▶ Page transition effects such as shades and mosaic</li><li>▶ Nested bookmarks</li><li>▶ PDF links, launch links (other document types), Web links, configurable link border style and color</li><li>▶ Document information: four standard fields (Title, Subject, Author, Keywords) plus user-defined info field (e.g., part number)</li><li>▶ File attachments</li><li>▶ Note annotations</li></ul>
Unicode	<ul style="list-style-type: none"><li>▶ Bookmarks (e.g., Greek or Russian)</li><li>▶ Contents and title of text annotations</li><li>▶ Document information fields, including user-defined field</li><li>▶ Attachment description and author name</li></ul>
Images	<ul style="list-style-type: none"><li>▶ Embed images in GIF, TIFF, JPEG, or CCITT file formats</li><li>▶ Images constructed by the client directly in memory</li><li>▶ Re-use image data, e.g., for repeated logos on each page</li></ul>
Pro-gramming	<ul style="list-style-type: none"><li>▶ Thread-safe: suited for deployment in multi-threaded server applications</li><li>▶ Configurable error handler, integrated with the host language's exception handling where available</li><li>▶ Configurable memory management procedures</li><li>▶ Configurable debugging facilities</li><li>▶ No memory leaks as confirmed by »Purify« check</li></ul>

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

PDFlib doesn't offer any compatibility option regarding the generated PDF. Asking PDFlib functions to produce one of the above-mentioned features implies that the PDF output will require Acrobat 4 (PDF 1.3) for proper use (although the output may be used with restrictions in Acrobat 3).

Table 1.2. PDFlib features which require Acrobat 4

Topic	Remarks
Hypertext	► File attachments are not recognized in Acrobat 3 (require full Acrobat 4) ► Different icons for notes are not recognized in Acrobat 3
Page size	► Acrobat 4 extends the limits for acceptable PDF page sizes
Unicode	► Unicode text doesn't work in Acrobat 3
Font	► The Euro symbol is not supported 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.

### 1.3 Features not implemented in PDFlib

Table 1.3 lists PDF features which are currently not implemented in PDFlib.

Table 1.3. Features which are currently not implemented in PDFlib

Feature	Remarks
Dealing with existing PDFs	PDFlib generates new PDF documents, but doesn't integrate or manipulate existing PDF content.
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.
Font subsetting	Font subsetting requires extended font processing.
TrueType fonts	Embedding TrueType fonts in PDF is a delicate and error-prone process. PDFlib currently doesn't support any kind of TrueType embedding.

## 2 PDFlib Language Bindings

This chapter is meant to give you a jump start to programming PDFlib in one or more of the supported languages. The first section gives a general overview, while each of the following sections will cover a particular language binding. The suggested reading order is to take a look at Section 2.1, »Overview of the PDFlib Language Bindings«, and subsequently pick the section(s) describing your favorite language binding(s).

### 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), give rise to a new quality in software portability (such as Java), or provide the glue among many different technologies while being completely operating system specific (such as Visual Basic).

It is our firm believe that a generic library such as PDFlib benefits very much from supporting a wide range of programming environments, thereby enlarging the potential user base while giving everyone the freedom to pick his favorite language for solving the particular problem at hand. This means you can call PDFlib routines without any C programming by simply writing a couple of script language instructions. PDFlib scripting greatly simplifies small to medium programming tasks, and is appropriate in many application areas in which the development, build, and debug overhead of C is considered too high.

This goal gives rise to quite a new issue in software portability. Instead of porting a given program in a given language to many different platforms, we are trying to maintain a coherent programming interface across many different languages! Keeping this in mind is very important when dealing with the PDFlib API. Despite the fact that early incarnations of the library supported some scripting languages, the programming interface has been very C-centric. Starting with PDFlib 2.0, the internals as well as the interface have been redesigned, so that they are usable from a wide variety of languages. The goal of multi-language portability also explains some properties of the interface which may be considered quirky in a pure C environment. In case you wonder about a particular interface feature, or some »obvious« enhancement to the interface comes to your mind, please reconsider the issue, taking into account the compatibility of your proposed enhancement to all supported languages.

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. The facilities provided for extension developers are numerous, and differ significantly among the languages. Given the amount of changes occurring in actively developed software, and the number of supported languages, this may quickly result in a maintenance nightmare, especially when we take the number of supported platforms into account.

Fortunately enough, this isn't the case due to a cute facility called SWIG<sup>1</sup> (Simplified Wrapper and Interface Generator), written by Dave Beazley <beazley@cs.uchicago.edu>. SWIG is brilliant in design and implementation. With the help of SWIG, PDFlib can easily be integrated into the Perl, Tcl, and Python scripting languages, and even Java with a hacked-up version of SWIG.

It's important to note that you don't need to install or use SWIG in order to make use of PDFlib scripting. All files necessary for a certain language binding are contained within the PDFlib distribution. By the way, SWIG support for PDFlib was suggested and in its first incarnation implemented by Rainer Schaaf <Rainer.Schaaf@t-online.de><sup>2</sup>.

For other language bindings not supported by SWIG it is either rather obvious what to do (such as C++), or a matter of digging through the relevant interface specifications and implementing the necessary glue material manually (such as Visual Basic).

**PDFlib Scripting API.** In order to avoid duplicating the PDFlib API reference manual for all supported scripting languages, this manual is considered authoritative not only for the C binding but also for the other languages. 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 should be a straightforward process. Alas, I was able to translate a C PDFlib application to Perl by simply deleting the include directives and adding a bunch of dollar signs to all variable names!

### 2.1.2 Availability and Special Considerations

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 have been tested. However, we strive to make the latest available versions of the respective environments work with PDFlib. Table 2.1 lists the language/platform combinations we used for testing.

Table 2.1. Tested language and platform combinations

Language	Unix (Linux and others)	Windows NT 4.0	MacOS 8.6 PPC
ANSI C	GCC (egcs-2.1.1) and other C compilers	Microsoft Visual C++ 6 and Watcom C 10.6	Metrowerks CodeWarrior 4
ANSI C++	GCC (egcs-2.1.1)	Microsoft Visual C++ 6 and Watcom C 10.6	Metrowerks CodeWarrior 4
Java	JDK 1.1.7	JDK 1.1.8 and 1.2.1	Macintosh Runtime for Java (MRJ) 2.1, based on JDK 1.1.7
Perl	Perl 5.005_03	ActivePerl build 517, based on Perl 5.005_03	MacPerl 5.2.0r4, based on Perl 5.004
Python	Python 1.5	Python 1.5.2	Python 1.51
Tcl	Tcl 8.1	Tcl 8.1	Tcl 8.1a2
Visual Basic	–	Visual Basic 6.0	–

**Note** Currently only the C language binding is fully regression-tested for a PDFlib release. Also, the C API is expected to be more stable than the scripting APIs since we intend to integrate the PDFlib scripting APIs into the native language paradigms more smoothly.

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!

Although all language bindings make use of shared libraries on Unix or dynamic link libraries (DLLs) on Windows, we don't make any attempt to build a unified shared library which can be used for one or more language bindings at the same time. Instead, each language binding will be served by a separate library. There are several technical reasons for such a strict distinction, and we expect the future life of PDFlib applications to be much easier that way.

### 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

Errors of a certain kind are called exceptions in several 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, or supplying NULL pointers for required string arguments)

If the library detects such an exceptional situation, a central error handler is called in order to deal with the situation, instead of passing special return values along the call stack.

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 subsequent sections below. Generally, we strive to promote the error to the respective language's native exception handling mechanism (if any), or let the library client decide what to do by installing his own error handler in PDFlib. 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 language-specific exceptions is based on SWIG as far as SWIG supports it; for other languages, we cooked up our own exception propagation mechanism.

If no special precautions like installing one's own error handler, or catching exceptions are taken, the default action is to issue an appropriate message on the standard

output channel and exit on fatal errors. The PDF output file may 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 messages in a GUI dialog box, and take other measures instead of aborting. More information on installing a custom error handler can be found in the respective language sections below; the details of implementing a custom error handler are discussed in Section 3.6, »Error Handling«.

## 2.1.5 Memory Management

A library such as PDFlib dynamically allocates and frees lots of small and large memory chunks.

*Note This is probably the right place to mention that due to rigorous testing, debugging, and »Purifying«<sup>1</sup> PDFlib is known not to lose a single allocated byte during a test run which generates thousands of PDF pages.*

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 *free/restore*) may in some language bindings be replaced by external procedures provided by the client. These procedures will be called for all PDFlib-internal memory allocation or deallocation.

It's generally 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 will mainly be of interest to the C or C++ programmer.

## 2.1.6 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, where 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.

Generally, applications written for PDFlib 2.0 or higher are expected to be compatible with later versions of the library.

<sup>1</sup> See <http://www.rational.com>

## 2.1.7 Summary of the Language Bindings

For easy reference, Table 2.2 summarizes important features of the available PDFlib language bindings. More details can be found in the respective section of this manual and further documentation in the PDFlib distribution.

Table 2.2. Summary of the language bindings

Language	Custom error handling	Custom memory management	Version control	thread-safe
C	client-supplied error handler	client-supplied functions	manually	yes
C++	client-supplied error handler	client-supplied functions	manually	yes
Java	Java exceptions	–	automatically	–
Perl	Perl exceptions	–	via package mechanism	–
Python	Python exceptions	–	manually	–
Tcl	Tcl exceptions	–	via package mechanism	–
Visual Basic	client-supplied error handler	–	unique GUID and type library	–

## 2.2 C Binding

### 2.2.1 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 special issues related to the calling convention 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.

### 2.2.2 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.

### 2.2.3 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: cannot open PDF file hello_c.pdf.\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", "default", 0);
    if (font == -1) {
        fprintf(stderr, "Couldn't set font!\n");
        exit(3);
    }

    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*/

    exit(0);
}

```

## 2.2.4 Error Handling in C

As noted in Section 2.1.4, »Error Handling«, PDFlib clients may install a custom error handler. This feature is mainly intended for clients written in the C or C++ language. It requires opening the PDF object with the *PDF\_new2()* function, as opposed to the simpler call *PDF\_new()*. A detailed description of the error handling machinery can be found in Section 3.6, »Error Handling«.

## 2.2.5 Memory Management in C

Similar to error handling, memory management for PDFlib can be completely delegated to client-supplied routines, which have to 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 *alloc/free/realloc* semantics, but may choose an arbitrary implementation. All routines will be supplied with a pointer to the calling PDF object. The single exception is that the very first call to the allocation routine will supply a PDF pointer of NULL. Client-provided memory allocation routines must be prepared to deal with such a NULL 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-specific data inside the PDF object, for use in memory management or error handling routines.

The signature of the memory management routines can be found in Section 4.1, »General Functions«.

## 2.2.6 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 version 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.

In addition, the PDFlib library file name will contain major and minor version numbers on some platforms.

## 2.3 C++ Binding

### 2.3.1 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 to 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->* object reference. Keep this in mind when reading the PDFlib API descriptions.

### 2.3.2 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 compilers.

In most environments there are inherent issues related to C++ deployment in 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).

### 2.3.3 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: cannot open PDF file hello_cpp.pdf.\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", "default", 0);
    if (font == -1) {
        fprintf(stderr, "Couldn't set font!\n");
        exit(3);
    }

    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;
```

```
    exit(0);  
}
```

### 2.3.4 Error Handling in C++

Taking the close relationship between C and C++ into account, it's amazing that there doesn't seem to be a portable way of integrating PDFlib's C-based error handling with the exception handling of C++. Although some combinations of *setjump/longjump* and C++ exceptions work with some compilers, we were unable to come up with a robust and portable solution to this problem. For this reason we stick to the C method of supplying an error handler when a PDF object is generated. An optional client-provided error handler can be supplied in the PDF constructor, which has the same signature as the *PDF\_new2()* function.

It's important to note that 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 C++ side, but has to deal with the C structure. For this reason, C++ error handlers must use the (rather private) data type name *PDF\_c* although the signature in the API reference calls for the *PDF* data type.

### 2.3.5 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 (see above).

### 2.3.6 Version Control in C++

Version control for the C++ binding is identical to the C binding.

## 2.4 Java Binding

### 2.4.1 How does the Java Binding work?

Starting with the Java<sup>1</sup> Development Kit (JDK) 1.1, Java supports a portable mechanism for attaching native language code to Java programs, the so-called Java Native Interface (JNI)<sup>2</sup>. The JNI provides programming conventions for calling native 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.

Fortunately, Harco de Hilster has provided a hacked version of SWIG which implements JNI support simply as another SWIG module. SWIG generates a C wrapper file, as well as a Java class definition file. This technique allows us to attach PDFlib to Java by

1. See <http://java.sun.com>

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

simply generating a wrapper file with SWIG, and loading the shared library from Java. 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.

## 2.4.2 Availability and Special Considerations for Java

Obviously, for developing Java applications you will need the JDK which includes support for the JNI. For compiling the PDFlib-supplied JNI wrapper file, you will need the Java include files for C.

Although Sun only delivers the JDK (and therefore the JNI) to the Win32 and Solaris platforms, the JDK has successfully been ported by the Linux community.<sup>1</sup> The Mac OS Runtime for Java (MRJ)<sup>2</sup>, version 2.0 and above, also supports the JNI.

For the PDFlib binding to work, the Java VM must have access to the PDFlib shared library and the PDFlib Java class file:

- ▶ On Unix systems the library name supplied in the PDFlib Java class file will be decorated according to the system's conventions for the names of shared libraries (usually by transforming *pdf\_java2.01* to *libpdf\_java2.01.so*). The library must be placed in one of the default locations for shared libraries (e.g., the */lib* directory), or in an appropriately configured directory (most systems use the `LD_LIBRARY_PATH` environment variable). The PDFlib class file *pdflib.java* must be compiled to *pdflib.class* and must be accessible via the `CLASSPATH` environment variable.
- ▶ On Windows systems the library name supplied in the PDFlib Java class file will be decorated according to the usual Windows conventions for DLLs (by transforming *pdf\_java2.01* to *pdf\_java2.01.dll*). The DLL must be placed in the Windows system directory, the current directory, or a directory which is listed in the `PATH` environment variable. The PDFlib class file *pdflib.java* must be compiled to *pdflib.class* and must be accessible via the `CLASSPATH` environment variable.
- ▶ On the Mac the library name supplied in the PDFlib Java class file is used without any change (*pdf\_java2.01*). 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. The PDFlib class file *pdflib.java* must be compiled to *pdflib.class* and must be accessible via the `CLASSPATH` environment variable.

## 2.4.3 The »Hello world« Example in Java

```
/* hello.java
 * Copyright (C) 1997-99 Thomas Merz. All rights reserved.
 *
 * PDFlib client: hello example in Java
 */

import pdflib.*;

public class hello
{
    public static void main (String argv[])
```

<sup>1</sup> See <http://www.blackdown.org>

<sup>2</sup> See <http://devworld.apple.com/java>

```

{
    long p;
    int font;

    p = pdflib.PDF_new();

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

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

    pdflib.PDF_begin_page(p, 595, 842);

    font = pdflib.PDF_findfont(p, "Helvetica-Bold", "default", 0);
    if (font == -1){
        System.err.println("Couldn't find font!\n");
        System.exit(1);
    }
    pdflib.PDF_setfont(p, font, 18);

    pdflib.PDF_set_text_pos(p, 50, 700);
    pdflib.PDF_show(p, "Hello world!");
    pdflib.PDF_continue_text(p, "(says Java)");
    pdflib.PDF_end_page(p);

    pdflib.PDF_close(p);
    pdflib.PDF_delete(p);
}
}

```

#### 2.4.4 Error Handling in Java

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

```

try {
    p = pdflib.PDF_new();
} catch (Throwable e) {
    System.err.println("Exception caught:\n" + e);
    return;
}

```

#### 2.4.5 Memory Management in Java

There are currently no provisions for client-side memory management, or for integrating PDFlib's internal memory management into the Java VM.

#### 2.4.6 Version Control in Java

Version control for the Java binding is done transparently when loading the shared library. The Java code for loading the PDFlib shared library relies on the major and minor

version numbers being contained in the library file name. This means that an exact match of the library version is required.

## 2.5 Perl Binding

### 2.5.1 How does the Perl Binding work?

Perl<sup>1</sup> supports a mechanism for extending the language (interpreter) via native C libraries. SWIG frees us from the chore of writing the necessary C interface files. In the case of Perl, SWIG generates a C wrapper file and a perl package module. The C module and the core PDFlib library are linked to 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 *use* and *package* statements.

### 2.5.2 Availability and Special Considerations for Perl

The Perl extension mechanism works by loading dynamic libraries at runtime, or statically linking the extension library to the Perl archive library. However, only the shared library method has been tested with PDFlib (instructions on building statically extended versions of the Perl interpreter can be found in the SWIG and Perl documentation).

In order to compile the PDFlib-supplied Perl wrapper file, you will need to have the Perl sources installed because the wrapper file needs the *EXTERN.h*, *perl.h*, and *XSUB.h* header files from the Perl source file set.

For the PDFlib binding to work, the Perl interpreter must have access to the PDFlib shared library and the module file *pdflib.pm*:

- ▶ On Unix systems both *pdflib.so* and *pdflib.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*. PDFlib's install mechanism will place the files in the correct directories. The PDFlib base shared library *pdflib2.01.so* must also be accessible. Typical output of the above command (on a Linux system) looks like

```
/usr/lib/perl5/site_perl/5.005/i686-linux
```

- ▶ On the Windows platform PDFlib supports the ActiveState port of Perl 5 to Windows, also known as ActivePerl.<sup>2</sup> Both *pdflib.dll* and *pdflib.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*. Typical output of the above command looks like

```
C:\Program Files\Perl5.005\site\lib
```

- ▶ For the Mac it should first be noted that shared libraries are by default only supported on the PowerPC platform. 68K-based machines require installing a facility called *CFM68K* (a description of this is beyond the scope of this manual). PDFlib supports

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

2. See <http://www.activestate.com>

the Macintosh port of Perl known as MacPerl<sup>1</sup>. Both the shared library *pdflib* and *pdflib.pm* will be found if placed in the current folder, or in one of the following folders:

```
<MacPerl>:lib:auto:pdflib
<MacPerl>:lib:MacPPC:auto:pdflib
```

where *<MacPerl>* denotes the Perl installation folder. The module file *pdflib.pm* may also be placed directly in the *lib* folder. In order to run the supplied samples, start Perl and open the script via »Script«, »Run Script«. It should be noted that the generated PDF output ends up in the Perl interpreter's folder if a relative file name is supplied as in the sample scripts.

### 2.5.3 The »Hello world« Example in Perl

```
#!/usr/bin/perl
# hello.pl
# Copyright (C) 1997-99 Thomas Merz. All rights reserved.
#
# PDFlib client: hello example in Perl
#

use pdflib 2.01;
package pdflib;

$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", "default", 0);
die "Couldn't set font" if ($font == -1);

PDF_setfont($p, $font, 18.0);

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

PDF_delete($p);
```

### 2.5.4 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 { $p = PDF_new() };
die "Exception caught" if $@;
```

1. See <http://www.macperl.com>

## 2.5.5 Memory Management in Perl

There are currently no provisions for client-side memory management, or for integrating PDFlib's internal memory management into the Perl interpreter.

## 2.5.6 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 2.01;
```

in order to make sure they will get the required library version (or a newer one). Since the library version number is contained in the package file, there is no need to retain the version number in the file name of the shared library.

# 2.6 Python Binding

## 2.6.1 How does the Python Binding work?

Python<sup>1</sup> supports a mechanism for extending the language (interpreter) via native C libraries. SWIG frees us from the chore of writing the necessary C interface files. In the case of Python, SWIG generates a single C wrapper file. The C module is linked to 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.

## 2.6.2 Availability and Special Considerations for Python

The Python extension mechanism works by loading dynamic libraries at runtime, or statically linking the extension library to the Python archive library. However, only the shared-library method has been tested with PDFlib (instructions on building statically extended versions of the Python interpreter can be found in the SWIG documentation).

In order to compile the PDFlib-supplied Python wrapper file, you will need to have the Python sources installed because the wrapper file needs the *Python.h* header file from the Python source file set. On the Mac, it suffices to install the Python Developer Kit instead of the full source package.

For the PDFlib binding to work, the Python interpreter must have access to the PDFlib shared library:

- ▶ On Unix systems the PDFlib shared library for Python *pdflib.so* will be searched in the directories listed in the PYTHONPATH environment variable. The PDFlib base shared library *pdflib2.01.so* must also be accessible.
- ▶ On Windows systems the PDFlib shared library *pdflib.dll* will be searched in the directories listed in the PYTHONPATH environment variable.
- ▶ On the Mac the PDFlib shared library *pdflib.ppc.slb* will be searched in the *Plugins* folder of the Python application folder.

<sup>1</sup> See <http://www.python.org>



## 2.6.3 The »Hello world« Example in Python

```
#!/usr/bin/python
# hello.py
# Copyright (C) 1997-99 Thomas Merz. All rights reserved.
#
# PDFlib client: hello example in Python
#

from sys import *
from pdflib 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", "default", 0)
if font == -1:
    print 'Couldn\'t set font!'
    exit(3);

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);
```

## 2.6.4 Error Handling in Python

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

```
try:
    p = PDF_new();
except:
    print 'Exception caught!'
```

## 2.6.5 Memory Management in Python

There are currently no provisions for client-side memory management, or for integrating PDFlib's internal memory management into the Python interpreter.

## 2.6.6 Version Control in Python

We are currently not aware of any intrinsic versioning scheme available in Python (if you know better, please let us know). Currently PDFlib applications in Python must use manual version control.

## 2.7 Tcl Binding

### 2.7.1 How does the Tcl Binding work?

Tcl<sup>1</sup> supports a mechanism for extending the language (interpreter) via native C libraries. SWIG frees us from the chore of writing the necessary C interface files. In the case of Tcl, SWIG generates a single C wrapper file. The C module is linked to 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.

### 2.7.2 Availability and Special Considerations for Tcl

The Tcl extension mechanism works by loading dynamic libraries at runtime, or statically linking the extension library to the Tcl archive library. However, only the shared-library method has been tested with PDFlib (instructions on building statically extended versions of the Tcl interpreter can be found in the SWIG documentation). For extending the Tcl interpreter with PDFlib, Tcl 7.5 or higher is recommended.

In order to compile the PDFlib-supplied Tcl wrapper file, you will need to have the Tcl sources installed because the wrapper file needs the *tcl.h* and *tk.h* header files from the Tcl source file set.

For the PDFlib binding to work, the Tcl shell must have access to the PDFlib shared library (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*:

- ▶ On Unix systems the library name *pdflib.so* supplied in the *pkgIndex.tcl* file must be placed in one of the default locations for shared libraries (e.g., the */lib* directory), or in an appropriately configured directory (most systems use the *LD\_LIBRARY\_PATH* environment variable). The PDFlib base shared library *pdflib2.01.so* must also be accessible.
- ▶ Unfortunately, Tcl doesn't itself produce a platform-specific decoration of the library name. On Windows you have to change the library name *pdflib.so* supplied in the *pkgIndex.tcl* file to the appropriate name *pdflib.dll*. 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 index file (and the DLL) will be searched for in the directories

```
C:\Program Files\Tcl 8.1\lib\tcl8.1\  
C:\Program Files\Tcl 8.1\lib\tcl8.1\pdflib
```

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

<sup>1</sup> See <http://www.scriptics.com> and [www.tclconsortium.org](http://www.tclconsortium.org)

```
System:Extensions:Tool Command Language:tcl8.1
System:Extensions:Tool Command Language:tcl8.1: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.

## 2.7.3 The »Hello world« Example in Tcl

```
#!/bin/sh
#
# hello.tcl
# Copyright (C) 1997-99 Thomas Merz. All rights reserved.
#
# 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 2.01

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 "default" 0 ]

if { $font == -1 } {
    puts stderr "Couldn't set font!"
    exit
}
    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
```

## 2.7.4 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 { set p [PDF_new] } result ] {  
    puts stderr "Exception caught!"  
    puts stderr $result  
}
```

## 2.7.5 Memory Management in Tcl

There are currently no provisions for client-side memory management, or for integrating PDFlib's internal memory management to the Tcl interpreter.

## 2.7.6 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 2.01
```

in order to make sure they will get the required library version (or a newer one, which is ok for PDFlib). Since the library version number is contained in the package file, there is no need to retain the version number in the file name of the shared library.

# 2.8 Visual Basic Binding

## 2.8.1 How does the Visual Basic Binding work?

Visual Basic<sup>1</sup> applications can easily access functions exported from a Dynamic Link Library (DLL), provided the function interface is supplied via a Type Library. Ideally, SWIG would also take care of this case and generate the necessary glue files. Unfortunately, a SWIG module for Visual Basic is currently not available. The PDFlib distribution therefore contains a hand-written module definition file *pdflib.def* which describes the functions exported from the PDFlib DLL, and a function description file *pdflib.idl* written in the Interface Definition Language (IDL). While the module definition file is used for generating the DLL, the IDL file is used for creating the PDFlib Type Library *pdflib\_vb.tlb*. Also, the *pdflib.h* header file has been prepared to declare the necessary export rules as well as the calling conventions required by Visual Basic. IDL implies using Microsoft's ID scheme for uniquely identifying an interface (see below).

In order to use the PDFlib DLL from a Visual Basic application, the PDFlib type library must be loaded into the VB project. This frees us from having to declare all PDFlib functions within the Visual Basic code. Since the PDFlib type library is supplied along with help strings for all functions, development tools with object browsers, such as Visual

<sup>1</sup> Visual Basic is a commercial product of Microsoft and not freely available. Visual Basic should not be confused with (although it's closely related to) Visual Basic for Applications (which is integrated in several Microsoft and third-party applications) or VBScript (which is integrated in some Web server and browser products). More information about Visual Basic can be found at <http://msdn.microsoft.com/vbasic/prodinfo>.

Basic 6.0, will let you browse the PDFlib functions and display short descriptions for each function.

## 2.8.2 Availability and Special Considerations for Visual Basic

In order to not confuse the PDFlib DLL for Visual Basic and the PDFlib C/C++ DLL, the former uses a file name of *pdflib\_vb.dll*. This file name is published via the type library, so registering the PDFlib type library or including it in your Visual Basic project is sufficient. In the Visual Basic 6 environment, this can be done by clicking *Project, References, Browse*, and pointing to the PDFlib type library (*pdflib\_vb.tlb*). The PDFlib Visual Basic DLL must be available in one of the directories searched by Windows, i.e., the current directory, the Windows and Windows\system32 directories, and the directories listed in the PATH environment variable.

*Note The location of the PDFlib DLL cannot easily be changed after registering PDFlib without breaking the connection between type library and DLL.*

## 2.8.3 The »Hello world« Example in Visual Basic

```
Attribute VB_Name = "Module1"
'
' hello.bas
' Copyright (C) 1997-99 Thomas Merz. All rights reserved.
'
' PDFlib client: hello example in Visual Basic
' Requires the PDFlib type library
' Load pdflib_vb.tlb via Project, References, Browse
```

Option Explicit

```
Sub main()
    Dim p As Long
    Dim err, font As Integer

    p = PDF_new

    ' Open new PDF file
    err = PDF_open_file(p, "hello_vb.pdf")
    If (err = -1) Then
        MsgBox "Couldn't open PDF file!"
    End
End If

PDF_set_info p, "Creator", "hello.bas"
PDF_set_info p, "Author", "Thomas Merz"
PDF_set_info p, "Title", "Hello, world (Visual Basic)!"

' start a new page
PDF_begin_page p, 595, 842

font = PDF_findfont(p, "Helvetica-Bold", "winansi", 0)
If (font = -1) Then
    MsgBox "Couldn't set font"
End
End If
```

```

PDF_setfont p, font, 24

PDF_set_text_pos p, 50, 700
PDF_show p, "Hello, world!"
PDF_continue_text p, "(says Visual Basic)"

PDF_end_page p' finish page

PDF_close p ' close PDF document

PDF_delete p
End Sub

```

## 2.8.4 Error Handling in Visual Basic

Due to the nature of the VB/DLL integration it's possible to call Visual Basic functions from C code via callbacks. This feature can be used in order to install a PDFlib error handler written in Visual Basic along the following lines:

```

Public Sub ErrorHandler(ByVal p As Long, ByVal typ As Integer, _
    ByRef msg As String)
    MsgBox "PDFlib Error:" & typ
End Sub

End Sub

...

p = PDF_new2(AddressOf ErrorHandler, 0#, 0#, 0#, 0#)

```

The address of the VB error handler is being passed to PDFlib in a call to *PDF\_new2()*. However, we were unable to find a way to access the DLL-supplied error message string within the error handler. Although it compiles fine, the whole thing crashes as soon as we try to access the error string inside the VB error handler.

## 2.8.5 Memory Management in Visual Basic

There are currently no provisions for client-side memory management, or for integrating PDFlib's internal memory management into the Visual Basic machinery.

## 2.8.6 Version Control in Visual Basic

Instead of simple major and minor version numbers, type libraries support the concept of a globally unique identifier (GUID) which uniquely describes a particular programming interface. Instead of messing around with different version numbers, a new software release may decide whether or not to actually support a certain interface identified via its GUID.

The PDFlib DLL for Visual Basic makes use of the GUID mechanism. The GUID for PDFlib 2.01 can be found in *pdflib\_vb.idl*, although it will only rarely be used directly.

# 3 Programming Concepts

## 3.1 General Programming Issues

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

PDLflib 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 PDLflib, since these usually point to problems in the client's ordering of function calls. PDLflib will throw an exception if bad parameters are supplied by a library client.

## 3.2 Coordinate Systems

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

$1\text{ pt} = 1\text{ inch} / 72 = 25.4\text{ mm} / 72 = 0.3528\text{ mm}$

PDLflib client programs may change the default user space by rotating, scaling, or translating, resulting in new user coordinates. The respective functions for these transformations are `PDF_rotate()`, `PDF_scale()`, and `PDF_translate()`. 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.

*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.*

Although PDF and PDLflib don't impose any restriction on the usable page size, Acrobat implementations suffer from architectural limits concerning the page size. Note that other PDF interpreters (such as Ghostscript) may well be able to deal with larger or smaller document formats. Although PDLflib will generate PDF documents with page sizes outside these limits, the default error handler will issue a warning message.

Table 3.1. Minimum and maximum page sizes supported by Acrobat 3 and 4

Acrobat 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 <sup>1</sup>	200" = 14400 pt = 508 cm

1. The documented limit for Acrobat 4 is 1/4" = 18 pt = 0.635 cm, but the above seems to be the real limit.

Common standard page size dimensions can be found in Table 3.2.<sup>1</sup> C macro definitions for these formats are available in *pdflib.h*.

Table 3.2. Common standard page sizes dimensions

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

### 3.3 Graphics and Text Handling

**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. 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. The interior is determined by one of two algorithms. 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.

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 as gray values between 0 and 1, or as 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).

**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. 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 4.3.3, »Path Segment Functions«) and its natural demise (i.e., one of the functions listed in Section 4.3.4, »Path Painting and Clipping Functions«):

- ▶ all functions listed in Section 4.3.1, »General Graphics State Functions« (e.g., changing line width or linecap)
- ▶ all functions listed in Section 4.3.2, »Special Graphics State Functions«

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>



- ▶ all functions listed in Section 4.4, »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«.

**Interactions among graphics and text functions.** At the beginning of each page, the text matrix is reset to the identity matrix, and the text position to the coordinate origin at (0,0). It is very important to understand that both text matrix and position are also reset to their respective defaults when one of the following PDFlib functions is called:

- ▶ all functions listed in Section 4.3.2, »Special Graphics State Functions«
- ▶ all functions listed in Section 4.3.3, »Path Segment Functions«
- ▶ all functions listed in Section 4.3.4, »Path Painting and Clipping Functions«
- ▶ The function `PDF_place_image()` for placing images

## 3.4 Font Handling

### 3.4.1 The PDF Core Fonts

PDF and Acrobat 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 platform encoding (see below). Using another encoding than the platform's default PDFlib 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 current platform 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*

### 3.4.2 Character Sets and 8-Bit Encoding

PDF supports several encoding methods (character sets) for 8-bit text fonts. PDFlib includes provisions for supporting diverse encoding vectors in the generated PDF output. The supported encoding vectors are referred to via symbolic names. Table 3.3 lists the symbolic encoding names supported in PDFlib (they are further described below). The supported encodings can be arbitrarily mixed in one document.

Table 3.3. Character encodings supported in PDFlib

Encoding	Description
<i>pdfdoc</i>	PDF's internal encoding which is similar to <i>winansi</i>
<i>winansi</i>	Windows encoding
<i>macroman</i>	Mac Roman encoding, i.e., the default MacOS character set
<i>builtin</i>	Original encoding used by non-text (symbol) or non-latin text fonts
<i>default</i>	<i>macroman</i> on the Mac, <i>winansi</i> on all other systems

**The pdfdoc encoding.** PDFDocEncoding (which is a superset of ISO 8859-1, also known as Latin 1) is shown in Figure 3.1 and plays a special role, since this encoding is always used for hypertext elements, such as bookmarks, annotations, or document information fields. For most clients this won't be much of a problem. However, since the Mac encoding substantially differs from PDFDocEncoding, it is necessary to convert Mac special characters to PDFDocEncoding when it comes to hypertext elements. For the same reason, it's impossible to incorporate several characters contained in the Mac character set in bookmarks.

**The winansi and macroman encodings.** These encodings reflect the »main« Windows character set and the MacOS character set, respectively. The exact definitions can be found in the respective header files in the PDFlib source file set, or in [1].

**The builtin encoding.** The encoding name *builtin* doesn't describe a particular character set 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), and non-latin text fonts (such as Greek and Cyrillic). Such fonts cannot be reencoded using one of the sup-

Fig. 3.1. The PDFDocEncoding character set as defined in PDF 1.3 (note the Euro character at position 0xA0)

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

ported encodings since their character names don't match those in these encodings. Therefore, *builtin* must be used for all symbolic or non-text fonts. 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 *pdfdoc*, *winansi*, and *macroman* encodings, while fonts with *FontSpecific* encoding can't, and must use *builtin* encoding instead. PDFlib will issue a warning message when an attempt is made to re-encode 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 default encoding.** Like *builtin*, the *default* encoding plays a special role since it doesn't refer to some fixed character set. Instead, *default* encoding will be mapped to *macroman* on the Mac, and *winansi* on all other systems. The *default* 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. Due to the nature of the *macroman* and *winansi* encoding, the *default* encoding is guaranteed to contain the usual ASCII character set in its lower half, while the upper half of the character set is platform-dependent. Apart from testing, these properties are only useful if the ASCII character set is sufficient for the application, e.g., plain English text without any special characters. Because of the varying contents of the upper half of the character set, *default* encoding is not recommended for more sophisticated applications.

*Note The preceding discussion relates to 8-bit encodings only. PDFlib also supports 16-bit Unicode encoding for hypertext elements as discussed in Section 3.4.5, »Unicode Support«.*

### 3.4.3 Font Outline and Metrics Files

**PDF font embedding.** PDF supports fonts outside the set of 14 core fonts in several ways. PDFlib is capable of embedding PostScript type 1 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 3.4 lists different situations with respect to font usage, each of which poses different requirements on the necessary font and metrics files. Currently, PDFlib supports the AFM (Adobe Font Metrics) file format for metrics information, and PFA (Printer Font ASCII) for PostScript Type 1 font outline information. PFA files must use Unix-style line-end conventions (NL = 0x0A as a line-end separator). There are several tools floating around the Internet for converting font and metrics file formats, most notably the Type 1 Utilities (*trutils*) for converting the PFB (Printer Font Bi-

Table 3.4. Different font usage situations and required metrics and outline files

Font usage	Required metrics files	Required font outline files
One of the 14 core fonts with PDFlib's default encoding <sup>1,2</sup>	None, since the platform metrics are built into the PDFlib binary	None, since the core font outlines are supplied by the Acrobat viewer
One of the 14 core fonts with an encoding other than PDFlib's default encoding <sup>2</sup>	The AFM core font metrics which are supplied with the PDFlib distribution	None, since the core font outlines are supplied by the Acrobat viewer
Non-core font which will not be embedded	Metrics file for the font	None
Non-core font which will be embedded	Metrics file for the font	Font outline file for the font

1. See Section 3.4.1, »The PDF Core Fonts« for a list of core fonts.  
2. See Section 3.4.2, »Character Sets and 8-Bit Encoding« for the definition of PDFlib's default encoding.

nary) font file format common on Windows systems to PFA, and the *pfm2afm* utility for converting Windows PFM (Printer Font Metrics) files to AFM format.

*Note* Future versions of PDFlib will support PFB and PFM files directly.

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.

**Use of TrueType fonts in PDF.** Although PDF technically supports embedded TrueType fonts, this is a murky area of PDF technology. The internals of TrueType and their behavior in PDF are not clearly documented, which in practise gives rise to several problems (such as missing characters or unsearchable text). For this reason, it is generally not possible to use a TrueType font in a PDF »by reference« (i.e., the PDF file doesn't contain the font, but only a font descriptor), and rely on Acrobat's use of installed TrueType fonts on the target system.

These problems are especially distinct for TrueType fonts which support multiple code pages. For these reasons PDFlib doesn't support the use of TrueType fonts in PDF (and you are well advised to avoid them in other PDF contexts, too).

**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 impose restrictions on the use of their fonts for non-print usage. 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 subsetting. Please check the legal issues of font embedding before attempting to embed fonts with PDFlib.

*Note* PDFlib currently doesn't implement font subsetting.

### 3.4.4 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. In addition to the static configuration file, dynamic

font 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 to extend the original UPR format for our purposes. The UPR file format as used by PDFlib will be described below.<sup>1</sup> There is an Adobe-supplied utility called *makespres* 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.
- ▶ 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.
- ▶ 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 resources described in the file. Each line describes one resource type. The list is terminated by a line with a single period character. Available resource types are described below.
- ▶ 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. Using the directory prefix a UPR file may, for example, point to some central PostScript font directory somewhere in the file system.
- ▶ A section for each of the resource categories listed at the beginning of the file. Each section starts with a line showing the resource category, followed by an arbitrary number of lines describing available resources. The list is terminated by a line with a single period character. Each resource data line contains the name of the resource (equal signs have to be quoted), an equal sign, and the corresponding relative or absolute file name for the resource. Relative file names will have the directory prefix (see above) applied. 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 3.5. Other resource categories may be present in the UPR file for compatibility with Display PostScript installations, but they will silently be ignored.

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>

Table 3.5. Resource categories supported in PDFlib

Resource type name	Explanation
FontAFM	PostScript font metrics file in AFM format
FontPFM	(reserved for future use)
FontOutline	PostScript font outline file
FontTT	(reserved for future use)

**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 one extra font:

```
PS-Resources-1.0
FontAFM
FontOutline

//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

FontOutline
Code-128=Code_128.pfa
.
```

**Searching for the UPR resource file.** If only the PDF core fonts with PDFlib's default encoding are to be used, a UPR configuration file is not necessary, since PDFlib contains all necessary font information built-in.

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

- ▶ The environment variable `PDFLIBRESOURCE` is examined and used as a resource file name, if set (this doesn't apply to the MacOS)
- ▶ If no file name is found, the client-settable *resource* 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 the file can't be opened, an *IOError* is raised.
- ▶ If the file can be opened, but a required resource category cannot be found, a *SystemError* is raised.

**Setting resources without a UPR file.** In addition to using a UPR file for font configuration, it is also possible to directly configure individual resources within the source code via the `PDF_set_parameter()` function. This function takes a category name and a corresponding resource entry as it would appear in the respective section of this category in a UPR resource file, for example:

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

### 3.4.5 Unicode Support

Starting with version 4, Acrobat fully supports Unicode text encoding. Unicode is a 16-bit character encoding scheme which covers all current and many ancient languages and scripts in the world.<sup>1</sup> PDFlib supports Unicode 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 the corresponding field names)
- ▶ Description and author of file attachments

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

- ▶ Unicode support is not available for the actual page descriptions but only for hyper-text elements as described above.
- ▶ 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 Unicode-enabled in their default configurations. Although Windows NT and MacOS support Unicode internally, availability of appropriate Unicode fonts is still an issue.<sup>2</sup>
- ▶ Acrobat doesn't seem to be able to handle more than one script in a single annotation. (It's currently unclear how »script« might be defined in this context – probably a Unicode codepage.)

In order to avoid duplicating all text-related API functions, PDFlib supports a dual-encoding approach with respect to all text strings supplied by the client for one of the above-mentioned Unicode-enabled functions. Unicode text can be processed by obeying the following principles:

- ▶ 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:

Hex (FE, FF) or 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.
- ▶ Since Unicode strings may contain null characters, the usual C convention for strings cannot be used. For this reason, Unicode strings must end with a Unicode null character, i.e., two null bytes.

1. More information on Unicode can be found at <http://www.unicode.org>.

2. For testing and producing screenshots we used a German version of Windows NT 4.0.

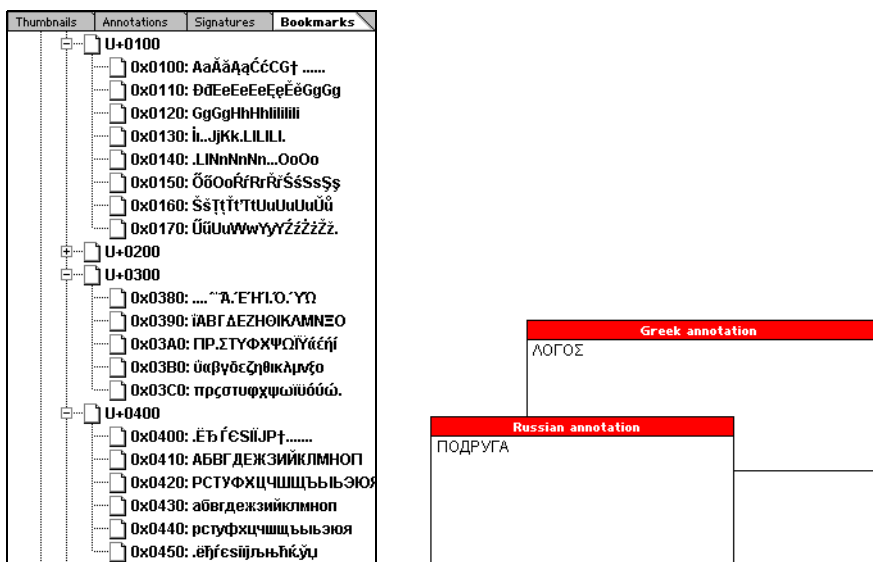


Fig. 3.2. Unicode bookmarks (left) and Unicode text annotations (right) as displayed in Acrobat 4 on Windows NT

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

```
\xFE\xFF\003\233\003\237\003\223\003\237\003\243\0\0
```

*Note Several languages among the PDFlib bindings natively support Unicode. However, the current PDFlib language bindings don't automatically convert from native language Unicode strings to PDFlib Unicode strings as described above.*

## 3.5 Image Handling

### 3.5.1 Image File Formats

Embedding raster images in the generated PDF is an important feature of PDFlib. PDFlib currently deals with the following image sources:

- ▶ **JPEG images:** All versions of PDF directly support the »baseline« flavor of JPEG compression which accounts for the vast majority of available JPEG files. In addition, Acrobat 4 and PDF 1.3 support progressive JPEG compression. PDFlib correctly deals with baseline and progressive JPEG images, but issues a warning message for progressive images since these are not compatible with Acrobat 3. PDFlib also applies a workaround which is necessary for correctly processing Photoshop-generated CMYK JPEG files.
- ▶ **GIF images:** PDFlib contains internal GIF handling code. Regular or interlaced GIFs may be used.
- ▶ **TIFF images:** Sam Leffler's TIFFlib can be plugged into PDFlib in order to support zillions of TIFF compression and encoding flavors. As driven by PDFlib, TIFFlib currently requires *width x height x 4* bytes of virtual memory for the decompressed image



data. Also, converted TIFF images always end up with 8 bit per color component per pixel, which is inadequate for black-and-white images. Both issues are expected to be resolved in future versions of PDFlib.

- ▶ CCITT images with raw Group 3 or Group 4 fax compressed image data.
- ▶ Raw uncompressed image data may be useful for some special applications, e.g., constructing a color ramp directly in memory.

If PDFlib is configured for compression, all image data will be output in the generated PDF in ZIP-compressed format (this behavior can be changed by using *debug* parameters). Notable exceptions are JPEG and CCITT images, which always retain their original compression scheme.

### 3.5.2 Embedding Images in PDF

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\_\**() functions return a handle which serves as an image descriptor. This handle can be used in a call to *PDF\_place\_image()*, along with positioning and scaling parameters:

```
if ((image = PDF_open_JPEG(p, "image.pdf")) == -1) {
    fprintf(stderr, "Error: Couldn't read image.\n");
} else {
    PDF_place_image(p, image, 0.0, 0.0, 1.0);
    PDF_close_image(p, image);
}
```

Note that the call to *PDF\_close\_image()* may or may not be required, depending on whether the same image will be used again (see below).

### 3.5.3 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 may account for enormous space savings.

### 3.5.4 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 some 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.

### 3.6 Error Handling

As described in Chapter 2, PDFlib provides a flexible mechanism for dealing with a certain kind of programming errors. Runtime errors in PDFlib applications fall into one of several classes, as shown in Table 3.6. Macro definitions for the error types are available in `pdflib.h`, and can be constructed by prefixing the error name with `PDF_` (e.g., `PDF_MemoryError`). The error handler will receive the kind of PDFlib error as an argument, and use the error type in its decision what to do.

Table 3.6. PDFlib runtime errors

Error name	Explanation
<i>MemoryError</i>	Not enough memory
<i>IOError</i>	Input/Output error, e.g. disk full
<i>RuntimeError</i>	Wrong order of PDFlib function calls
<i>IndexError</i>	Array index error
<i>TypeError</i>	Argument type error
<i>DivisionByZero</i>	Division by zero
<i>OverflowError</i>	Arithmetic overflow
<i>SyntaxError</i>	Syntactical error
<i>ValueError</i>	A value supplied as argument to PDFlib is invalid
<i>SystemError</i>	PDFlib internal error
<i>NonfatalError</i>	A problem was detected but processing continues
<i>UnknownError</i>	Other error

The opaque data pointer argument to `PDF_new2()` is useful for multi-threaded applications which may want to supply a handle to thread-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.

When the error handler is called, the PDF output file is still open. Client-supplied error handlers may wish to close the output file (if they are not responsible themselves anyway, which would be the case if they opened the PDF with `PDF_open_fp()`). Another important task of the error handler is to clean up PDFlib internals using `PDF_delete()` and the supplied pointer to the PDF object. PDFlib functions other than `PDF_delete()` should not be called from within a client-supplied error handler.

Client supplied error handlers are expected to not return to the library function which raised the exception. C programmers may achieve this by using the *setjump/longjump* facility.

# 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 discussed in appropriate notes.

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 \*) parameter which has to be supplied as the first argument to almost all PDFlib functions. Also, where the C API allows a NULL value for a string argument, an empty string "" may also be used to achieve the same effect from the scripting languages. Please refer to the respective language section in Chapter 2 for more language-specific details.

## 4.1 General Functions

**void PDF\_boot(void)**

Boot PDFlib. Recommended for the C language binding, although currently not necessarily required. For all other language bindings, booting is either not necessary, or accomplished automatically by the language binding mechanism.

**void PDF\_shutdown(void)**

Shut down PDFlib. Recommended for the C language binding, although currently not required.

**int PDF\_get\_majorversion(void)**

Returns the PDFlib major version number.

*Note This function is not available in the Java, Perl, Tcl, and Visual Basic bindings because these bindings supply their own versioning scheme.*

**int PDF\_get\_minorversion(void)**

Returns the PDFlib minor version number.

*Note This function is not available in the Java, Perl, Tcl, and Visual Basic bindings because these bindings supply their own versioning scheme.*

**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 used to refer 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. For example, in the Java binding the *long* data type is used. This doesn't really affect PDFlib clients, since all they have to do is to pass the PDF handle as the first argument to all functions.

*Note This function is not available in the C++ binding since it is hidden in the PDF constructor.*

```
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. When creating a new PDF object, 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.

*Note In the C++ binding this function is indirectly available via the PDF constructor. The function arguments need not be given since default values of NULL are supplied.*

```
void PDF_delete(PDF *p)
```

Delete a PDF object. This will free all 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.

*Note PDF\_delete() should also be called from client-supplied error handlers.*

*Note In the C++ binding this function is indirectly available via the PDF destructor.*

```
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.

```
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. The function returns -1 on error, and 1 otherwise.

*Note In the C++ binding this function is hidden in the overloaded PDF\_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.

The supplied file handle *fp* must have been opened for writing by the caller. It's important to note that for some platforms, most notably Windows, it is absolutely necessary to open the file in binary mode in order to prevent the C runtime library from messing with the line end characters which would result in corrupt PDF output. In order to open files in binary mode, use something like

```
fp = fopen("filename.pdf", "wb");
```

*Note This function is only available in the C binding. In the C++ binding, it is hidden in the overloaded PDF\_open() call.*

**void PDF\_close(PDF \*p)**  
Finish the generated PDF document, and close the output file if the PDF has been opened with *PDF\_open()*.

**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, »Coordinate Systems«. A list of commonly used page formats can be found in Table 3.2. Note that there are C convenience data structures for some common page formats (see Section 4.7, »Convenience Stuff«).

**void PDF\_end\_page(PDF \*p)**  
Must be used to finish a page description.

**void PDF\_set\_parameter(PDF \*p, const char \*key, const char \*value)**  
Set some PDFlib-internal parameters controlling PDF generation. Currently supported parameter keys and values are shown in Table 4.1.

Table 4.1. Keys and values for the PDFlib configuration parameters

Key	Values and explanation
any category name allowed in UPR files	The corresponding resource file line as it would appear for the respective category in a UPR file (see Section 3.4.4, »Resource Configuration and the UPR Resource File«)
resourcefile	Relative or absolute file name of the PDFlib resource file in UPR format. 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.
debug	<p>A string in which each character activates some debugging feature within PDFlib. The debugging options below are available (provided PDFlib has been built in a debugging configuration). All debugging options are off by default except where noted otherwise.</p> <p>a Produce ASCII instead of hex output for image and font data c Disable compression w Issue warning messages for non-fatal errors (on by default) m Memory allocation (malloc) r Memory allocation (realloc) f Memory allocation (free) s Print some statistics about the generated PDF document u Don't unlink (delete) PDF file on error</p>
nodebug	A string in which each character deactivates some debugging feature within PDFlib. See the debug table entry above for a list of supported characters.

## 4.2 Text Functions

### 4.2.1 Font Handling Functions

**int PDF\_findfont(PDF \*p, const char \*fontname, const char \*encoding, int embed)**  
Prepare a font with the supplied *encoding* for later use with *PDF\_setfont()*. The metrics will be loaded, and if *embed* has the value 1, the font file will be checked (but not yet used, since font embedding is done at the end of the generated PDF file). *encoding* is one of *builtin*, *pdfdoc*, *macroman*, *macexpert*, *winansi* or *default* (see Section 3.4.2, »Character

Sets and 8-Bit Encoding»). Note that in order to use arbitrary encodings, you will need a metrics file for the font (see Section 3.4.3, »Font Outline and Metrics Files«). If the return value is -1, the required files (metrics and possibly outline file) couldn't be successfully opened. In this case, the font cannot be used (at least not with the requested encoding and embedding settings). Otherwise, the return value can be used as font argument to other font-related functions.

*PDF\_findfont()* can safely be called outside of page descriptions.

**void PDF\_setfont(PDF \*p, int font, float fontsize)**

Set the current font in the given *fontsize*. The font descriptor must have been retrieved via *PDF\_findfont()*. This function must only be called within a page description.

**const char \*PDF\_get\_fontname(PDF \*p)**

Return 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.

**float PDF\_get\_fontsize(PDF \*p)**

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.

**int PDF\_get\_font(PDF \*p)**

Return the identifier of the current font which must have been previously set with *PDF\_setfont()*. This function must only be called within a page description.

## 4.2.2 Text Output Functions

**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 point (via *PDF\_moveto()*) must have been set before.

**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.

**void PDF\_continue\_text(PDF \*p, const char \*text)**

Move to the next line (as determined by the leading parameter, see *PDF\_set\_leading()*) and print *text*.

**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()*.

**void PDF\_set\_leading(PDF \*p, float leading)**

Set the leading, which is the distance between baselines of adjacent lines of text. The leading parameter is set to the default value of 0 at the beginning of each page.

**void PDF\_set\_text\_rise(PDF \*p, float rise)**

Set the text rise parameter to a value of *rise* units. The text rise parameter specifies the distance between the desired text position and the default baseline. Positive values of

text rise move the baseline up. This may be useful for superscripts and subscripts. The text rise parameter is set to the default value of 0 at the beginning of each page.

**void PDF\_set\_horiz\_scaling(PDF \*p, float scale)**

Set the horizontal text scaling to a value of *scale* percent. Text scaling shrinks or expands the text by a given percentage. The text scaling parameter is set to the default value of 100 at the beginning of each page.

**void PDF\_set\_text\_rendering(PDF \*p, int mode)**

Set the current text rendering mode to one of the values given in Table 4.2. The text rendering parameter is set to the default value of 0 (= solid fill) at the beginning of each page.

*Note Invisible text doesn't appear on the page, but can be searched and indexed. This may be useful for attaching OCR'd text to scanned pages in order to make the contents searchable.*

Table 4.2. Values for the text rendering mode

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

**void PDF\_set\_text\_matrix(PDF \*p, float a, float b, float c, float d, float e, float f)**

Set the text matrix which describes a transformation to be applied to the current text font, e.g. for skewing the text. The text matrix is set to the default identity matrix (1, 0, 0, 1, 0, 0) at the beginning of each page, and when certain PDFlib functions are called (see Section 3.3, »Graphics and Text Handling«). The six floating point values make up the matrix in the same way as in PostScript and PDF (see references).

**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, and when certain PDFlib functions are called (see Section 3.3, »Graphics and Text Handling«).

**void PDF\_set\_char\_spacing(PDF \*p, float spacing)**

Set the character spacing value, i.e., the horizontal shift of the current point after placing the individual characters in a string. The *spacing* value is given in text space units. It is reset to the default of 0 at the beginning of a new page.

**void PDF\_set\_word\_spacing(PDF \*p, float spacing)**

Set the word *spacing* value, i.e., the horizontal shift of the current point after individual words in a text line. In other words, the current point is moved by *spacing* units horizon-



tally after each space character (0x20). The spacing value is given in text space units. It is reset to the default value of 0 at the beginning of a new page.

## 4.3 Graphics Functions

### 4.3.1 General Graphics State Functions

*Note* Don't use general graphics state functions within a path description (see Section 3.3, »Graphics and Text Handling«).

**void PDF\_setdash(PDF \*p, float b, float w)**  
Set the current dash pattern to *b* black and *w* white units. In order to produce a solid line, choose *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. In order to produce a solid line, choose *length* = 0 and *darray* = NULL. The dash parameter is set to a solid line at the beginning of each page.

**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 between 0 and 2 inclusive. The *linejoin* parameter specifies the shape at the corners of paths that are stroked, as shown in Table 4.3. The linejoin parameter is set to the default value of 0 at the beginning of each page.




Table 4.3. Values of the linejoin parameter

Value	Meaning
0	miter joins
1	round joins
2	bevel joins

**void PDF\_setlinecap(PDF \*p, int linecap)**  
Set the linecap parameter to a value between 0 and 2 inclusive. The linecap parameter controls the shape at the ends of open paths with respect to stroking, as shown in Table 4.4. 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 4.4. Values of the linecap parameter

Value	Meaning	Example
0	butt end caps	
1	round end caps	
2	projecting square end caps	

**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.

**void PDF\_set\_fillrule(PDF \*p, const char\* 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 need to change the fill rule. The fill rule is reset to the default value of *winding* at the beginning of each page.

### 4.3.2 Special Graphics State Functions

All graphics state parameters are restored to their default values at the beginning of a page. The default values are documented in the respective function descriptions. Functions related to the text state are listed in Section 4.2, »Text Functions«.

*Note All special graphics state functions reset the text position and matrix.*

**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 graphics state later (e.g., a custom coordinate system) without setting all relevant parameters explicitly again.

**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.

*Note Although PDF doesn't impose any limit on the nesting level of save/restore pairs, applications are strongly advised to keep the nesting level below 12 in order to avoid printing problems caused by restrictions in the PostScript output produced by PDF viewers.*

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

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

*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.*

```
void PDF_rotate(PDF *p, float phi)
```

Rotate the user coordinate system by *phi* degrees.

### 4.3.3 Path Segment Functions

*Note All path segment functions reset the text position and matrix.*

```
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.

```
void PDF_lineto(PDF *p, float x, float y)
```

Draw a line from the current point to  $(x, y)$ .

```
void PDF_curveto(PDF *p,  
float x1, float y1, float x2, float y2, float x3, float y3)
```

Draw a Bézier curve from the current point to  $(x_3, y_3)$ , using  $(x_1, y_1)$  and  $(x_2, y_2)$  as control points.

```
void PDF_circle(PDF *p, float x, float y, float r)
```

Draw a circle with center  $(x, y)$  and radius  $r$ .

```
void PDF_arc(PDF *p, float x, float y, float r, float alpha1, float alpha2)
```

Draw a circular arc with center  $(x, y)$ , radius  $r$ , extending from *alpha1* to *alpha2* degrees.

```
void PDF_rect(PDF *p, float x, float y, float width, float height)
```

Draw a rectangle with lower left corner  $(x, y)$  and the supplied *width* and *height*.

```
void PDF_closepath(PDF *p)
```

Close the current path, i.e. draw a line from the current point to the starting point of the path.

### 4.3.4 Path Painting and Clipping Functions

*Note All path painting and clipping functions reset the text position and matrix.*

```
void PDF_stroke(PDF *p)
```

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

```
void PDF_closepath_stroke(PDF *p)
```

Close the current path and stroke it with the current line width and the current stroke color. This operation clears the path.

```
void PDF_fill(PDF *p)
```

Fill the interior of the current path with the current fill color. The interior of the path is determined by one of two algorithms (see *PDF\_setfillrule()*). Open paths are implicitly closed before being filled

```
void PDF_fill_stroke(PDF *p)
```

Fill and stroke the path with the current fill and stroke color, respectively.

**void PDF\_closepath\_fill\_stroke(PDF \*p)**

Close the path, fill, and stroke it.

**void PDF\_endpath(PDF \*p)**

End the current path.

**void PDF\_clip(PDF \*p)**

Use the current path as the clipping path. The clipping path is set to the default value of the page size at the beginning of each page.

## 4.4 Color Functions

*Note Don't use color functions within a path description (see Section 3.3, »Graphics and Text Handling«).*

**void PDF\_setgray\_fill(PDF \*p, float g)**

Set the current fill color to a gray value with  $0 \leq g \leq 1$ . The gray fill parameter is set to the default value of  $0 = \text{black}$  at the beginning of each page.

**void PDF\_setgray\_stroke(PDF \*p, float g)**

Set the current stroke color to a gray value with  $0 \leq g \leq 1$ . The gray stroke parameter is set to the default value of  $0 = \text{black}$  at the beginning of each page.

**void PDF\_setgray(PDF \*p, float g)**

Set the current fill and stroke color to a gray value with  $0 \leq g \leq 1$ . The gray parameter is set to the default value of  $0 = \text{black}$  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  $(0, 0, 0) = \text{black}$  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  $(0, 0, 0) = \text{black}$  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  $(0, 0, 0) = \text{black}$  at the beginning of each page.

## 4.5 Image Functions

*Note Not all file formats may be supported by a particular PDFlib implementation.*

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

```
int PDF_open_JPEG(PDF *p, const char *filename)
int PDF_open_TIFF(PDF *p, const char *filename)
int PDF_open_GIF(PDF *p, const char *filename)
```

Open and analyze a raster graphics file in one of the supported file formats. The returned image handle, if not -1, may be used in subsequent image-related calls.

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.

```
int PDF_open_CCITT(PDF *p,
const char *filename, int width, int height, int BitReverse, int K, int BlackIs1)
Open a CCITT G3 or G4 compressed bitmap file. 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 (the latter is untested yet).
- 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.

```
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 3.5.1, »Image File Formats«); the *source* parameter denotes where the image data comes from, and can attain the values *fileref*, *url*, or *memory* (see Section 3.5.4, »Memory Images and External Image References«). The relationship among the *source*, *data*, and *length* parameters is explained in Table 4.5.

Table 4.5. Values of the *source*, *data*, and *length* parameters of *PDF\_open\_image()*

source	data	length
fileref	string with a platform-independent image file name (see [1])	unused, should be 0
url	string with an image URL conforming to RFC 1738	unused, should be 0
memory	pointer to (or string containing) image data; the image data is compressed according to the <i>type</i> parameter	length of (compressed) image data in memory. If <i>type</i> is <i>raw</i> , <i>length</i> must be equal to <i>width</i> x <i>height</i> x <i>components</i>

The *width* and *height* parameters describe the dimensions of the image. The number of color *components* must be 1, 3, or 4. The number of bits per component *bpc* must be 1, 2, 4, or 8. *width*, *height*, *components*, and *bpc* must always be supplied.

*params* is only used if *type* has the value *ccitt*, and must be NULL or empty otherwise. 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 a NULL or 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.

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.

```
int PDF_get_image_width(PDF *p, int image)
```

Return the width of an image in pixels.

```
int PDF_get_image_height(PDF *p, int image)
```

Return the height of an image in pixels.

```
void PDF_close_image(PDF *p, int image)
```

Close the image. This only affects PDFlib's associated internal image structure. The actual image file is not affected by this call since it has already been closed at the end of the corresponding *PDF\_open\_\**() 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.

```
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 current page, and the image is scaled by the supplied scaling factor. Non-uniform scaling may be achieved with *PDF\_scale()*, optionally bracketing the sequence with *PDF\_save()* and *PDF\_restore()*.

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 on page descriptions, i.e. between *PDF\_begin\_page()* and *PDF\_end\_page()*.

*Note* This function resets the text position and matrix.

## 4.6 Hypertext Functions

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

### 4.6.1 Bookmarks

```
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()*.

The 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.

### 4.6.2 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 four standard information field names, or up to one custom field name (see Table 4.6). If a custom field name is used, it must consist of printable ASCII characters except any of the following: blank ' ', %, (, ), <, >, [, ], {, }, /, and #.

Table 4.6. 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
(any custom name)	User-defined field name. PDFlib supports one additional document information field which may be arbitrarily named.

### 4.6.3 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 of a particular Acrobat installation. PDFlib, however, allows setting the page transition mode and duration for each page separately.

```
void PDF_set_transition(PDF *p, const char *type)
```

Set a transition effect for the current page. Set the page transition effect for the current and any subsequent pages until another call to *PDF\_set\_transition()*. The transition type strings given in Table 4.7 are supported. *type* may also be NULL to reset the transition effect. The default transition is *replace*, i.e., no special transition effect.

```
void PDF_set_duration(PDF *p, float t)
```

Set the page display duration in seconds for the current page. The default duration is one second.

Table 4.7. Values for the transition type





Key	Explanation
<i>split</i>	Two lines sweeping across the screen reveal the page
<i>blinds</i>	Multiple lines sweeping across the screen reveal the page
<i>box</i>	A box reveals the page
<i>wipe</i>	A single line sweeping across the screen reveals the page
<i>dissolve</i>	The old page dissolves to reveal the page
<i>glitter</i>	The dissolve effect moves from one screen edge to another
<i>replace</i>	The old page is simply replaced by the new page (default)

### 4.6.4 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 annota-
tion is activated. The icon parameter controls the display of the unopened file attach-
ment in Acrobat, as shown in Table 4.8.
```

*Note* PDF file attachments are only supported in Acrobat 4. 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.

Table 4.8. Values for the icon name for file attachments

Key	attachment icon	Key	attachment icon
<i>graph</i>		<i>pushpin</i>	
<i>paperclip</i>		<i>tag</i>	

### 4.6.5 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 cor-
ners in default user space coordinates. contents and title may be encoded with PDF-
DocEncoding or Unicode. The icon parameter controls the display of the unopened note
attachment in Acrobat, as shown in Table 4.9. The annotation will be opened if open = 1,
and closed if open = 0.
```

*Note* Different note icons are only available in Acrobat 4. Acrobat 3 viewers (and apparently Unix versions of Acrobat 4) will display the »note« type icon regardless of the supplied icon parameter.



*Note* Line breaks in note annotations are not reliably displayed in all PDF viewers (most notably Acrobat 4.0 for Windows NT).

Table 4.9. Values for the icon name for note annotations

Key	note annotation icon	Key	note annotation icon
comment		newparagraph	
insert		key	
note		help	
paragraph			

4.6.6 Links

`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 4.10.

`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 file 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 future page which does not yet exist. However, the application must make sure that the target page will actually be generated; PDFlib will issue a warning message otherwise. *dest* specifies the destination zoom. It can attain one of the values specified in Table 4.10.

Table 4.10. Values for the *dest* parameter of `PDF_add_pdflink()` and `PDF_add_locallink()`

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*.

`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 d1, float d2)`  
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*.

## 4.7 Convenience Stuff

`<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()`.

*Note* These values are only supplied for the C and C++ bindings. Other language clients may use the values provided in Table 3.2.

# 5 The PDFlib License

PDFlib is subject to the »Aladdin Free Public License«.<sup>1</sup> The complete text of the license agreement can be found in the file *license.pdf*. In short and non-legal terms:

- ▶ You may use and distribute PDFlib non-commercially.
- ▶ You may develop free software with PDFlib.
- ▶ You may develop software for your own use with PDFlib.
- ▶ You may not sell any software based on PDFlib without obtaining a commercial license.

Note that this is only a 10-second-description which is not legally binding. Only the text in the *license.pdf* file is considered to completely describe the licensing conditions.

A commercial PDFlib license is required for all uses of the software which are not explicitly covered by the Aladdin Free Public License. Commercial licensees will benefit from the following advantages:

- ▶ A license agreement detailing the terms of use
- ▶ Technical support
- ▶ Prioritized handling of change and feature requests

Please contact the author for details on obtaining a commercial PDFlib license:

Thomas Merz  
Consulting & Publishing  
Tal 40  
80331 München, Germany  
<http://www.ifconnection.de/~tm>  
[tm@muc.de](mailto:tm@muc.de)  
fax +49/89/29 16 46 86

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

## 6 References

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

[1] Adobe Systems Inc.: Portable Document Format Reference Manual, Version 1.3.  
Available from <http://partners.adobe.com/asn/developer/PDFS/TN/PDFSPEC.PDF>

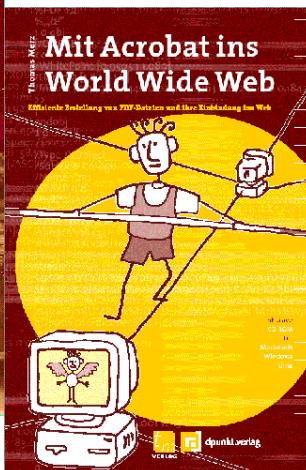
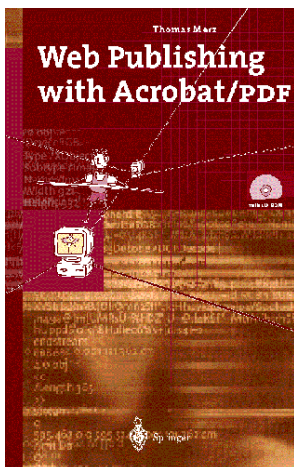
[2] Adobe Systems Inc.: PostScript Language Reference Manual, third edition.  
Published by Addison-Wesley, ISBN 0-201-37922-8, also available from  
<http://partners.adobe.com/asn/developer/PDFS/TN/PLRM.pdf>

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

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

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

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



# Index

## O-9

16-bit encoding 39  
8-bit encodings 33

## A

Acrobat 4 compatibility 10  
Adobe Font Metrics (AFM) 35  
AdobeStandardEncoding 35  
AFM (Adobe Font Metrics) 35  
Aladdin free public license 59  
annotations 39, 56  
API (Application Programming Interface)  
    reference 44  
attachments 39, 56  
availability of PDFlib 12

## B

baseline compression 40  
Beazley, Dave 12  
binary mode 45  
bindings 11  
BitReverse 53  
BlackIs1 53  
BOM (Byte Order Mark) 39  
bookmarks 39, 55  
builtin encoding 33  
Byte Order Mark 39

## C

C binding  
    error handling 16  
    general 15  
    memory management 16  
    version control 17  
C++ binding  
    error handling 19  
    general 17  
    memory management 19  
    version control 19  
categories of resources 38  
CCITT 41, 53  
character sets 33  
clip 32  
color 32  
color functions 52  
commercial license 59  
compatibility  
    Acrobat 4 10

    Acrobat Reader 9  
    configuration parameters 46  
    convenience stuff 58  
    coordinate system 31, 50  
    coordinate systems 31  
    core fonts 33  
    current point 32

## D

debug parameter 46  
default coordinate system 31  
default encoding 33  
descriptor 35  
document information fields 39, 55

## E

embedding fonts 35  
encoding 33  
error handling 42  
    API 45  
    error names 42  
    general 13  
    in C 16  
    in C++ 19  
    in Java 21  
    in Perl 23  
    in Python 25  
    in Tcl 28  
    in Visual Basic 30  
Euro character 34  
external image references 41

## F

features of PDFlib 8  
file attachments 39, 56  
fill 32  
fonts  
    descriptor 35  
    embedding 35  
    general 33  
    legal aspects of embedding 36  
    metrics files 35  
    outline files 35  
    PDF core set 33  
    pfm2afm 36  
    PostScript 35  
    resource configuration 36  
    TrueType 36  
    type 1 35

type 1 utilities 35  
Unicode support 39  
FontSpecific encoding 35

## G

general graphics state 49  
GIF 40, 53  
graphics functions 49  
graphics state 49, 50

## H

hello world example  
  general 13  
  in C 15  
  in C++ 18  
  in Java 20  
  in Perl 23  
  in Python 25  
  in Tcl 27  
  in Visual Basic 29  
hypertext functions 54

## I

icons  
  for file attachments 56  
  for notes 57  
image data, re-using 41  
image file formats 40  
image functions 52  
image references 41  
ISO 8859-1 34

## J

Java binding  
  error handling 21  
  general 19  
  memory management 21  
  version control 21  
JPEG 40, 53

## K

K parameter for CCITT images 53

## L

language bindings: see bindings  
Latin 1 encoding 34  
leading 47  
licensing conditions 59  
links 57  
longjump 43

## M

macroman encoding 33

makepsres utility 37  
memory images 41  
memory management  
  API 45  
  general 14  
  in C 16  
  in C++ 19  
  in Java 21  
  in Perl 24  
  in Python 25  
  in Tcl 28  
  in Visual Basic 30

## N

nodebug parameter 46  
note annotations 39, 56

## O

ordering constraints 32

## P

page size definitions 32  
page size formats 58  
page size limitations in Acrobat 31  
page transitions 55  
parameters 46  
path 32  
path painting and clipping 51  
path segment functions 51  
PDF 1.3 9, 34  
PDF\_add\_bookmark() 55  
PDF\_add\_launchlink() 57  
PDF\_add\_locallink() 57  
PDF\_add\_note() 56  
PDF\_add\_pdflink() 57  
PDF\_add\_weblink() 58  
PDF\_arc() 51  
PDF\_attach\_file() 56  
PDF\_begin\_page() 46  
PDF\_boot() 44  
PDF\_circle() 51  
PDF\_clip() 52  
PDF\_close() 46  
PDF\_close\_image() 54  
PDF\_closepath() 51  
PDF\_closepath\_fill\_stroke() 52  
PDF\_closepath\_stroke() 51  
PDF\_continue\_text() 47  
PDF\_curveto() 51  
PDF\_delete() 45  
PDF\_end\_page() 46  
PDF\_endpath() 52  
PDF\_fill() 51  
PDF\_fill\_stroke() 51  
PDF\_findfont() 46  
PDF\_get\_font() 47

- PDF\_get\_fontname() 47
- PDF\_get\_fontsize() 47
- PDF\_get\_image\_height() 54
- PDF\_get\_image\_width() 54
- PDF\_get\_majorversion() 44
- PDF\_get\_minorversion() 44
- PDF\_get\_opaque() 45
- PDF\_lineto() 51
- PDF\_moveto() 51
- PDF\_new() 44
- PDF\_new2() 45
- PDF\_open\_CCITT() 53
- PDF\_open\_file() 45
- PDF\_open\_fp() 45
- PDF\_open\_GIF() 53
- PDF\_open\_image() 42, 53
- PDF\_open\_JPEG 53
- PDF\_open\_TIFF() 53
- PDF\_place\_image() 54
- PDF\_rect() 51
- PDF\_restore() 50
- PDF\_rotate() 51
- PDF\_save() 50
- PDF\_scale() 50
- PDF\_set\_border\_color() 58
- PDF\_set\_border\_dash() 58
- PDF\_set\_border\_style() 58
- PDF\_set\_char\_spacing() 48
- PDF\_set\_duration() 55
- PDF\_set\_fillrule() 50
- PDF\_set\_horiz\_scaling() 28
- PDF\_set\_info() 55
- PDF\_set\_leading() 47
- PDF\_set\_parameter() 39
- PDF\_set\_parameter() 46
- PDF\_set\_text\_matrix() 48
- PDF\_set\_text\_pos() 48
- PDF\_set\_text\_rendering() 48
- PDF\_set\_text\_rise() 47
- PDF\_set\_transition() 55
- PDF\_set\_word\_spacing() 48
- PDF\_setdash() 49
- PDF\_setflat() 49
- PDF\_setfont() 47
- PDF\_setgray() 52
- PDF\_setgray\_fill() 52
- PDF\_setgray\_stroke() 52
- PDF\_setlinecap() 49
- PDF\_setlinejoin() 49
- PDF\_setlinewidth() 50
- PDF\_setmiterlimit() 49
- PDF\_setpolydash() 49
- PDF\_setrgbcolor() 52
- PDF\_setrgbcolor\_fill() 52
- PDF\_setrgbcolor\_stroke() 52
- PDF\_show() 47
- PDF\_show\_xy() 47
- PDF\_shutdown() 44

- PDF\_stringwidth() 47
- PDF\_stroke() 51
- PDF\_translate() 50
- pdfdoc encoding 33
- PDFDocEncoding 34
- PDFlib
  - features 8
  - features not implemented 10
  - program structure 31
  - thread-safety 9, 15
- pdflib.upr 38
- PDFLIBRESOURCE variable 38
- Perl binding
  - error handling 23
  - general 22
  - memory management 24
  - version control 24
- PFA (Printer Font ASCII) 35
- PFB (Printer Font Binary) 35
- pfm2afm 36
- PHP3 7
- platforms 12
- Portable Document Format Reference Manual 60
- PostScript fonts 35
- PostScript Language Reference Manual 60
- Printer Font ASCII (PFA) 35
- Printer Font Binary (PFB) 35
- program structure 31
- Purify 9
- Python binding
  - error handling 25
  - general 24
  - memory management 25
  - version control 26

## R

- raster images
  - functions 52
  - general 40
- raw image data 41
- raw images 53
- references 60
- resource category 38, 46
- resource parameter 38
- resourcefile parameter 46
- restrictions of PDFlib 10
- RGB color 32

## S

- scripting API 12
- setjump 43
- special graphics state 50
- standard page sizes 32
- stroke 32
- structure of PDFlib programs 31
- subscript 48
- superscript 48

SWIG (*Simplified Wrapper and Interface Generator*) 12

## T

*t1utils* 35

*Tcl binding*

*error handling* 28

*general* 26

*memory management* 28

*version control* 28

*text functions* 46

*text rendering modes* 48

*thread-safety* 9, 15

*TIFF* 40, 53

*TIFFlib* 40

*TrueType fonts* 36

*type 1 fonts* 35

*type 1 utilities* 35

*type library* 28

## U

*Unicode* 39

*UPR (Unix PostScript Resource)* 36

*dynamic update* 46

*file format* 37

*file searching* 38

*URL* 42, 58

*user space* 31

## V

*version control*

*general* 14

*in C* 17

*in C++* 19

*in Java* 21

*in Perl* 24

*in Python* 26

*in Tcl* 28

*in Visual Basic* 30

*Visual Basic binding*

*error handling* 30

*general* 28

*memory management* 30

*type library* 28

*version control* 30

## W

*weblink* 58

*winansi encoding* 33